In-Network Data Processing using FPGAs

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

DAVID ANDREAS SIDLER

Master of Science ETH in Computer Science, ETH Zurich

born on 16.03.1988

citizen of Küsnacht, SZ, Switzerland

accepted on the recommendation of

Prof. Dr. Gustavo Alonso (ETH Zurich), examiner
Prof. Dr. Timothy Roscoe (ETH Zurich), co-examiner
Prof. Dr. Ankit Singla (ETH Zurich), co-examiner
Dr. Ken Eguro (MSR Redmond), co-examiner

2019
In recent years, network bandwidth has increased at a rapid pace, moving from 10G, to 40G/100G, to 400G. CPU frequency on the other hand has been stagnant. As a result, more and more CPU cycles have to be allocated to network processing to keep up with the increasing network bandwidth. At the same time, we have seen exponential growth in data requiring distributed systems to process it. While these systems benefit from the increased network bandwidth, their ability to scale out is limited by inefficiencies in I/O and data movement. Under the premise that most of such computation is distributed, one question remains as to whether data processing operations can be offloaded to the network. In this thesis, we explore two approaches to enable in-network data processing and thereby reduce burden on the CPU.

Our first approach places data processing accelerators, such as FPGAs, in the network. To enable network-attached accelerators, we introduce a scalable network stack architecture for FPGAs supporting thousands of concurrent connections, a requirement of data center applications. Based on this architecture, we implement a TCP/IP stack that allows the seamless integration of accelerators into the existing network infrastructure.

In our second approach, we enable data processing operations in the network by enhancing the functionality of the network card. In particular, we present Smart Remote Memory (STROM), which extends Remote Direct Memory Access (RDMA) with the capability to deploy acceleration kernels on the network card. These acceleration kernels can, for instance, directly access the remote host memory to traverse remote data structures such as lists or indexes. In addition to that, they can execute stream-based operations, e.g., filtering or aggregation, on data that is passing through the network card to the host memory, or vice versa.

As part of this thesis, we implemented and released an open source TCP/IP stack for
reconfigurable hardware. The stack enables network-attached accelerators and has been deployed successfully in a number of research and commercial systems. Following on from this success, we released an open source FPGA-based RDMA NIC, which we used as a prototyping platform for STROM, to facilitate more research in this area.
Zusammenfassung


Unser erster Ansatz platziert spezialisierte Datenverarbeitungsgeräte wie FPGAs, nahe am Netzwerk, indem sie direkt an das Netzwerk angeschlossen werden. Um dies zu ermöglichen, stellen wir eine skalierbare Netzwerk-Stack-Architektur für FPGAs vor. Die Architektur kann Tausende von Verbindungen gleichzeitig unterstützen, was eine Voraussetzung für Anwendungen im Rechenzentrum ist. Basierend auf dieser Architektur implementieren wir einen TCP/IP-Stack, wodurch spezialisierte Datenverarbeitungsgeräte nahtlos in die vorhandene Netzwerkinfrastruktur integriert werden können.

Mit unserem zweiten Ansatz, genannt Smart Remote Memory (STROM), ermöglichen wir die Verarbeitung von Daten direkt im Netzwerk, indem wir die Funktionalität der Netzwerkkarte erweitern. Insbesondere den Speicherdirektzugriff über das Netzwerk (Remote Direct Memory Access) erweitern wir mit der Möglichkeit, Datenverarbeitungsprogramme auf der Netzwerkkarte zu installieren und auszuführen. Diese Programme können beispiel-
sweise direkt auf den Speicher eines entfernten Rechners zugreifen, um Datenstrukturen wie Listen oder Indizes auszulesen. Darüber hinaus können die Datenströme vom Netzwerk zum Arbeitsspeicher oder in entgegengesetzter Richtung durch Operationen wie, z.B. Selektion oder Aggregation, verarbeitet werden.

Acknowledgments

First of all, I would like to express my gratitude to my advisor Gustavo Alonso for his advice on my master’s and doctoral studies. He granted me a lot of freedom, always supported me, and had patience when I hit a roadblock. I want to thank Ankit Singla, and Timothy Roscoe for agreeing to be on my PhD committee and for their insightful feedback on my work. Likewise, I thank Ken Eguro for mentoring me during my internship, for giving me advice on my doctoral studies, and for being on my PhD committee. His feedback has significantly improved the quality of my dissertation.

I would like to thank Michaela Blott for mentoring me as a young researcher and introducing me to high-level synthesis and network processing, both of which built the base for the work presented in this dissertation. I also want to thank Aleksandar Dragojevic and Junyi Liu for giving me the opportunity to work on project Honeycomb.

I am grateful for having Zsolt as a colleague and friend. Our many fruitful collaborations were a pleasure.

Furthermore, I want to thank my colleagues in the System Group that made it a pleasure to be part of the group: Pratanu, Muhsen, Kaan, Johannes, Renato, Claude, Zeke, Zhenhao, Fabio, Dario, and Monica.

Finally, I thank Chimedtseyen and my family for their support and love.
# Contents

1 Introduction ................................................. 1
   1.1 Problem Statement ........................................ 5
   1.2 Contributions ............................................ 6
   1.3 Thesis Outline ............................................ 7
   1.4 Related Publications ..................................... 8

2 Field Programmable Gate Array ........................... 11
   2.1 Architecture .............................................. 11
   2.2 FPGA Design Flow ......................................... 12
   2.3 Programming .............................................. 14
   2.4 System Integration ......................................... 14
   2.5 High-Level Synthesis (HLS) ............................... 16

3 Scalable Network Stack Architecture .................... 19
   3.1 Architecture .............................................. 21
   3.2 Scalability ................................................. 23
      3.2.1 Scaling Connections ................................. 23
      3.2.2 Scaling Data Path ................................... 24
   3.3 Data Structures ........................................... 25
      3.3.1 Hash Table ............................................ 25
4.4 Optimizations ................................................................. 55
  4.4.1 Challenges in Practice .............................................. 56
  4.4.2 Assumptions ............................................................. 56
  4.4.3 Nagle's algorithm ...................................................... 57
  4.4.4 Delayed Acknowledgment .......................................... 59
  4.4.5 Retransmission- & Probe-Timers ................................. 60
  4.4.6 On-chip RX buffer ................................................... 61
  4.4.7 Reducing memory access to the TX buffer ...................... 61
4.5 Evaluation ................................................................. 62
  4.5.1 Throughput ............................................................. 62
  4.5.2 Latency ................................................................. 65
  4.5.3 Reducing DDR memory access ................................. 66
  4.5.4 Resources ............................................................. 67
4.6 Enabling Applications on the FPGA .................................. 68
  4.6.1 cloudFPGA ............................................................ 68
  4.6.2 Caribou ............................................................... 69
  4.6.3 Memcached Accelerator ............................................. 71
4.7 Related Work ............................................................. 73
4.8 Summary and Discussion .............................................. 74

5 FPGA-based RDMA Network Card ..................................... 77
  5.1 Background .............................................................. 78
    5.1.1 Remote Direct Memory Access (RDMA) ....................... 78
    5.1.2 RDMA over Converged Ethernet (RoCE) .................... 79
    5.1.3 DMA on FPGA ..................................................... 80
  5.2 Architecture ............................................................ 81
    5.2.1 DMA Engine and Driver ........................................ 82
    5.2.2 Memory Management ............................................ 83
## Contents

5.2.3 Software-Hardware Interaction .............................................. 83  
5.3 RoCE Network Stack .............................................................. 84  
  5.3.1 Receiving Data Path ......................................................... 85  
  5.3.2 Transmitting Data Path ..................................................... 86  
  5.3.3 State-keeping Data Structures .......................................... 86  
  5.3.4 Scalability ........................................................................ 87  
5.4 Evaluation ............................................................................... 87  
  5.4.1 Microbenchmark: DMA ....................................................... 88  
  5.4.2 Latency ............................................................................ 89  
  5.4.3 Latency Breakdown ............................................................ 91  
  5.4.4 Throughput ....................................................................... 91  
  5.4.5 Resources ......................................................................... 92  
5.5 Summary and Discussion .......................................................... 93  

6 Smart Remote Memory .................................................................. 95  
  6.1 Background ........................................................................... 97  
    6.1.1 Communication Models .................................................. 97  
    6.1.2 Programmable NICs and Switches ..................................... 97  
  6.2 STROM - Design Decisions ...................................................... 98  
    6.2.1 Extending the capabilities of RDMA ................................. 98  
    6.2.2 Low Power CPU vs Programmable Fabric ....................... 99  
    6.2.3 Programmability ............................................................. 100  
    6.2.4 Capabilities .................................................................... 100  
  6.3 STROM - Implementation ....................................................... 101  
    6.3.1 Protocol ......................................................................... 101  
    6.3.2 STROM Kernel ............................................................... 103  
    6.3.3 Software Integration/API .................................................. 107  
  6.4 Use Cases for STROM ............................................................ 107
### Contents

6.4.1 Traversing Remote Data Structures ........................................ 107  
6.4.2 Data consistency check ......................................................... 115  
6.4.3 Data Shuffling ................................................................. 117  
6.4.4 Resources ................................................................. 122  
6.5 Related Work ................................................................. 122  
6.6 Summary and Discussion ....................................................... 123

7 Conclusions ........................................................................... 125  
7.1 Summary ................................................................. 125  
7.2 Research Outlook ............................................................... 126
In this thesis, we address the growing discrepancy between network bandwidth and CPU frequencies. Network bandwidth has been increasing by two orders of magnitude since 2002, while single-threaded CPU frequencies have been stagnating. This discrepancy means that more and more CPU cycles have to be allocated to network processing. Combined with the exponential growth of data collected and generated, there is an urge to find novel approaches that can process data efficiently and promptly at scale. The approach explored in this thesis moves data processing closer to or into the network, taking advantage of the high bandwidth while avoiding the inefficiencies of the von Neumann architecture of the CPU.

Network Bandwidth Growth

In recent years, network bandwidth has increased drastically, starting in 2002 with the transition from 1 G Ethernet to 10 G Ethernet, followed by the introduction of 40 G and 100 G Ethernet in 2010, and most recently with the first vendors starting to ship 400 G network equipment. During the same period, single-threaded CPU frequencies have been stagnant, as illustrated by Figure 1.1. To keep up with the increasing network bandwidth, it is necessary to allocate more and more CPU cycles to network processing. The introduction of 10 G Ethernet highlighted many inefficiencies in existing OS network stacks. This is especially apparent for stateful protocols, e.g., TCP/IP, and small packet pay-
loads, making line-rate processing a challenge. The cost of processing TCP/IP, the most widely deployed stateful protocol, was analyzed by Foong et al. [FHH+03]. Their analysis reinforced the 1 Hz/1 bps rule of thumb which stipulates that to process 1 bps of network traffic, 1 Hz of CPU processing is necessary. More critically, they showed that with increasing CPU frequencies, the disparity between the CPU speed and the latencies of the memory and I/O increases. This means that more CPU cycles are wasted waiting for those peripherals, increasing the 1 Hz/1 bps ratio further. Moreover, this disparity leads to idle times in the range of microseconds making it infeasible to schedule other work in between, an issue that is also apparent in high-performance networks such as Infiniband or low latency storage devices.

To cope with the increasing network bandwidth and address inefficiencies in network processing, numerous hardware features were introduced to offload functions to the network card. Early on, packet validation functions such as checksum computation were offloaded to the network card. TCP Segmentation Offload [CGY01] (TSO) delegates the segmentation to the NIC, thereby reducing the per-segment overhead on the CPU and PCIe bus. Interrupt coalescing [WWL05] further reduces the per-packet overhead by trading off latency. Receive-Side Scaling (RSS) addresses the scalability in multicore systems and implements multiple receiving hardware queues. With the purpose of assigning a sepa-
rate queue to each core, it reduces the synchronization across cores. Direct Cache Access (DCA) [HIT05, KHM09] enables direct placement of received packets into the cache of the CPU, bypassing the memory, that with increasing network bandwidth, could become a bottleneck. Foong et al. [FHH+03] analyzed a number of these features and showed that all of them provide minor performance improvements, for instance checksum offload results in a 10% improvement. The reason for this is the underlying bottleneck of moving the packets in and out of the processor. Therefore, they concluded that reducing data copies has a higher impact than, for instance, checksum offloading. The hotspots in TCP processing remain to be data copies, interrupt processing, socket and protocol processing, and kernel overheads.

Regarding TCP offload, there are two variants: TCP chimney offload [mic] and full TCP offload engines (TOE). The former offloads only the fast path, i.e. processing of payload packets, to the network card. The control path, i.e. setup and tear-down of connections, remains on the CPU. The latter offloads the complete stack to the network card. Both solutions have generally limited hardware resources to store the connection state and, once exhausted, performance can degrade significantly. Further, TOEs, despite the full offload, can still suffer from substantial kernel overheads if they are not well supported by the OS.

In contrast with the hardware features discussed so far, user space network stacks, e.g., Intel’s DPDK [inta], are another approach to address the overhead of data copies and context switching by bypassing the OS. State-of-the-art implementations [JWJ+14, KSP+19] have shown line-rate processing of up to 40 Gbit/s, but at the cost of allocating multiple CPU cores to the network stack.

Remote Direct Memory Access (RDMA) networks originating from supercomputers completely offload the network stack to the network card and abolish the common socket interface. While RDMA can provide very low latency and high bandwidth, its low-level application interface and the lack of memory management make it challenging to build applications [FA09, BCG+16] that can take full advantage of its capabilities.

### Exponential Data Growth

While CPU frequencies have been stagnant for a number of years, we have seen an exponential growth in data collected and processed. To cope with this massive amount of data, more distributed data processing systems are being proposed. Many of these systems partition data across the nodes to minimize data exchange and reduce synchronization be-
between nodes. However, with an increasing number of nodes and more complex algorithms or queries executed on the data, synchronization and data exchange is inevitable and can easily become the bottleneck of the system. As a result, there is paramount interest in adopting high-performance networks, e.g. RDMA, for distributed systems to alleviate those potential bottlenecks.

For compute-intensive applications, the exponential growth in data has led to the adoption of accelerators and specialized hardware such as GPUs, tensor flow engines (TPUs [Jou16]), FPGAs [CCP+16], etc. in the data center. This has transformed the landscape of the data center from being very homogeneous to having a more diverse environment with nodes or whole racks specialized in certain tasks. Accelerators are significantly reducing the cost of the compute-intensive part of the application. As a result, inefficiencies in the data movement are becoming more evident.

Data movement is also a main contributor to the execution time in applications with low compute intensity such as databases or key value stores where many operations are memory-bound [BMK99, BKM08]. We can see that these applications suffer from the same inefficiencies as the ones observed in network processing, which include memory management, moving data in and out of the processor, and the discrepancy between CPU frequency and memory latency. Considering the benefits of network processing offload, there is an opportunity to achieve similar benefits when offloading data processing to the network. Moreover, with numerous emerging accelerators and the increased heterogeneity of data centers, it is timely to explore specialized hardware that enables in-network data processing.

In-Network Data Processing

The widespread adoption of software-defined networking (SDN) has led to the emergence of programmable network switches [Cav] and network cards (smartNICs) [Mel, Bro]. Since their functionality arises from the scope of SDN, it is limited to installing routing rules, providing more flexibility to the network operator. Microsoft built their own specialized smartNIC [FPM+18] allowing them to offload their SDN stack from the host to the network card. The resulting benefits are reduced load on the host CPU, freeing up cores for other tasks, more predictable latencies, especially tail latencies, and the ability to cope with high network bandwidths, in their case up to 50G. These benefits stem from removing inefficiencies introduced by the operating systems, the memory hierarchy of the CPU, and
1.1. Problem Statement

the interaction between the CPU and the network card. Further, the data-flow architecture
of the smartNIC can efficiently process packets at a very high rate and does not suffer
from the data movement bottleneck observed in CPU implementations.

Inspired by programmable network equipment for SDN, in this dissertation we seek to
explore how data processing can be moved into the network and benefit from the low over-
head of network devices and their data-flow architecture. In fact, data-flow architectures,
which are prevalent in implementations for reconfigurable hardware, have shown great
potential to accelerate data processing operations [MTA09a, WIA14]. Further, placing
them close to or in the network reduces the data movement bottleneck and removes the
overheads of CPU-based systems.

In this work, we consider two approaches to enable in-network data processing: 1) Moving
data processing accelerators into the network by directly attaching them to the network.
This approach requires network processing capabilities on the accelerator itself. We present
a scalable network stack architecture for FPGAs addressing the requirement of data center
applications. Based on this architecture, we implement a TCP/IP stack that enables
network-attached data processing accelerators. 2) Moving data processing to the network
card by enhancing the network protocols with data operations. In particular, we introduce
Smart Remote Memory (STROM) to accelerate distributed systems by extending RDMA
with data processing and enhanced data access capabilities.

1.1 Problem Statement

In this thesis, we explore how to enable in-network data processing in light of increasing
network bandwidth and the exponential growth of data, demanding scalable, distributed
data processing systems. Even though the ideas presented in this thesis were implemented
and evaluated on FPGAs, we want to emphasize that many of them could be applied to
other types of accelerators or smart network cards.

The primary research questions addressed in this dissertation are:

1. How can a hardware design supporting stateful network processing balance the trade-
offs between line-rate processing and maintaining the state for a high number of
concurrent connections?
Chapter 1. Introduction

2. How can data processing accelerators such as FPGAs be attached to the network without introducing unnecessary overhead that could void the benefit of acceleration?

3. How can network processing be extended with data processing capabilities while maintaining its favorable characteristics, e.g. line-rate processing and low latency? Further, what is the right abstraction to expose this functionality to the application?

4. What are the type of operations that can be offloaded into the network card to benefit distributed systems?

1.2 Contributions

This dissertation makes four key contributions:

- A scalable network stack architecture where the number of concurrent connections and the width of the data path can be parameterized. Together with the clock frequency, the data path width determines the processing bandwidth. This is in contrast to existing network stacks which support only a limited number of concurrent connections, in the range of a few dozen, and are tailored to a fixed data path.

- Enabling network-attached accelerators, e.g., key-value store nodes or distributed storage nodes, through a scalable TCP/IP stack implementation for reconfigurable hardware supporting thousands of connections. The implementation has a high degree of flexibility such that it can be adjusted to different application requirements, e.g., capacity of the receiving buffers, packet processing latency, and number of concurrent connections.

- Wisent, an open source RDMA network card that facilitates research exploring RDMA and how its functionality can be extended or enhanced. To our knowledge, this is the first open and freely available hardware implementation of RDMA. The network card deploys a ROCE v2 stack supporting one-sided read and write operations and up to 16,000 queue pairs. It operates at 10 Gbit/s line-rate and provides latency within the range of commercial network cards while being completely implemented in programmable logic.
- Smart Remote Memory (STROM), a mechanism to extend RDMA with processing kernels that are deployed on the network card. These kernels can accelerate data accessing and processing operations. The former can reduce the number of required network round trips by executing multiple memory accesses in a single operation. The latter can execute operations on data that is either written to or read from the remote host memory, adding only a negligible latency overhead. STROM introduces a generic interface enabling the development of a wide range of different acceleration kernels.

1.3 Thesis Outline

Chapter 2 gives a brief introduction of Field Programmable Gate Arrays (FPGAs).

Chapter 3: Scalable Network Stack Architecture

In this chapter, we present a scalable network stack architecture that supports thousands of concurrent connections and line-rate packet processing. The architecture provides flexibility in two dimensions: 1) the maximum number of concurrent connections and 2) the width of the data path to adjust the target bandwidth. These two parameters are defined ahead of compile-time and affect the resource usage of the network stack. By exposing them to the user, the optimal configuration can be chosen to satisfy the trade-off between application requirements and FPGA resource usage.

Chapter 4: Scalable TCP/IP Stack

This chapter describes the implementation of a TCP/IP stack based on the scalable network stack architecture introduced previously. The presented implementation operates at 10 Gbit/s line-rate and enables network-attached accelerators. Different configurations regarding the receiving and transmitting buffers are discussed, as well as how they can be tailored to the application to reduce latency and the DDR footprint, as well as DDR bandwidth.
Chapter 5: FPGA-based RDMA Network Card

In this chapter, we introduce Wisent, a FPGA-based RDMA network card. The goal of Wisent is to provide a research platform to evaluate optimizations and extensions to RDMA. The card implements RoCE v2 with support for one-sided read and write operations. We further discuss memory management to make host memory accessible to the DMA engine on the network card and how the application running on the host can interact with the network card.

Chapter 6: Smart Remote Memory

In chapter 6, we present Smart Remote Memory (STROM), a mechanism to accelerate distributed applications by offloading processing kernels to the network card. These kernels, called STROM kernels, enable multi-step data access operations and in-network processing of RDMA streams. We discuss how we altered the Infiniband protocol to extend it with support for STROM kernels and present three use cases inspired by distributed applications.

Finally, Chapter 7 summarizes the dissertation and discusses potential future work.

1.4 Related Publications

Part of the work in this thesis has already been published and is listed here for reference:


Use cases based on the work in this thesis have also been published:


Field Programmable Gate Array

Field programmable gate arrays (FPGAs) are hardware chips that can be reprogrammed arbitrarily many times and, once programmed, behave similarly to application-specific integrated circuits (ASICs) [TW13].

2.1 Architecture

In contrast to an ASIC where the logic is implemented with logic gates, in the FPGA logic gates are represented through lookup tables (LUTs) that encode the function of the logic gate. An n-input LUT can represent any Boolean function with up to n Boolean arguments. Modern FPGAs from Xilinx and Intel deploy 6-input LUTs. Since LUTs are implemented using SRAM, the encoded logic function can be reprogrammed by rewriting the content of the SRAM. Multiple LUTs together with carry logic, flip-flops and multiplexers are grouped into a logic cell, called slices by Xilinx and ALMs by Intel. The resources within a slice can be configured to implement logic functions, arithmetic functions, or memory. Slices are further grouped into configurable logic blocks (CLBs), also called logic array blocks (LABs) by Intel.

Apart from CLBs, the FPGA consists of dedicated on-chip memory, called block RAM (BRAM), digital signal processing units (DSPs), and a programmable interconnect. CLBs, BRAMs, and DSPs are organized in columns on the fabric, see Figure 2.1. All elements
Chapter 2. Field Programmable Gate Array

Figure 2.1: Internals of a Field Programmable Gate Array (FPGA)

on the FPGA can operate concurrently providing immense parallelism. Note that this parallelism can compensate to some degree for the lower clock frequency of FPGAs in comparison to CPUs.

2.2 FPGA Design Flow

The FPGA design flow, which maps a program written in a hardware description language to an FPGA configuration, is a complex process since many constraints have to be taken into account. To explain the process, we focus on the Xilinx design flow, as illustrated in Figure 2.2.

Synthesis translates code written in a hardware description language into a gate-level netlist. This netlist describes the complete circuit with logical elements, e.g., gates, flip-flops, etc.

Implementation consists of three steps: Translate, Map, and Place & Route. Translate combines the netlist from the Synthesis with constraints specifying physical elements, such as I/O pins or buttons, and produces a logic design file. Map then divides this logic design
Figure 2.2: FPGA Design Flow

2.2. FPGA Design Flow

into smaller sub blocks that can be mapped to device resources such as CLBs, BRAMs, DSPs, or I/O blocks. In the next step, **Place & Route**, places these sub blocks into specific locations on the FPGA chip and interconnects them. **Place & Route** takes into account all constraints including timing constraints defining the maximum path delays to meet the user-defined clock frequency. To find a solution that meets all constraints, it uses the simulated annealing algorithm, a heuristic, that can require multiple iterations, making this the most time consuming step in the whole design flow.

Finally, a **Bitstream** file is generated, which describes the complete configuration of the FPGA in a bit-serial format. This **Bitstream** can be loaded onto the device to program it.
2.3 Programming

Applications implemented on the FPGA can benefit from two types of parallelism: First, there is data parallelism similar to SIMD (Single instruction multiple data) on the CPU, but the functionality that can be executed in a single clock cycle on the FPGA can be more complex and customized to the application. Second, pipeline parallelism allows executing different steps in an application concurrently and deploying them as a pipeline. Thanks to the on-chip interconnect and memory that can be used as FIFOs, data can be passed between pipeline stages with minimal overhead. The two types of parallelism can also be combined to take full advantage of the available resources.

Given the architecture of the FPGA, there are a number of constraints, some of which are determined by the clock frequency, by the amount of available resources, or by the location of resources. As we have seen, the FPGA design flow tries to fulfill these constraints. However, it is also essential that the hardware engineer comes up with a design that is feasible, e.g. does not exceed the available resources, and maps well to the underlying architecture, e.g. is aware of dedicated resources (BRAM and DSP), and is sufficiently pipelined to limit the path delay. Because of this, it is a challenging task to implement a design that achieves a high resource utilization through pipeline or data parallelism while operating at a high clock frequency. Therefore, most applications target a frequency in the range of 100-400 MHz. The newest FPGA devices can be clocked up to 1 GHz [Hut], but this feature has yet to be commonly used.

2.4 System Integration

Originally, FPGAs were commonly deployed as glue logic to interface off-the-shelf ASICs. With the increasing amount of logic resources available, they became a viable option to accelerate data processing operations. Given the limited amount of on-chip memory – in the range of a few megabytes – FPGAs are often used as a bump-in-the wire accelerator for stream processing [MTA09b, NSJ15], which only requires a limited amount of state. However, modern FPGA boards usually have DDR memory in the range of a few gigabytes attached to the FPGA, and the first devices equipped with high bandwidth memory (HBM) will be available shortly.

In recent years, multiple platforms for deploying FPGAs in the data center have emerged.
2.4. System Integration

A key aspect of each platform is the placement of the accelerator within the system. In Microsoft’s Catapult [CCP+16], the FPGA is placed on the datapath between the network card and the top-of-rack switch, as shown in Figure 2.3a. Additionally, the FPGA also connects over PCIe to the CPU. This setup enables \textit{bump-in-the-wire} processing of network packets and compute offload over PCIe.

The Intel Xeon+FPGA platform [OSC+] (Figure 2.3b) and IBM’s CAPI for Power8 [SBJS15] are two examples of hybrid architectures where the FPGA is embedded as a normal processor in the system. The Intel Xeon+FPGA system provides cache-coherent memory access over QPI, while CAPI does so over PCIe. The shared memory architecture of these systems enables tight coupling between the user logic implemented on the FPGA and software running on the CPU, without having to explicitly move data to and from the accelerator.

A more traditional approach is taken by the public cloud offerings from Amazon, Alibaba, Huawei, and Baidu where the FPGA is PCIe-attached (Figure 2.3c). This setup requires either explicit data movement or a copy of the data in the FPGA-attached DRAM.
2.5 High-Level Synthesis (HLS)

FPGAs are traditionally programmed using hardware description languages such as Verilog or VHDL. Recently, high-level languages and synthesis tools have emerged which enable translation of C/C++ or OpenCL code to logic gates [Alt, Xila, CCA+13].

OpenCL programming environments are provided by Intel and Xilinx. A key advantage of OpenCL is that it not only allows programming of the accelerator, but also handles the interaction between the host and the accelerator. Specifically, it simplifies the data movement and issuing of tasks. OpenCL can provide task and data parallelism which is suitable for FPGAs, but also other accelerators such as GPUs or MICs (Many Integrated Core Architectures). However, it cannot fully leverage the pipeline parallelism of FPGAs. To address this, both FPGA vendors introduced vendor-specific pragmas that enable direct data movement between OpenCL kernels. These pragmas enable pipeline parallelism between kernels, thereby avoiding memory accesses.

The two main compilers that synthesize C/C++ to a hardware description language are LegUP HLS [CCA+13] and Vivado HLS [Xila]. LegUP HLS originates from academia and can target FPGAs from numerous vendors. Vivado HLS was developed by Xilinx and can only target their own FPGAs. In this work, we make extensive use of Vivado HLS to speed up the development time and improve design flexibility.

In contrast to hardware description languages, Vivado HLS provides significantly higher design abstractions such as data structures, built-in concepts for data streams with hidden flow control, simplified stitching of modules, and automated BRAM and FIFO instantiations. With that, the code becomes much more expressive, focused on the actual functionality, and less cluttered with hardware design details.

Given that the HLS and OpenCL compiler of Xilinx share the same back-end, there are two different programming styles that can be adopted in Vivado HLS. One of them focuses on data parallelism as in OpenCL by extensively using for loops to iterate over input and output memories. The other one, which we use in this work, focuses on pipeline parallelism by enabling data-flow architectures. Figure 2.4 illustrates how C++ functions are mapped to hardware modules that operate concurrently and can be pipelined. The example shows a simple map-reduce application where the mapper function partitions an input data stream into two streams. Each of these streams is buffered in a FIFO and then consumed by one of the reducer functions. The resulting hardware implementation (Figure 2.4b) pipelines
the mapper and reducer modules.

An added bonus of HLS is that it can target different device families and apply code transformations to achieve a specific target clock frequency.

(a) C++ implementation of a data-flow consisting of a mapper and two reducer functions.

(b) The mapper and reducer modules are pipelined and exchange data over two FIFOs.

Figure 2.4: A simple data-flow architecture in Vivado HLS.
Facilitating network-attached accelerators requires the implementation of a network stack on the accelerator itself. We consider the acceleration of data center applications, which consist of multiple services, serve numerous clients concurrently, and have to adhere to strict service level agreements (SLA). To address these requirements, it is vital that the network stack on the accelerator has the following properties: 1) high throughput to cope with the ever-increasing network bandwidth, 2) low latency to ensure low overall latency, 3) high connection count to serve many clients concurrently, and 4) interoperability with existing infrastructure and applications. Not fulfilling any of these properties could diminish the usefulness of the accelerator. For instance, if only a limited number of clients can be served, the accelerator might not be fully utilized. Or if the latency is prohibitively high, it might increase the overall latency such that SLAs cannot be met.

Targeting FPGAs as data processing accelerators, we examine existing network stacks for reconfigurable hardware. Unlike software network stacks which target any type of application by exposing their functionality through a generic API, implementations for reconfigurable hardware come in a much larger variety in terms of features and usability. In fact, most implementations target a specific domain or application and limit the functionality to the requirements of that application. Further, the implementation is often tailored
Figure 3.1: Generic architecture that supports only a single or limited number of connections.

towards a specific device and network bandwidth. One reason for this tailoring, especially for embedded applications, is to minimize resource usage. However, FPGAs have evolved from small chips being used as glue logic to multi-die chips consisting of billions of transistors. This abundance of transistors allows us to implement a more generic network stack that can support a wider range of applications, similar to software implementations. At the same time, we want to avoid inefficient use of the available resources. Therefore, we want to provide a flexible solution that can be parameterized at compile time such that resource usage is determined by the requirements specified by the application developer.

Our scalable network stack architecture provides flexibility in numerous dimensions such that a single implementation can be adapted to different application requirements and constraints. In particular, our architecture allows adapting the width of the data path to adhere to different bandwidth requirements, enables portability between devices through high-level synthesis [Xila], and adjusts its memory footprint according to the application requirements. As a result, our architecture, similar to OS network stacks, can support a wide range of applications and requirements.
3.1 Architecture

Existing architectures mainly target two domains: embedded applications [LLSH05, Uch08] and high-frequency trading (HFT) [DKYW16, din]. For the former, the focus is on low resource usage under the assumption of a low packet rate and a limited number of connections. The latter has to provide ultra-low latency communication for a limited number of connections. Despite the different target metrics (resources vs. latency), both domains have in common that they only support a low number of connections. Therefore, their designs have some similarities, and can be generalized as shown in Figure 3.1.

On the receiving data path, the packet is first validated by checking the CRC of the Ethernet frame and protocol checksum(s) if available, e.g., IP, UDP, TCP. If a packet is determined to be invalid, it is dropped, otherwise it is stored in the RX Buffer, generally implemented using BRAM. For most embedded implementations, this buffer can only hold a single packet at a time, which is sufficient assuming a low packet rate. On the other hand, for line-rate implementations targeting high-frequency trading, the buffer has to hold multiple packets or even needs to represent a receiving window, as for instance in TCP. This requires an RX Buffer with multiple kilobytes of capacity. Finally, a parsing module processes the packet in the RX buffer and forwards the payload to the application.
In case of a stateful protocol, it also updates and maintains the state stored in registers. On the transmitting path, a packet generating module interacts with the application and constructs the full packet in the BRAM-based TX Buffer. Similar to the RX Buffer, the capacity varies depending on the application. For reliable communication, packets have to remain in the TX Buffer until acknowledged. Therefore, its capacity has to be 10-100s of kilobytes for line-rate processing, while in the embedded case buffering of a few frames can be sufficient. Once the packet is fully composed in the TX Buffer, it is passed through a transmitting module that calculates and inserts any required checksums or the Ethernet CRC. For stateful protocol processing, the state is kept in registers both to have low latency access and to not incur additional BRAM usage. Generally, BRAMs are solely used as packet buffers.

Our scalable architecture is shown in Figure 3.2. It implements a data-flow architecture on the receiving and transmitting data path. Thanks to heavy pipelining, packets are processed in a cut-through manner, with the exception of the checksum calculation, which requires a store-and-forward. At a high level, each stage processes one protocol layer of the packet and each protocol stage is pipelined internally to achieve line-rate processing. Depending on the protocol and the application, the RX Buffer is either implemented using BRAM or DRAM to buffer the payload of the incoming packets. For stateful protocols, finite state machines (FSMs) in the corresponding protocol stage check and update the state. In contrast to existing designs, the state is stored in BRAM, providing capacity for thousands of connections. Finally, the application is able to retrieve the payload by accessing the RX Buffer. On the transmitting path, the payload received from the application is stored in the TX Buffer. In case of reliable communication, all transmitted packets have to be kept until acknowledged such that they can be retransmitted if necessary. Therefore, we map the TX Buffer to DRAM. Starting from the highest layer, the packet header is generated and prepended to the payload if available. Then, the packet is forwarded to the next stage where the next header is generated and prepended.

Comparing existing designs to our architecture, there are two main differences: 1) connection state is moved from registers into BRAM and 2) packet or payload buffers are moved from BRAM to DRAM. Both changes are motivated by the goal to support thousands of concurrent connections. Given that the state for a single connection requires tens of bytes, the total capacity has to be in the range of 10-100s of kilobytes. While this is clearly beyond the capacity of registers, it comfortably fits into on-chip BRAMs.

For the payload buffers, considering reliable communication at 10 Gbit/s line-rate and a
3.2. Scalability

latency of 1 ms until packets are acknowledged, up to 1.25 MB of data can be in-flight. However, since the latency can vary between different connections and fluctuate depending on network conditions, the application, and the remote node, we store the transmitted but unacknowledged payload in DDR memory. The RX Buffer can be mapped to either BRAM or DRAM depending on the application.

It is clear that the higher capacity memories incur higher access latencies. However, our pipelined data-flow architecture is able to hide the increased access latencies and can support line-rate processing even for small packets, as shown in detail in Section 4.3. However, our approach has a higher end-to-end latency – e.g. from the point a packet is received by the device until the payload is passed to the application – in comparison to ultra-low latency designs. Nevertheless, the data path latency of our architecture is below 10 µs, sufficiently low for data center applications, our main target for this work. Moreover, apart from HFT, many applications have similar or even more relaxed latency requirements.

3.2 Scalability

Our architecture is scalable in two dimensions, the number of connections and the width of the data path.

3.2.1 Scaling Connections

As described, our architecture stores the connection state in BRAMs. Access from different modules is provided through arbitration. The additional latency introduced by the arbitration together with the access latency is hidden by pipelining. The benefit of using BRAMs is not only that more connections can be stored, but the number of connections can easily be adapted by varying the size of the BRAMs. In fact, the BRAM usage scales linearly with the number of connections, see Section 4.5.4. In some existing implementations, one instance of the network stack can support only a single connection, and support for multiple connections is achieved through multiple instances. For instance, the TCP/IP stack by Ding et al. [DKYW16] requires one instance per connection, which means it would exhaust the device resources at 42 connections. On top of that, with an increasing number of processing pipelines, the utilization per pipeline is reduced, leading to an inefficient use of resources.
Apart from scalability, moving the state to BRAMs also results in an architecture where the packet processing pipelines are clearly separated from the state-keeping modules. This separation enables independent operation of the receiving and transmitting pipeline while providing concurrent access to the state from both paths.

### 3.2.2 Scaling Data Path

As mentioned, protocol processing is heavily pipelined with a pipeline stage for each protocol consisting of multiple sub-stages. In the first sub-stage, the protocol header is parsed and removed. The remainder of the packet is then re-aligned in the second sub-stage. In case the protocol is stateful, the remaining sub-stages consist of finite state machines (FSM) and other control modules, see Figure 3.3.

In detail, the *Parse Handler* module extracts relevant fields from the protocol header and forwards them on a separate metadata bus to the next stage or module. In this case, the metadata is passed to the FSM, which will check the state of the incoming packet against the state stored in the *State Table* and, if necessary, update it. Based on this, the FSM makes a decision whether the packet can be forwarded or has to be discarded and communicates this to the *Packet Dropper* module. The transmitting path is implemented analogously. First, a FSM combines metadata received from the application with metadata
stored in the state-keeping modules. Then, based on this metadata, a *Generating Header* module generates the protocol header.

The design of our packet processing pipeline clearly separates the data path from the control path, where the extracted metadata is used to check and maintain the state. This separation of the control and data plane allows scaling the width of the data path without affecting the control plane. Different network bandwidths can be targeted by adapting the width of the data path together with varying the clock frequency of the circuit. In our architecture, the width can be changed through a simple compile-time parameter.

### 3.3 Data Structures

We identify a number of data structures as common building blocks when implementing a network stack in hardware. Their design principle is common across different network protocols. We present their functionality and how they can be implemented efficiently on FPGAs given the limited on-chip memory capacity. In addition, all of them use on-chip memory to store the state, making them scalable with the number of supported connections.

#### 3.3.1 Hash Table

To easily map network packets to a specific connection or flow, commonly the 5-tuple, consisting of source IP address and port number, destination IP address and port number, and the protocol, is used as a unique identifier. Since the 5-tuple has a size of 13 B, it is more efficient to map it to a unique connectionID that is part of a dense space. This mapping can be achieved through a hash table where the 5-tuple acts as the key and the unique connectionID as the value. We use this mechanism in the TCP/IP stack – see Chapter 4 – where we map the 4-tuple (omitting the protocol) to a unique session ID. This mapping occurs when a packet enters the system such that all further processing can use the dense session ID as a unique identifier.

In this work, we use an existing implementation of a BRAM-based hash table, in particular a variation of the *Exact Match Binary CAM Search IP for SDNet*, an IP core from Xilinx [Xilb]. In comparison to a traditional TCAM, it uses less resources and scales linearly
in its resource requirements with the number of entries. Up to 85-90% of the allocated memory capacity can be used to store key-value pairs.

### 3.3.2 State Table

Since our network stack architecture does not have a centralized design with a single module updating state for incoming and outgoing packets, state needs to be accessible and modifiable by multiple modules concurrently and likely from both data paths. We store the state in BRAM-based tables that allow access from different modules through arbitration. The following operations can be executed on the table entries: read-only, write-only, and read-modify-write (rmw). If more than one module is able to execute an update operation on the same entry and field, a mechanism for mutual exclusion is required to guarantee consistency.

**Figure 3.4:** The State Table data structure with locking mechanism to guarantee consistency.

**Design and Implementation**

A *State Table* stores the state in a dual port BRAM allowing concurrent read and write operations. For fast lookups, the connectionID is used as the index in the table. Each module accessing the *State Table* has its individual access port dedicated to one of the three
3.3. Data Structures

operations. The optional locking mechanism is implemented directly in the arbitration logic that arbitrates between the different modules accessing the State Table. Instead of having a lock for each entry in the table, which can result in thousands of locks, we implement one lock for each read-modify-write (rmw) port. Figure 3.4 illustrates the case where three ports can update the state, two of them through a rmw operation. For instance, if Module 0 wants to execute a rmw, it first issues a read operation. On this read, the Arbiter checks if the to-be-accessed entry is locked by another module, in this case Module 1. If the entry is not locked, it will lock it for Module 0 and return the entry. When the corresponding write operation is executed by Module 0, the entry is released. This mechanism allows concurrent access of different entries by different modules. Only in case of a conflict, i.e., an update to the same entry, the locking mechanism stalls the access by one of the modules to achieve mutual exclusion. Further, our locking mechanism introduces minimal space overhead; each lock requires one bit to indicate if it is active and an index to identify the locked entry.

While we have seen how consistency can be guaranteed through locks, in many cases it is possible to avoid locking by either splitting the fields between two or more tables such that only a single module requires rmw access or by limiting the rmw access to certain fields and thereby avoiding overlap between different modules. The former approach was used in our TCP/IP implementation (Chapter 4) where we split the fields of the receiving and transmitting window into the RX SAR and TX SAR Table such that no locks are required. If this is the case, the implementation can be simplified since it only has to arbitrate between the different ports without providing mutual exclusion.

Since entries can be directly accessed through the connectionID, the State Table has no resource overhead apart from the arbitration logic and the locks, which is negligible.

3.3.3 Multi-Queue

A number of protocols, for instance Infiniband, achieve reliability by keeping track of every packet in-flight. However, the number of active connections and packets in-flight per connection is not known a priori and can vary over time. Therefore, a flexible data structure is required to efficiently track all in-flight packets. In software, a data structure can dynamically grow and shrink, since memory can simply be allocated at run-time.

To accomplish the same dynamic properties in hardware, we implement the Multi-Queue data structure which leverages the fact that despite this dynamic behavior at run-time,
there is an upper limit on the number of packets in-flight assuming no network failures occur and given a certain network bandwidth and average round trip time to the other nodes. The Multi-Queue data structure logically represents a fixed number of queues where each queue can have a variable capacity at run-time, but the combined capacity of all queues is fixed. With this data structure, we address the fact that on-chip memory capacity on the FPGA is limited and memory management is ideally avoided to maintain low-latency access.

Each queue in the Multi-Queue can be manipulated using the following operations:

- $\text{push}(q, T)$ to append an element $T$ to queue $q$
- $\text{front}(q)$ to access the first element in queue $q$
- $\text{pop}(q)$ to remove the first element in queue $q$

Apart from keeping track of in-flight packets, this functionality is also required in RDMA to keep track of all outstanding read operations and their corresponding local memory address. This is the address where the retrieved data has to be stored. Whenever a new read request is transmitted, the corresponding local memory address is appended to the queue of that connection. When a reply to a read request is received, the first element of the corresponding queue is retrieved to obtain the local memory address such that the payload of that packet can be written to this memory address.

Figure 3.5: The Multi-Queue data structure maps multiple queues, each with a variable capacity, to fixed-sized on-chip memory.
3.3. Data Structures

Design and Implementation

To implement the Multi-Queue, we represent each queue as a linked list of elements containing the relevant data, e.g. a unique packet identifier. The elements of all queues are stored in a BRAM of fixed capacity, the Element Table in Figure 3.5. A second BRAM, the List Table, stores the metadata for each linked list. An entry in this table indicates if the queue is empty and points to the head and tail elements of the list, which are stored in the Element Table. All elements in the Element Table which are not assigned to a list are also stored in the free list, which is implemented as a FIFO.

The push\((q, T)\) operation consists of three steps: 1) retrieving the entry at index \(q\) in the List Table and dequeuing a free element from the free list, 2) updating the next pointer in the current tail element, and 3) updating the tail in the List Table to point to the dequeued element and assign the data \(T\) to this element.

To execute the front\((q)\) operation, two lookups are required: the first to retrieve the head pointer from the List Table and the second to retrieve the data value from Element Table. The pop\((q)\) operation executes the same lookups, but also updates the head value in the List Table and invalidates the element in the Element Table and adds it to the free list.

The flexibility of the Multi-Queue is achieved through indirections that are encoded in metadata. Storing this metadata incurs space overhead, which can be quantified with the following formula:

\[
N \times (\log_2(M) + \log_2(M) + 1) + M \times (\log_2(M) + 2) + M \times \log_2(M)
\]

where \(N\) corresponds to the number of queues and \(M\) to the total number of list elements. Table 3.1 shows the overhead for some example configurations of the Multi-Queue.

3.3.4 Timers

Network protocol implementations require timers for different purposes, such as detecting packet loss, connection break-down, or periodic heart-beats. In most cases, it is sufficient to have a single timer per event and per connection. Our timer design can cover events that have to be triggered after a given period of inactivity, which comprises most events occurring in network processing. The main operations required are setting a timer, which includes starting as well as restarting it, and clearing a timer, which cancels it.
Table 3.1: Space overhead of the Multi-Queue data structure.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$M$</th>
<th>data</th>
<th>total</th>
<th>overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,024</td>
<td>16,384</td>
<td>64 bits</td>
<td>1,570 Kbits</td>
<td>521 Kbits</td>
</tr>
<tr>
<td>4,096</td>
<td>16,384</td>
<td>64 bits</td>
<td>1,659 Kbits</td>
<td>610 Kbits</td>
</tr>
<tr>
<td>16,384</td>
<td>16,384</td>
<td>64 bits</td>
<td>2,015 Kbits</td>
<td>967 Kbits</td>
</tr>
<tr>
<td>1,024</td>
<td>16,384</td>
<td>256 bits</td>
<td>4,716 Kbits</td>
<td>521 Kbits</td>
</tr>
<tr>
<td>4,096</td>
<td>16,384</td>
<td>256 bits</td>
<td>4,805 Kbits</td>
<td>610 Kbits</td>
</tr>
<tr>
<td>16,384</td>
<td>16,384</td>
<td>256 bits</td>
<td>5,161 Kbits</td>
<td>967 Kbits</td>
</tr>
</tbody>
</table>

Figure 3.6: Timer, based on true dual port BRAMs allowing concurrent iteration over timer entries and updating of timers.

**Design and Implementation**

Our timer implementation is based on an array with one entry per connection. The entries can be directly addressed through the unique connectionID. Each entry contains a time period and a flag indicating if the timer is active. This array is mapped to a true dual port BRAM such that the entries can be modified by two different pointers concurrently, see Figure 3.6. The update pointer is used to either set a timer by setting the time period and the active flag or clearing a timer by disabling the active flag. The iterator pointer sequentially iterates over all entries in the array. If an entry is active and the time period is larger than 0, the iterator decrements the time period by 1. If the timer is active and the time period is equal to 0, the corresponding event is triggered and the timer is canceled. Depending on the timer, the entry can be extended with additional fields such as a retry counter to keep track of how often the timer was triggered or an event type in case the timer can fire different types of events.
When setting the value for the time period, the size of the array and the clock period has to be considered. For an array of size $N$, the iterator decrements every entry only every $N$ clock cycles. Therefore, when a timer is set, the absolute time period has to be converted according to the following formula:

$$time\ period = \frac{absolut\ time\ period}{clock\ period \times N}$$

Neely et al. [NBS10] presented a similar design for their alarm clock style timer. Using two processes, one to set the timer and one to iterate over all timers. However, in their implementation, the iterator takes five cycles per entry while in our case it only takes one cycle, resulting in a finer time granularity.

### 3.4 Evaluation

The scalable architecture was evaluated by implementing two different protocol stacks, a TCP/IP stack, see Chapter 4, and a RDMA over Converged Ethernet (RoCE) stack, see Chapter 5. In this section, we discuss how the resource usage scales when varying the number of concurrent connections or the width of the data path. The performance evaluation can be found in the corresponding chapters.

#### 3.4.1 Scaling Connections

Both implementations, TCP/IP and RoCE, were deployed on a Xilinx XCVU9P FPGA. Figure 3.7 reports the resource usage in LUTs representing logical resources and BRAMs representing on-chip memory. The number of connections or queue pairs (QP) is varied from 512 to 16,384. The logic resource usage stays almost constant for all configurations, while the increase in on-chip memory is linear with the number of connections. This reflects the separation of packet processing from the state-keeping data structures, one of the key characteristics of the scalable architecture.

#### 3.4.2 Scaling Data Path

We varied the data path width of the RoCE stack from 8 B to 64 B to illustrate how the resource usage scales, see Figure 3.8. The logic resources show a sublinear increase, since
Chapter 3. Scalable Network Stack Architecture

Figure 3.7: Scaling of resource usage when changing the number of connections respectively queue pairs (QP). The circuit is clocked at 156.25 MHz with a data path width of 8 B.

they are allocated to the control plane, which is unaffected and decoupled from the data path that is scaled. BRAMs are either used to keep state or to implement FIFOs on the data path. Only the latter is affected when increasing the data path width, leading to a higher BRAM usage.

When the data path width is increased from 8 B to 16 B, the BRAM utilization decreases by 3. This is caused by the synthesis that determines whether the FIFOs on the data path are mapped to BRAMs or not. An over proportional increase of BRAM usage can be observed when doubling the data path width, from 32 B to 64 B. This increase stems from the HLS compiler, which maps every FIFO with a width of 64 B and a depth larger than one to BRAMs, while this threshold is set at eight for narrower FIFOs.

3.5 Related Work

Existing network architectures target different application domains. Each domain is optimizing for different performance metrics and constraints. For instance, high-frequency trading (HFT) focuses on ultra-low latency, while on the other hand deployments in embedded environments are optimized for low resource and power usage.

In the area of embedded computing, existing work takes different trade-offs to adhere to these constraints including but not limited to deploying stateless instead of stateful communication protocols, limiting the number of concurrent connections, limiting the packet size to reduce the memory requirements, or limiting the processing bandwidth.
For instance, Lofgren et al. [LLSH05] present a UDP/IP stack that targets embedded deployments and can be deployed in three variations with different trade-offs regarding parallelism, latency, and resource usage. Their minimal variant omits support for ARP and ICMP, assumes a fixed IP address, and operates at half-duplex rate. The medium variant operates at 100 MBit/s in full-duplex mode. This variant also supports ARP for four nodes and replies to ping messages. And finally, their advanced variant provides the same functionality as the medium variant but increases the maximum message size from 256 B to 1518 B and the bandwidth from 100 Mbit/s to 1 Gbit/s. Another approach in the embedded space by Chung et al. [CLL+07] only offloads layer 1-3, which are stateless to the reconfigurable logic. Layer 4 processing that includes TCP, which is stateful, is handled by an embedded PowerPC CPU. Uchida et al. [Uch08] present a full TCP/IP stack including ARP, ICMP, UDP, and TCP, targeting small FPGA devices. The design can reach up to 1 Gbit/s bandwidth, but is limited to a single connection.

Work by Ding et al. [DKYW16] focuses on a low-latency TCP/IP implementation. In their design, for each connection a separate TOE module has to be instantiated. Each instance implements a full processing pipeline including two 64 KB buffers, one for each direction. This approach has clear limitations in terms of scaling out the number of connections. In fact, the BRAM usage of 10 TOE modules supporting 10 concurrent connections is equivalent to the BRAM usage of the TCP/IP stack presented in Chapter 4 that can support up to 10,000 connections. A similar design approach is used by Dini Group [din] where each TOE instance only supports a single connection and multiple instances are required to support multiple connections.
3.6 Summary

We have shown that our network stack architecture is scalable in two dimensions: 1) the number of connections, due to a clear separation of the state-keeping data structures and the packet processing pipelines, and 2) the width of the data path due to the separation of the control and data plane in the packet processing pipelines. In contrast to existing architectures which had a strong focus on low latency or low resource usage, we focus on a high number of concurrent connections and reusability. To achieve this, we trade-off latency by moving from fast memory with limited capacity, to slower memory with higher capacity, allowing us to store more state and packet payload. Our architecture maintains line-rate processing by hiding the additional access latency of the slower memory.

The architecture was used to implement a TCP/IP stack (Chapter 4) as well as a RDMA over Converged Ethernet (RoCE) stack (Chapter 5). Both of these stacks have the properties described in this chapter. The TCP/IP stack operates at line-rate and can support between 500 and 10,000 connections on a Virtex 7 device. The RoCE stack supports between 512 and 16,384 queue pairs and can operate at data path widths of 8B, 16B, 32B and 64B, resulting in 10-80G line-rate processing when clocked at 156.25 MHz. In addition to the data path width, the clock frequency can be increased to support a rate of 100G. The on-chip memory footprint in terms of BRAMs scales linearly with the number of connections/queue pairs. Portability of these two implementations between different devices and device generations is achieved through the use of high-level synthesis [Xila].

With additional pipelining and other optimizations, it might be possible to operate a design based on the scalable architecture beyond 100G, however, we recognize two fundamental limitations that will limit the bandwidth eventually: clock frequency and data width. One option to increase the clock frequency at which the design is operating is through improvements at the silicon level, e.g., higher density, that lead to shorter propagation delays. We observed this, for instance, when porting a design from a Virtex 7 FPGA (28 nm) to a UltraScale+ chip (16 nm) where the same design can be clocked at a higher frequency without any modifications. Another option is to pipeline the design further, but this can be challenging or infeasible for some functions, e.g., state updates. Additionally, more pipelining can increase the pressure on the routing network and negatively affect the achievable clock frequency. The second limitation is the scalability of the data path, which is limited by the bit-wise architecture of the FPGA. Apart from the data bits, the data bus also contains control signals. For instance, a single bit valid signal indicates if the
current data word is valid. While the 512 data bits of a 64 B data bus can be consumed and processed separately, the *valid* signal has to be propagated to all 512 bits to make sure they are only processed when the data is actually valid. The wider the data bus, the larger the fanout of this control signal, resulting in a longer path delay, which can affect the achievable clock frequency.

One way to address these limitations and support line-rate processing beyond 100 G is to deploy multiple network stacks in parallel. However, this approach requires that either the state is partitioned across the different stacks or shared among the stacks. In the former case, load balancing can be problematic since the network traffic is unlikely to be uniformly distributed across connections. In the latter case, it would be challenging to maintain consistency without impairing performance.
Scalable TCP/IP Stack

Accelerators are most often PCIe-attached and co-located with a CPU that acts as their host. Thereby, the accelerator can be managed through its host. However, this setup also introduces a number of restrictions and limitations; for instance, if the accelerator is processing tasks sent by clients over the network to its host. The host is involved in receiving these tasks, forwarding them to the accelerator, checking for their completion, and finally transmitting the results back to the client. This introduces latency and overhead stemming from the interaction between the host CPU and the accelerator, network processing on the host, and data movement to/from the accelerator. Further, the PCIe link to the accelerator has some unfavorable characteristics including high latency and low bandwidth for small data transfers. The latter can be addressed by batching of tasks, which in turn further increases the overall latency.

Network-attached accelerators not only avoid these overheads, but have the potential to receive and process tasks at a low latency. However, this assumes that a capable network stack is implemented on the accelerator itself. In this chapter, we introduce a TCP/IP stack for reconfigurable accelerators. The interoperability of TCP/IP guarantees a seamless integration of the accelerator into existing infrastructure. In fact, TCP/IP is the cornerstone of modern network communications with its support for reliable data transfer including flow control, congestion avoidance, duplicate data suppression and in-order delivery. However, this is associated with substantial complexity. Foong [FHH+03] et al. stipulate that 1 Hz of CPU processing is required to send or receive 1 bps of TCP/IP, which
equates to eight cores clocked at 1.25 GHz for 10 Gbit/s line-rate processing. The reasons for this are manifold and well understood. First of all, as TCP is connection-oriented, the implemented network stack needs to keep state for every connection, something which naturally becomes more complex with the number of open sessions. Secondly, to ensure reliable data transfer, data has to be buffered until an acknowledgment has been received. Additionally, segmentation and reassembly are needed together with out-of-order processing to packetize incoming and outgoing data streams from the application layer. Finally, TCP is interrupt-intensive in nature as, for example, every time a packet is received or a transmission time-out occurs, an interrupt is triggered.

As our implementation shows, many of these challenges do not apply to hardware implementations. One reason is that the hardware circuit is deterministic, leading to a predictable processing time per packet header and payload byte. This means that, for instance, connection state lookups and updates consume a fixed number of clock cycles for each packet. Moreover, thanks to the hardware parallelism, all modules are running concurrently; as an example the timers tracking time-outs are running independently of the logic that is processing packets. Further, interrupts or events that trigger the generation of a packet, e.g. acknowledgment, are only affecting the transmission path, but are not interfering with the application or receiving path.

Our design aims to enable a wide range of data processing accelerators targeting applications such as big data frameworks, databases, key-value stores [IABV13], and distributed systems. To maximize the stack’s applicability, it is essential to create a flexible solution that allows to efficiently and easily adapt the design to different bandwidth requirements, number of concurrent connections, buffer capacities, and latency constraints, while using a minimal resource footprint. This flexibility is achieved by adopting the scalable network stack architecture introduced in Chapter 3. Additionally, the adopted C++-based design flow using high-level synthesis (HLS) makes it easier to introduce application-specific optimizations.

### 4.1 System Architecture

Implementing a TCP/IP stack that can be seamlessly integrated into existing network infrastructure involves the implementation of a number of protocols such as Address Resolution Protocol (ARP), Internet Control Message Protocol (ICMP), IP, UDP, TCP and
4.1. System Architecture

Dynamic Host Configuration Protocol (DHCP), whereby each protocol handles a different aspect of the communication. This is reflected in Figure 4.1, which illustrates the system architecture of our stack within the dashed lines. At the lowest layer, the stack interfaces with the Ethernet Network Interface, which includes both the Media Access Control (MAC) and the physical layer that handle layer 1 and 2 functionality of the network stack, both of which are standard Xilinx IP cores.

Incoming packets are parsed by the IP Input Handler, which first parses the Ethernet header to extract the EtherType that determines if an ARP or IPv4 packet is contained. In case of IPv4, the next stage in the pipeline validates the IP checksum and parses the IPv4 header to determine if it matches any of the supported IP-based protocols. If this is the case, the packet is forwarded to the corresponding module. Invalid packets are discarded and unsupported protocols are forwarded to a separate interface.

The ARP module handles the address resolution, which is required to map IP addresses to MAC addresses. The module resolves ARP requests from other nodes and also broadcasts a request in case a miss in the ARP table occurs. The ARP table is a lookup table that maps IP addresses to their corresponding MAC addresses. To simplify the implementation,
we assume that the subnet mask corresponds to $255.255.255.0$. This results in an upper bound of 256 nodes for the subnet. As a result, the lookup table can be implemented in an on-chip block RAM (BRAM) with 256 entries and can be directly addressed using the host identifier (the least significant byte of the IP address) to lookup the MAC address. If the subnet includes significantly more nodes, this direct mapping would require too much on-chip memory and could be replaced with a hash table, as introduced in Section 3.3, that allows for a more flexible mapping such that only a subset of the nodes has to be stored in the ARP table.

ICMP enables the exchange of control related messages between two hosts and is processed in the `ICMP` module. Our implementation supports the most popular subset, namely “echo” or “ping” messages, “destination unreachable” messages, which are created upon packet reception for a closed UDP port, and “TTL expired” messages, which are returned when datagrams with an expired time-to-live (TTL) field are observed. Its main functionality, handling of “ping” messages, is useful in practice to determine if the device is alive.

The `UDP` module handles the processing of the corresponding protocol, which is a stateless light-weight protocol that allows for quick and easy data exchange between two end-points.

The `IP Output Handler` merges the packet streams coming from the different protocol modules. For all IP-based packets, it calculates and inserts the checksum into the header. In the next stage, it extracts the destination IP address to issue a lookup request to the ARP table. The ARP module replies with the corresponding MAC address or a flag to indicate the missing entry. In case of a miss, the packet is discarded and the ARP module issues an ARP request for the given IP address. If the lookup is successful, the `IP Output Handler` generates and prepends the Ethernet header to the packet and passes it to the `Ethernet Network Interface` for transmission.

The majority of the design complexity resides with the `TCP` module and forms the key part of the stack. We describe its architecture in detail in Section 4.2. Finally, our stack also includes a minimal DHCP client, which allows for dynamic configuration of the device’s IP address over the network.
4.2 TCP Architecture

The TCP module is the key component of the TCP/IP stack. To achieve our objectives, we adopted the design philosophy of the scalable architecture introduced in Chapter 3 and describe how we mapped TCP processing to it.

![Figure 4.2: Block diagram of the TCP module](image)

4.2.1 Overview

Figure 4.2 shows how the TCP module is implemented as a scalable architecture. The fundamental aspect of the design is that it is divided into two parallel paths, one for incoming packets (RX), and one for outgoing packets (TX). The paths not only contain a protocol engine (RX and TX Engine) that is parsing/generating packets and maintaining the state, but also a packet buffer (RX and TX Buffer) and an application interface (RX and TX App If). The protocol engines handle all protocol processing, the packet buffers store the packets between the engines and the application interfaces which provide and
receive data to and from the user application. The two paths are implemented as data-flow architectures and store and share connection state information through the central data structures located at the heart of the architecture. This separation leads to the desired scalability of the state-keeping data structures, which determine how many concurrent connections can be supported.

4.2.2 Session lookup

The Session lookup module provides two types of mapping: First, a mapping from the four-tuple (source and destination IP addresses and TCP ports) as present in the network packet to a unique connection ID, also called a session ID. This mapping is provided through the hash table design described in Section 3.3. In this implementation, the key size consists of 96 bits for the four-tuple and the value of 14 bits representing the session ID. Second, a reverse mapping from the session ID to the four-tuple is provided. This is implemented in BRAM through direct mapping. The reverse mapping is necessary to generate the packet header when transmitting packets, since the application only has to provide the unique session ID to identify a connection. Except during connection setup, the application is not aware of the destination IP address or port and only deals with the unique session ID. Similarly, the session ID is used internally as an index for all data structures. As the functionality indicates, the first mapping is used by the RX path and the second is accessed by the TX path. The Session lookup module ensures consistency across the two mappings.

4.2.3 TCP State and Port Table

The Port Table and TCP State Table are both implemented as State Tables, as described in Section 3.3. The Port Table keeps track of the state of each port, which can be closed, listening, or active. According to the standard, we support two port ranges, one for static ports (0 to 32,767), which we use for listening ports, and one for dynamically assigned or ephemeral ports (32,768 to 65,535), which are used for active connections. For every incoming packet, the RX Engine queries the Port Table to check if the destination port is either in the listening or active state. If this is not the case, then the packet is immediately discarded and an event for a RST packet is issued. Similarly, the application interfaces access the Port Table to listen on a port or get a new ephemeral port. The TCP State
Table stores the current TCP state of each connection. The state values represent the states as specified by RFC793 [Pos81], i.e. CLOSED, SYN-SENT, SYN-RECEIVED, etc. State values can be updated from the RX Engine but also from the TX App If when a new connection is opened. As described, consistency is guaranteed through read-modify-write (rmw) operations, which ensure mutual exclusion. Since the locking is fine-grained, only the currently accessed entry is locked.

### 4.2.4 Timers

TCP involves three different time-based events; each of these events is triggered by a different timer: The Retransmission Timer keeps track of the retransmission intervals for packets which have been sent but not acknowledged by the remote host. The Probe Timer is set in case data is available but cannot be sent immediately and transmission has to be postponed. This is, for instance the case when the Usable Window is full. Finally, the Time-Wait Timer handles the long time-out in the TIME-WAIT state before the connection reaches the CLOSED state. All timers are implemented using the approach described in Section 3.3 where the number of entries corresponds to the number of supported connections. For retransmission, it is sufficient to have one timer per connection, as explained in RFC6298 [PMJM01], rather than one for every segment. The iterative probing of our timer implementation is a viable approach, as TCP timers operate in the millisecond range. Given a clock period of 6.4 ns and one timer entry access per clock cycle, the timer can update 156,250 connections per millisecond. Or in other words, with 10,000 entries, the granularity of the timer is 64 µs, which is sufficiently fine-grained.

### 4.2.5 Event Engine

Table 4.1 lists all event types that are defined within the TCP module. Events can be generated by the three timers, the RX Engine, and the TX App If. All events are merged into a single event stream in the Event Engine module. Events from the RX Engine are prioritized over all other events to avoid backpressure towards the RX path, which could result in packet loss. The timers are able to handle backpressure by the Event Engine by delaying their events. Since these events are based on time-outs in the range of milliseconds, the impact of the delay is negligible. Backpressure towards the TX App If can temporarily block the application from transmitting data, however, we consider this
Table 4.1: Event types defined in the TCP module.

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX</td>
<td>Regular transmission of payload.</td>
</tr>
<tr>
<td>RT</td>
<td>Retransmission for a specific connection.</td>
</tr>
<tr>
<td>ACK</td>
<td>Acknowledgment, might be delayed by Ack Delay module.</td>
</tr>
<tr>
<td>SYN</td>
<td>Generates a SYN packet to setup a new connection.</td>
</tr>
<tr>
<td>SYN-ACK</td>
<td>Generates a SYN-ACK packet to setup a connection initiated by the other end.</td>
</tr>
<tr>
<td>FIN</td>
<td>Generates a FIN packet to tear down the connection.</td>
</tr>
<tr>
<td>RST</td>
<td>Generates a RST packet, e.g. when an invalid packet was received or a SYN packet was received on a closed port.</td>
</tr>
<tr>
<td>ACK-NODELAY</td>
<td>Acknowledgment that cannot be delayed; this event is used during connection establishment instead of ACK.</td>
</tr>
</tbody>
</table>

A worthwhile trade-off to avoid packet loss on the RX path. The aggregated event stream is passed to the Ack Delay module.

4.2.6 Ack Delay

The Ack Delay module either delays acknowledgments, merges two outstanding acknowledgments, or piggybacks outstanding acknowledgments with transmission events, as described in RFC1122[E.89].

The module is implemented similarly to the Timer module (Section 3.3.4). It also consists of two independent processes and has one timer per connection. The first process dequeues events from the input stream. If the event is of type ACK, it checks if a delay timer is already active. If this is the case, the new ACK event can be merged with the one that is being delayed and forwarded to the TX Engine. Otherwise, the new ACK event is delayed by activating a timer. All other events are directly forwarded and all of them except for SYN and RST disable the timer if it is active, since they piggyback an acknowledgment. The second process iterates over all active timers; in case a timer expires, it outputs an ACK event for the corresponding connection.


4.2.7 TCP buffer and window management

TCP processing requires the buffering of payloads to facilitate retransmission and flow control for both receiving and transmitting. Our implementation allocates two fixed-sized buffers of 64 KB per connection. This means that for a design supporting 10,000 connections, a total of 1.3 GB of external memory is necessary. Since this amount of memory space is not available on-chip, external DDR memory is used. The buffers are implemented as circular buffers, each in a pre-allocated region of the memory. Managing each of the buffers requires a set of pointers to keep track of various, relevant locations in the segment stream. These pointers are stored in the TX SAR and RX SAR Tables, which are instrumental for handling all segmentation and reassembly (SAR) functionality, as well as maintaining the TCP windows.

Figure 4.3 illustrates the two packet buffers. For RX, the buffer space is divided as follows: data that was received (and acknowledged) but not read by the application, available free space to receive new segments, and a non-contiguous number of blocks of out-of-order (OOO) received data (as described in Section 4.3.3). On the transmission side, we need to keep track of three partitions: transmitted but not yet acknowledged data, data written by the application to the buffer but not yet sent, and finally free space. In addition to these three pointers, the TX SAR also stores the Send Window and the Congestion Window. The former is advertised by the other device and indicates the size of its receive buffer.
Figure 4.4: Block diagram of RX Engine with separation of data path, control plane, and state-keeping data structures.

The latter is continuously updated by the Congestion Avoidance algorithm, a mechanism to avoid congestion as described in RFC5681[MPE09]. The Usable Window, as shown in the figure, can be computed on the fly based on the Send and Congestion Window and is not explicitly stored in the table. Both tables hold one entry per connection and are implemented as a State Table, as explained in Section 3.3. Given that the buffer is limited to 64KB, only the received, ack’ed and transmitted pointers which correspond to the sequence and acknowledgment number in the TCP header have to be stored as 32 bit values, while for all other pointers and windows, 16 bit values are sufficient.

By carefully dividing the pointers of the two TCP windows into two separate tables, we are able to avoid any lock and can still maintain the consistency of the entries. This reduces complexity and latency when accessing and updating the SAR Tables.

4.2.8 RX Engine

The RX Engine consists of multiple stages building a data-flow, see Figure 4.4. The first stage parses the IPv4 header to extract the IP addresses and the length of the packet. These fields are used in the second stage to construct the TCP pseudo-header, which is required to verify the TCP checksum. The checksum verification infers a store-and-forward introducing latency depending on the packet size, but maintaining full line-rate
4.2. TCP Architecture

processing. Once the packet is determined to be valid, it reaches the next stage where the TCP header is parsed to extract all relevant fields\(^1\) (SEQ number, ACK number, window size) and flags (SYN, ACK, RST). All this metadata is fed into two finite state machines (FSMs). The first one checks the destination port against the Port Table. If the port is open, it issues a lookup request for the given four-tuple to the Session Lookup module. The second FSM queries the state of the TCP connection of the current segment from the State and SAR Tables. It then checks if a TCP state transition has to be performed and it updates the ACK number, the Send Window, and Congestion Window in the TX SAR Table. Further, it updates pointers in the RX SAR Table, sets or clears timers as needed, and finally issues events to trigger, for instance, the transmission of an acknowledgment packet. In case any of the state machines or the checksum verification yield that the packet is invalid, it is discarded and not further processed. If the packet is valid and its payload is within the required receive window, then it is written to the RX Buffer and the application is notified. If the received data is valid but out of order, then the data is written to the RX Buffer and processed as described in Section 4.3.3.

The implementation of the RX Engine clearly separates the data plane consisting of checksum validation, header parsing, and dropping of invalid packets from the control plane, which consists of the two FSMs that check the packet against the current state, update it if necessary, control timers, and trigger events.

4.2.9 TX Engine

The TX Engine is responsible for the transmission of TCP segments, which is always triggered by an event. Events originate, for instance, from the application requesting a data transmission, from the timers triggering retransmission, or from the application initiating a new connection. As the RX Engine, the TX Engine consists of multiple pipeline stages building a data-flow, see Figure 4.5.

The first stage is a finite state machine that processes the incoming events. Since the session ID is part of the event, the data structures can be immediately queried for all the necessary metadata required to generate the packet. This metadata is then forwarded to the TCP, TCP pseudo-, and IPv4 header modules to generate the corresponding headers. The construction of these headers requires a reverse session lookup to determine the source and destination IP addresses and TCP ports. After the payload is fetched from external

\(^1\)TCP options are currently not supported and are ignored.
memory, it is appended to the generated TCP header. In the next stage, the TCP header is extended with the pseudo-header to calculate the TCP checksum. As on the RX path, this incurs a store-and-forward of the whole packet. After removing the pseudo-header, the last stage of the pipeline prepends the previously generated IPv4 header to the packet and forwards it to the *IP Output Handler*.

### 4.2.10 Application Interface

There is a separate application interface for the RX and TX path, and each of them is separated into a control and data path, as shown in Figure 4.6.

On the RX path, the application can put a TCP port into listening state through the `listenPortReq` interface and is notified about the port state change on the `listenPortRsp` interface. Through the `notification` interface, the application is informed either about data available in the *RX Buffer* or connection termination by the other end. To retrieve data from the *RX Buffer*, the application issues a request to the `rxDataReq` interface containing the session ID and length to be retrieved. This request is answered by the TCP module by providing the stream of data on the `rxDataRsp` interface.

The application can open active connections through the `openConReq` interface providing the IP address and TCP port of the destination. Through the `openConRsp`, it will then either receive the session ID of the new connection or be notified that the connection could...
4.2. TCP Architecture

not be established. The application can close a connection by issuing the session ID to the closeConReq interface. To transfer data over an existing connection, the application has to provide the session ID, the length, and the payload to the txDataReq interface. For each requested transfer, the TCP module will return a response on the txDataRsp interface indicating potential errors and the remaining buffer space for that connection.

4.2.11 Life of a Packet

This section describes the life of a packet for the typical three-way TCP handshake when a session is set up. It highlights many different event types and shows the interaction between the RX and TX Engine. In this scenario, the FPGA acts as a server and listens on a specific port while another device connects as a client to that port.

1. A SYN packet arrives from the Ethernet Network Interface.
2. The IP Input Handler determines that this is a valid IP packet and checks its IP checksum.
3. It then establishes that this is a TCP segment and forwards it to the TCP module.
4. Within the **TCP** module, the TCP pseudo-header is first constructed and the TCP checksum is verified.

5. Subsequently, the destination port is queried from the *Port Table* and verified that it is in the LISTEN state.

6. Then, a new entry is created in the *Session Table* and the corresponding session ID is returned.

7. The **RX Engine** queries all meta information from the **TCP State** and **SAR Tables**. After ensuring that this is a new connection, the state is transitioned from LISTEN to SYN-RCVD.

8. The **RX Engine** initializes the **SAR tables** and triggers a SYN-ACK event.

9. The SYN-ACK event passes through the **Event Engine** and is forwarded by the **ACK Delay** module, without delay, to the **TX Engine**.

10. The **TX Engine** extracts the session ID from the SYN-ACK event.

11. Using it as a key, the **TX Engine** then queries all necessary meta information to construct a SYN-ACK segment.

12. It then computes and inserts the TCP checksum as well as part of the IP header with IP addresses and TCP ports.

13. The packet is streamed to the **IP Output Handler**, which computes and inserts the IP checksum. It also queries the MAC address corresponding to the destination IP address from the ARP Table and prepends the MAC header.

14. An ACK packet arrives to complete the handshake.

15. Again, the **IP Input Handler** validates the packet as a TCP segment.

16. The segment is then forwarded to the **TCP** module, where the **RX Engine** parses and verifies the TCP checksum for correctness.

17. The **RX Engine** queries the *Port Table* to ensure that the port is still open.

18. Then, the session ID is retrieved from the *Session Lookup*.

19. The **RX Engine** queries the state and meta information from the **TCP State** and **SAR Tables**, respectively. After verifying that the processed ACK is replying to the previously sent SYN-ACK, the state is transitioned from SYN-RCVD to ESTABLISHED. Pointers in the **SAR Tables** are updated and written back.

20. The connection is now successfully established and data can be exchanged.
4.3 Key Characteristics

This section expounds the key characteristics in our architecture to meet the driving requirements, which include 10 Gbit/s line-rate, scalability to high session count, out-of-order processing, high-level synthesis, and support for flow-control.

4.3.1 10 Gbit/s line-rate support

The TCP module was designed to process traffic at high data rates with 10 Gbit/s link-rate independently of the segment size which required a careful investigation of all data paths. To ensure this, we have designed the data paths between all modules in a data-flow fashion with a 8 B data bus clocked at 156.25 MHz. Furthermore, we checked that external memory access requirements were met. Roughly speaking, we require twice the incoming and outgoing throughput in memory bandwidth, as every packet in RX and TX direction is read and written once. Therefore, the requirement amounted to 40 Gbit/s for the memory access which is less than 17% of the total of 239 Gbit/s of theoretical bandwidth that the two 64 B 932 MHz SODIMM interfaces found on the VC709 board provide. Since TCP segments might be stored at any byte-offset and the length can vary to a great extent, the actual required bandwidth might be significantly higher depending on the workload.

Finally, the most critical aspect of the design is the shared central data structures, as for each segment, they have to be accessed and sometimes updated from various modules simultaneously. It is crucial to verify that even in the worst case, this process does not constrain the bandwidth. We first consider the available time budget and then evaluate whether this requirement can be met for the most contentious data structures: minimum Ethernet frames together with preamble and inter-frame gap amount to 84 B on the wire (64 B + 8 B + 12 B). With that, the highest possible packet rate is around 15 million packets per second (Mpps). Given a data bus width of 8 B and a 156.25 MHz clock frequency, the TCP module has to accept a new segment every 11 cycles, which determines the upper limit for all connection state processing. We thus have to ensure that all sub-modules have timely access to the state information and can complete their processing within the given time limit.

The modules in which the highest contention occurs are the TCP State, RX SAR and TX SAR Table. As seen in Figure 4.2, they are accessed from the two protocol engines and the application interfaces. As explained above, we need to service all accessing modules
in the budgeted 11-cycle window. For this, we have detailed all the required accesses in Table 4.2. The **TCP State Table** is accessed from the **RX Engine**, which performs a *rmw* operation and from the **TX App If**, which reads the state of a connection before sending data out. Because these two accesses might happen concurrently, the entry which is undergoing a *rmw* operation has to be locked to ensure consistency. As listed, the locked access from the **RX Engine** takes five cycles. Assuming the worst case in which the **TX App If** accesses the same entry simultaneously, an additional two cycles have to be budgeted for a read access. This brings the total number of cycles per packet to seven, which is well within budget.

The **RX SAR Table** is more contentious as it is being accessed from three modules, namely the **RX Engine**, the **RX App If** and the **TX Engine**. The latter only does a read operation while the former two perform both *rmw* operations. However, as the modules all operate on separate fields of the data structure, locking during the *rmw* is not required and operations can be interleaved. As shown in the table, the three read operations are followed by the two write operations, which brings the sum to 11 cycles and exactly fits the budgeted window. Similarly, the **TX SAR Table** is accessed by the **RX Engine**, the **TX App If** and the **TX Engine**. Like the **RX SAR**, it can be accessed by all modules concurrently without jeopardizing consistency, as they are processing different fields of the data structure. The table shows the two read operations followed by the three write operations, taking in total nine cycles, which is well within the budget. With that, all data paths of the system are designed to sustain the line-rate requirement.

### 4.3.2 Scalability

As mentioned in Chapter 3, a critical aspect of our architecture is that the resource requirements scale linearly with the number of supported sessions such that we can support deployments in data centers where thousands of servers communicate directly with each other. All tables (**TCP State**, **Port**, **SAR**) and **Timers** hold one entry per connection, as described in Section 3.3; therefore, they scale linearly. Further, the hash table used for the session lookup data structure also scales linearly with the number of sessions. Thus, scaling is linear with the number of supported sessions and limited by the amount of BRAM inside the FPGA while logic resource usage will remain mostly constant.
Specific to TCP is the requirement for receive and transmission buffers, which are advertised during the connection setup. This means with increasing session count, more buffer space is required. The packet buffers are stored externally, and, as previously determined, require $64\,\text{KB} \times 2 \times \text{number of sessions}$. Even for 10,000 sessions, this amounts to 1.3 GB which can easily be met with one 2 GB SODIMM.

### 4.3.3 Out-of-Order (OOO) Packet Processing

Although packet reordering and single packet loss are less common in modern networks, especially in the data center, our design supports as an option out-of-order reception of TCP segments. The OOO processing occurs in the *RX Engine* and makes use of the *RX Buffer* and *RX SAR Table*. The OOO segments are arranged into OOO blocks that consist of two pointers: the length of the block and the offset of the block from *received*, as seen in Figure 4.3, which in essence equates to the sum of the sequence (SEQ) number and payload length of the previous in-order segment\(^2\). Each pointer is 16 bits by default, which is less than the 32 bits of SEQ number required otherwise. This saves 32 bits per block, which is significant across 10,000 sessions. The *RX Engine* expects the SEQ number of each packet it receives to be the same as *received*. If this is not the case, the packet is considered to be out-of-order and the following steps are executed:

\(^2\)Wrap-around of SEQ numbers and pointers are being considered.
1. If the SEQ number is between \textit{read by app} and \textit{received}, then the packet is dropped as it has already been received.

2. Otherwise, the OOO segment is accepted. The payload is written directly into the \textit{RX Buffer} and the two pointers to the block are written to the \textit{RX SAR Table}.

3. If OOO segments overlap or are adjacent to previously received OOO data, then the pointers are updated accordingly.

Similarly to the standard Linux TCP/IP stack, the OOO processing functionality in our design is highly customizable as the maximum number of OOO segments and the maximum distance to \textit{received} are defined through compile-time parameters. This way it is possible to tweak the OOO processing to a particular application and the available on-chip memory, or omit it entirely.

4.3.4 Congestion Control

Flow control and congestion avoidance are essential features of TCP/IP. We have adapted some of the algorithms to map them more efficiently to the underlying hardware. For instance, we implemented a simplified variation of the \textit{Slow Start} and \textit{Congestion Avoidance} algorithm as detailed in RFC5681\cite{MPE09} with an aim to limit the design’s complexity. As an optimization, the so-called initial \textit{Congestion Window} is set to $10 \times \text{Maximum Segment Size}$ as proposed by \cite{DRC10}. The \textit{Congestion Avoidance} algorithm, which takes over after \textit{Slow Start}, increases the window size in incremental steps instead of deploying one of the more elaborate approaches used in real-world deployment such as TCP New Reno, TCP BIC, TCP CUBIC or Compound TCP. We defer a hardware-conscious implementation of these algorithms to future work.

Generally, packet loss is detected through the retransmission timers that time-out and trigger a retransmission event. To avoid any false positives, the time-out periods are conservative, increasing the time it takes to detect packet loss. In our implementation, we also support the \textit{fast retransmit/fast recovery} algorithm, as proposed in RFC2581 \cite{MPW99}. The purpose of this algorithm is to detect and repair packet loss immediately. In this algorithm, the receiver sends an immediate duplicate acknowledgment whenever it receives an out-of-order segment, an indicator for packet loss. On the sender side, the arrival of three consecutive duplicate acknowledgments is treated as an indication that a packet has
been lost and triggers a retransmission event without waiting for the retransmission timer to expire. This means the algorithm can significantly decrease the retransmission latency.

Finally, to avoid the so-called small packet issue, we implement Nagle’s algorithm [Nag84]. When transmitting small packets, the overhead of the packet headers is drastic. In the extreme case of transmitting a single byte, the packet size is 41 B (1 B data, 40 B header), leading to an overhead of 4,000%. This overhead not only reduces the network bandwidth utilization, but can also incur congestion and result in packet loss and retransmissions.

To reduce this overhead and avoid congestion, Nagle’s algorithm aims to maximize the amount of payload transmitted with each segment. The algorithm works as follows: if there is already unacknowledged data in-flight and the payload to be transmitted is smaller than the Maximum Transmission Unit (MTU), it delays the transmission in the hope that more payload has to be transmitted on the same connection shortly. The transmission is delayed until all data in-flight is acknowledged or enough data is available to send a complete MTU. Whenever there is no unacknowledged data in-flight, the payload is transmitted immediately, independent of its size. Depending on the communication pattern, the algorithm can significantly improve the network utilization with a minor increase in latency.

4.4 Optimizations

TCP/IP makes very few assumptions about the underlying network and provides useful guarantees such as reliable transmission, in-order delivery or control flow. Our implementation follows the TCP specification as close as possible to provide the same guarantees and rich functionality. However, targeting data center applications, we can rely on a network that is in a controlled environment and provides stronger guarantees than the Internet. Recent work [PLL+15, DSC+15] has shown that data center networks exhibit infrequent reordering of messages, have a fixed length network topology, and provide high reliability. In such a network, some functionality of the TCP protocol, which is based on conservative assumptions about the underlying network, can easily be relaxed without giving up any guarantees of TCP. The intuition here is that by relaxing or disabling certain features, latency and throughput of data center applications can be further improved.

In addition to a more reliable network, in a data center environment, we also have more knowledge about the application itself. Using this knowledge allows us to make certain
assumptions and introduce a number of optimizations to reduce the latency and minimize access to the DDR memory by tailoring the TCP/IP stack to the application.

We use two example applications, a key-value store [IABV13, CLW+13] and a low-latency consensus protocol [ISAV16] implemented on the FPGA, to expound the challenges of integrating the application and the network stack, the assumptions we make about the application, and the optimizations we introduce.

4.4.1 Challenges in Practice

Two challenges have to be addressed when integrating an application with the network stack. First, the memory bandwidth available on an FPGA board, in our case a Xilinx VC709, has to be shared among the application and the TCP/IP stack. As shown, the stack uses DDR memory to buffer the payload of incoming and outgoing packets. When adding an application, which also requires access to the DDR memory, the memory bandwidth can easily become a bottleneck. Second, although our hardware implementation has a path latency of 1-5 µs depending on packet size, data center applications commonly have to meet very strict latency requirements. In particular, the service provider and the client have a service level agreement which clearly defines median and, especially, tail latencies.

An additional issue we observe when using the TCP/IP stack for inter FPGA communication is that, despite having a low latency implementation in hardware, latency is introduced by the TCP protocol itself in the form of Delayed Acknowledgments, conservative timers, and Nagle’s algorithm.

4.4.2 Assumptions

For the two example applications, we make the following three assumptions: First, client requests fit into a single maximum transmission unit (MTU). Second, clients are synchronous, which means they do not have more than one outstanding request. Third, the application also implemented on the FPGA is designed to process data at line-rate.

The first assumption limits the client request size to 1500 B given the maximum size of the Ethernet frame, respectively 9000 B for jumbo frames. Key-value stores are often used for read-intensive workloads [AXF+12]. In case of a GET or similar read-only operation in other applications, the request only contains a single or few unique identifiers to retrieve the object(s). Therefore, these type of requests are not affected by this limitation. However,
this does not hold for requests that insert or update a value, for instance \texttt{SET} in a key-value store. For these requests the MTU limits the maximum value size. While this limitation seems quite restrictive, the analysis by Atikoglu et al. \cite{AXF+12} on real-world workloads has shown that, for most workloads, 90\% of the values are 500 B in size or less.

Second, clients are synchronous, which means they do not have more than one outstanding request. More specifically, this assumes an environment with many clients and that the next request depends on the current request. Note that a single request can retrieve multiple objects, such as in \texttt{MULTI-GET}. It is clear that, in this scenario, the utilization per connection is low, but given a large number of clients the application can still be fully utilized. A benefit of this assumption is that requests cannot be fragmented across segments in the case of re-transmission.

The third assumption takes advantage of the fact that it is likely that the application implemented on the FPGA is designed to operate at the network line-rate. If this is the case, payload written into the \textit{RX Buffers} is consumed immediately, keeping the buffers almost empty. In other words, the buffer capacity of 64 KB per connection is never utilized and the buffers could be replaced by a single FIFO buffering the payloads, which is immediately consumed by the application. The drawback of using a FIFO is that in the event the application cannot maintain line-rate processing packets might be dropped and have to be re-transmitted.

We recognize that these assumptions might not hold for all connections. In the case of the distributed consensus, multiple outstanding requests and data chunks split across multiple segments are transmitted between FPGAs. Similarly, in the case of the key-value store, some connections might execute \texttt{SET} operations with values exceeding the MTU size.

Regarding the data center network, we assume high reliability and infrequent reordering of messages, as shown by \cite{PLL+15, DSC+15}.

### 4.4.3 Nagle’s algorithm

As described in Section 4.3.4, we implement Nagle’s algorithm \cite{Nag84} to improve the network bandwidth utilization. The algorithm achieves this by delaying the transmission of small packets in the hope that more payload has to be transmitted on the same connection shortly. If this is the case, the amount of payload per packet can be increased, thereby reducing the overhead introduced by the packet header.
Chapter 4. Scalable TCP/IP Stack

It is commonly known [Che05] that, depending on the application, Nagle's algorithm can interact badly with the mechanism of Delayed Acknowledgments, see Figure 4.7. In practice this means that a data transmission is delayed by Nagle's algorithm but the application is not intending to transmit more data, which means the algorithm delays the transmission until all in-flight data is acknowledged. In the worst case, the receiver is at the same time delaying the acknowledgment for the data in-flight, hindering Nagle's algorithm from making progress. This interaction can lead to a significant increase in latency for certain segments and impact the overall performance.

To avoid this interaction, in most software stacks, Nagle's algorithm can be disabled through the TCP_NODELAY flag [lin, win]. For the client-facing communication, based on our assumptions, Nagle's algorithm is not triggered. But for inter FPGA communication, it is beneficial to disable the algorithm to reduce latency and avoid negative interaction with Delayed Acknowledgments. To give applications this option, we extended our high-level synthesis implementation with a compile-time TCP_NODELAY flag that disables Nagle's algorithm completely. In contrast to software stacks, it is not possible to choose this option at run-time. The advantage of applying this option at compile-time is that disabling Nagle's algorithm also saves hardware resources. As for software, disabling the algorithm delegates the responsibility to the application to avoid a large bandwidth overhead by transmitting very small packets.

Figure 4.7: Interaction between Nagle’s algorithm and Delayed Acknowledgment increases the latency of the response significantly.
4.4. Optimizations

(a) Acknowledgments are not delayed.

(b) ACK for request is delayed and piggybacked on the response.

(c) ACK for first and second request packet are merged together.

Figure 4.8: Interaction between client and FPGA without delaying acknowledgments (a), with piggybacking acknowledgments on data packets (b), and with merging delayed acknowledgments (c).

4.4.4 Delayed Acknowledgment

Similar to Nagle’s algorithm, the Delayed Acknowledgment described in RFC1122 [E.89] is a mechanism to increase the network bandwidth utilization. In TCP, data packets get acknowledged with an ACK packet, a simple control packet which contains no payload. However, ACK packets share the available bandwidth with packets containing data. The
mechanism of Delayed Acknowledgment tries to reduce the number of ACKs that are transmitted by delaying ACKs by up to 0.5 seconds, thereby creating the opportunity to either piggyback them with a data packet or merge two of them together. Since every data packet contains the ACK number in its header, a delayed ACK can be discarded if a data packet for the same connection is transmitted within the 0.5 seconds. Or in other words, the ACK is piggybacked onto the data packet, as illustrated in Figure 4.8b. A delayed ACK can also be merged with a later-generated ACK packet, reducing the number of ACK packets (Figure 4.8c). However, at most two ACKs can be merged, as specified in RFC1122.

In the context of inter FPGA communication, we considered removing the Delayed Acknowledgment logic completely to reduce the latency of acknowledgments. However, experiments have shown a significant reduction in goodput, especially for small packets, which are common in a distributed application where many control messages are exchanged. Instead of removing the delay, we reduced it to 64µs with the expectation that the application is sending a response within 5µs where the ACK can be piggybacked on.

4.4.5 Retransmission- & Probe-Timers

The original TCP specification has defined very conservative time-out values for retransmission, probing, and connection time-out. These values were defined for unreliable and slow wide-area networks (e.g. the Internet). In contrast, data centers deploy reliable high-speed links, which lead to more predictable round trip times (RTTs). To reduce the response time to failures, we adapted the time-out values according to our infrastructure, which is a common optimization. Our current implementation uses pre-defined time periods for given events and does not distinguish between different connections. However, in distributed FPGA-based systems, there are inter FPGA connections as well as connections to CPU-based systems. For the former, due to the low-latency communication, short time-out values result in better performance, while they might harm performance for the latter. To address this, a future extension of our implementation would be able to estimate the RTT for every connection and determine time-out periods based on this.
4.4. Optimizations

4.4.6 On-chip RX buffer

For the RX buffer, we can make two assumptions about the application. First, client requests will always fit into an MTU-size segment and therefore no reassembly of TCP segments on the RX path is required. Second, the application which is also implemented on the FPGA can consume data at line-rate. Based on these assumptions, the individual RX Buffers in DDR memory can be replaced by a single FIFO that holds a limited number of packets before they are consumed by the application (Figure 4.9). This optimization can be enabled through the `RX_DDR_BYPASS` flag and applies to all connections.

In case there are a number of connections, e.g., for inter FPGA communication, where these assumptions do not hold, the necessary reassembly is delegated to the application on the FPGA. Given that the segments are still delivered in-order and the number of connections between FPGAs is limited by the number of FPGA nodes, the required resources and complexity for the reassembly is minimal and scales linearly with the number of inter FPGA connections.

4.4.7 Reducing memory access to the TX buffer

Initially, buffering the payload on the TX path was a requirement for Nagle’s algorithm to accumulate payload and for the retransmission of payload in case of packet loss. When Nagle’s algorithm is disabled by the `TCP_NODELAY` flag, the TX Buffers are solely required for the case of retransmission. This change in requirements allows for another optimization.
Chapter 4. Scalable TCP/IP Stack

Instead of always writing the payload first to the TX Buffer, from where it is later read by the TX Engine, the payload can be directly forwarded from the TX App If to the TX Engine (Figure 4.9). To ensure that the payload can also be retransmitted if necessary, it is concurrently written to the TX Buffer in DDR memory. This optimization reduces the required memory bandwidth, since on transmission, data is only written to DDR memory, and on retransmission, the payload is only read from the DDR memory. This bounds the required memory bandwidth to 10 Gbit/s for transmission, which is a reduction of 50%. In addition to reducing the memory bandwidth, directly forwarding the payload to the TX Engine also reduces the latency on the TX path.

4.5 Evaluation

In this section, our implementation is evaluated in regards to throughput, latency and resource consumption. For the performance evaluation, we utilize the following setup: The presented network stack was implemented on a Xilinx VC709 board, which features a Xilinx Virtex7 XC7VX690T FPGA. The design utilizes two 932 MHz DDR3 SODIMMs with 4 GB each and one 10 G network interface. The network interface and TCP/IP stack are operating at a frequency of 156.25 MHz. In the experimental setup, the card is connected via a Cisco Nexus 5596UP switch to ten servers, each equipped with two 8 Core Intel Xeon E5-2630 v3 CPUs clocked at 2.4 GHz and 256 GB of main memory, running Linux (kernel 4.14) and an Intel 82599ES 10G Ethernet Controller. If not stated otherwise, each experiment was run for 120 seconds.

4.5.1 Throughput

To evaluate the throughput, we implemented an iperf\(^3\) compatible application on the FPGA, which consumes all incoming packets and can independently generate a stream of outgoing packets.

All measurements report application level throughput, which is the throughput that the application can effectively achieve. Obviously, its maximum value is lower than the actual link bandwidth of 10 Gbit/s and depends on the Maximum Segment Size (MSS), which affects the ratio between actually transmitted data and its overhead. The maximum

\(^3\)https://sourceforge.net/projects/iperf2/
4.5. Evaluation

Ethernet frame is 1500 B not considering jumbo frames. After taking the headers into account, this leaves a maximum MSS value of 1460 B that corresponds to actual payload. Using 1460 B as our default MSS and taking the packet overhead consisting of Ethernet, IP and TCP headers into account, we can compute the maximum possible TCP throughput to be:

$$\frac{10 \text{ Gbit/s} \times (1460 \text{B} \times 8)}{(1460 \text{B} + 24 \text{B} + 12 \text{B} + 40 \text{B}) \times 8} = 9.5 \text{ Gbit/s}$$

9.5 Gbit/s should be considered an upper bound since some segments, e.g. SYN, ACK, are not carrying payload and other non-TCP packets, like ARP, are also consuming part of the available bandwidth.

To evaluate the RX path, two servers are running iperf to generate traffic. For the TX path, we used four server machines to act as iperf servers and consume the traffic generated by the FPGA. The implementation of iperf uses one thread per connection, but its performance degrades rapidly when more threads than available cores are used. Therefore, we limit this experiment to a maximum of 64 connections corresponding to the total number of hardware threads available in the two server machines. Figure 4.10 shows the throughput measured by iperf running on the servers. Using two connections (two machines with one connection each), the theoretical peak bandwidth of 9.5 Gbit/s can
already be reached. With four and eight connections, there is some variation, which might be caused by scheduling overhead for the different threads. With increasing connections, this overhead can be hidden, since the load is distributed among multiple connections and threads. For eight to 64 connections, the RX bandwidth is around 1 Gbit/s, below the theoretical peak. We verified that no packets are dropped by the FPGA and, therefore, we consider in-cast congestion at the switch as a potential cause for the reduced bandwidth.

To overcome the limitation of iperf and evaluate the throughput for a higher number of connections, we implemented our own version of iperf that is optimized for many concurrent connections but cannot necessarily achieve a high throughput for a single connection. This implementation is based on golang that can handle many connections thanks to its lightweight threads, called goroutines. Using golang-iperf, we evaluated the throughput for 100 to 10,000 connections, see Figure 4.11. For 100 connections, the ten load-generating machines almost reach the peak bandwidth of 9.5 Gbit/s, but with an increasing number of connections the bandwidth stabilizes around 9 Gbit/s. For the TX throughput, we observe a similar behavior up to 6,000 connections. To determine the cause for the decline in TX throughput from 6,000 to 10,000 connections, we varied the number of machines that consume the traffic. We observed that with fewer machines, the decline shifts to the left.
4.5. Evaluation

From this, we conclude that for more than 6,000 connections the experiment is bound by the ten machines receiving the network traffic and not by the FPGA.

4.5.2 Latency

The latency of the two data paths was measured on the hardware itself. Figure 4.12 shows the latency with increasing payload size. Since the checksum computation on each path requires a store-and-forward of the complete segment, the latency increases linearly with the payload size. The latency on the RX path is slightly higher, since it requires a hash table and Port Table lookup, leading to more data structure accesses than on the TX path. The RX_DDR_BYPASS optimization that uses on-chip memory for the RX Buffer instead of DDR leads to a reduction between 0.5 and 1.6\(\mu s\). Similarly, using the TCP_NODELAY flag the latency of the DDR memory is avoided on the TX path by directly forwarding the payload to the TX Engine. As a result, the latency is reduced by 0.3 to 1.5\(\mu s\). Ignoring the clock cycles for the store-and-forward, the processing time on the RX path with RX_DDR_BYPASS is a constant 85 cycles and the TX path with TCP_NODELAY 70 cycles. Overall, these optimizations provide a latency reduction of up to 50%.

Figure 4.12: Data path latencies in the TCP/IP stack.
Figure 4.13: Impact of the available memory on the processing rate of different TCP configurations. The dotted line indicates the theoretical peak throughput.

### 4.5.3 Reducing DDR memory access

Figure 4.13 shows the bi-directional throughput for different configurations of the TCP/IP stack. For this evaluation, an echo server was implemented on the hardware to guarantee equal load on both data paths. When the RX and TX Buffer are mapped to two different memory channels, the theoretical peak throughput can be achieved for a payload of 256 B or larger. However, if the two buffers are mapped to the same channel sharing the available bandwidth, the stack can no longer process packets at line-rate independent of the payload size. Interleaving the accesses to RX and TX Buffers of different connections results in a random access pattern that impacts the achievable memory bandwidth. In addition to that, in our implementation, the SEQ number maps directly to the byte-offset in the buffer, which can lead to unaligned accesses incurring additional overhead. Thus, despite the theoretical bandwidth of 119 Gbit/s, in practice the bandwidth of a single channel is not sufficient.

One major goal for the `RX_DDR_BYPASS` and `TCP_NODELAY` optimizations is to reduce the memory bandwidth and footprint. As Table 4.3 shows, when both optimizations are enabled, the required bandwidth drops by a factor of 4x. As a result, this configuration
4.5. Evaluation

Table 4.3: Resource usage for different configurations of the TCP/IP stack supporting 10,000 connections.

<table>
<thead>
<tr>
<th>RX_DDR_BYPASS</th>
<th>TCP_NODELAY</th>
<th>CLB</th>
<th>BRAM</th>
<th>Mem. alloc.</th>
<th>Mem. bw.</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>10,603</td>
<td>318.5</td>
<td>1,300 MB</td>
<td>40 Gbit/s</td>
</tr>
<tr>
<td>-</td>
<td>x</td>
<td>10,658</td>
<td>316.0</td>
<td>1,300 MB</td>
<td>30 Gbit/s</td>
</tr>
<tr>
<td>x</td>
<td>-</td>
<td>10,168</td>
<td>316.0</td>
<td>650 MB</td>
<td>20 Gbit/s</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>10,049</td>
<td>314.0</td>
<td>650 MB</td>
<td>10 Gbit/s</td>
</tr>
</tbody>
</table>

Table 4.4: Resource usage by component and in percentage of total resources available on the device.

<table>
<thead>
<tr>
<th>Component</th>
<th>CLB</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network interface</td>
<td>2,975</td>
<td>2.8%</td>
</tr>
<tr>
<td>Memory interface</td>
<td>21,184</td>
<td>19.6%</td>
</tr>
<tr>
<td>TCP/IP stack</td>
<td>10,603</td>
<td>9.8%</td>
</tr>
<tr>
<td>TCP module</td>
<td>6,810</td>
<td>6.3%</td>
</tr>
</tbody>
</table>

can achieve line-rate processing for payload sizes of 256 B and larger while using only a single memory channel, see Figure 4.13. As an additional benefit, these optimizations free up a memory channel that can be allocated to an application implemented on the FPGA.

4.5.4 Resources

Table 4.4 lists the resource usage of our prototype in both absolute and relative terms. Network and memory interfaces are using existing Xilinx IP cores. The TCP/IP stack contains the following modules: *IP Input Handler*, *IP Output Handler*, *ARP*, *ICMP*, and *TCP*. The resources for the last one are also listed separately. For the 10,000 connections, the TCP/IP stack uses 318.5 BRAMs, which account for 21.7% of the available BRAM capacity. Since BRAMs are mostly used for the data structures, their usage can be reduced by lowering the number of supported sessions, see Table 4.5. Given this, we argue that the design leaves sufficient space on the given device to implement large-scale applications, even for a large amount of concurrently active sessions. As Table 4.3 shows, the different optimizations have no significant impact on the resource usage.
Chapter 4. Scalable TCP/IP Stack

Table 4.5: Resource scaling with increasing number of supported connections.

<table>
<thead>
<tr>
<th>#Connections</th>
<th>500</th>
<th>1K</th>
<th>2K</th>
<th>4K</th>
<th>8K</th>
<th>10K</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRAM</td>
<td>116</td>
<td>118.5</td>
<td>131.5</td>
<td>157</td>
<td>211.5</td>
<td>318.5</td>
</tr>
</tbody>
</table>

4.6 Enabling Applications on the FPGA

The main goal for this TCP/IP implementation is to enable data center applications and provide a flexible network stack implementation that can be used by different applications. To facilitate this, we made the implementation presented in this chapter available as open source.

4.6.1 cloudFPGA

IBM’s cloudFPGA platform [WAHH15, WPAH16, AWH+17] aims to make FPGAs accessible to cloud users as a standalone resource. To achieve this goal, the researchers propose to break the strict coupling of FPGAs and CPUs common in PCIe-attached deployments by disaggregating FPGAs and deploying them as network-attached accelerators.

As a first step, they extend Open Stack, which is a framework for cloud resource management, with a new accelerator service that manages FPGA resources. Through this, FPGAs become a standalone resource in the data center that can be scaled independently of CPU, memory, or storage. Apart from allocating single FPGAs, the user can define a so-called FPGA Fabric. An FPGA Fabric is a set of FPGAs which are connected over a user-specified logical network topology. User-defined fabric descriptions are passed to the Open Stack accelerator service, which is responsible for the optimal allocation of resources.

To make the FPGA a network-attached accelerator, they implement a Network Service Layer (NSL) in hardware, see Figure 4.14. The NSL contains the network stack presented in this chapter. They simplified the interaction between the application and the network stack, by implementing a simplified interface that exposes fewer FIFOs to the user logic. Additionally, the NSL implements a Management Layer that interacts with Open Stack to allocate and manage the FPGA as a resource.

4https://github.com/fpgasystems/fpga-network-stack

68
4.6. Enabling Applications on the FPGA

Finally, they built FPGA boards equipped with Kintex UltraScale FPGAs and a carrier board that can host 32 boards. Two carrier boards fit into a 2U chassis, which means that a full rack can host up to 1024 FPGAs and a total of 16 TB of DDR4 memory. Each FPGA in this rack is connected over 10 Gbit/s TCP or UDP to the network.

They evaluated their platform using a text analytics application that uses regular expression cores on the FPGA and have shown a more than 10 x improvement in latency and throughput compared to a software and a PCIe-attached FPGA solution.

4.6.2 Caribou

One of our own efforts is Caribou [ISAV16, ISA17], an intelligent distributed storage layer. Caribou is deployed in the context of distributed database systems where storage and processing nodes are commonly separated. Data stored in the storage nodes is shared across all processing nodes. This provides flexibility and simplifies load balancing at the processing layer. However, it incurs significant data movement between the two layers.

Caribou (Figure 4.15) implements FPGA-based intelligent storage nodes where each node
provides access to DRAM over a simple key-value store interface. The storage nodes in Caribou address the data movement overhead between the storage and processing layer by providing near-data processing capabilities. Fault tolerance is achieved through replication across the storage nodes.

Most existing key-value store implementations [IABV13, BKL+13, XLJ+16] for FPGAs target caching use cases where hash collisions do not have to be resolved. In the case of storage, collisions have to be handled. Therefore, in Caribou, we implemented a cuckoo hash table that handles collisions. Further, a slab-based memory allocator ensures efficient use of memory even when supporting multiple value sizes and enables re-use of deallocated memory regions.

To provide fault tolerance, Caribou implements Zookeeper’s atomic broadcast (ZAB). ZAB is roughly equivalent to Paxos, but its protocol is significantly easier to understand, since it makes simplified assumptions about the communication. Specifically, the communication channels have to be lossless and strongly ordered, as provided by TCP. The ZAB hardware implementation is tightly integrated with the network stack. Latency of a consensus round as measured by the leader is below $10\mu$s, which is more than an order of magnitude lower than software-based implementations such as Libpaxos3 [lib] and Etcd (Raft) [etc].

Caribou enables near-data processing through selection push down to the storage nodes. This results in a reduction of the data that has to be transferred from the storage to the processing layer. Queries sent to the storage nodes can contain selection predicates.
4.6. Enabling Applications on the FPGA

The predicates are used to parametrize the selection logic on the FPGA at run-time to the query. Selection predicates can be applied to structured data or unstructured data in the form of text. For structured data, Caribou supports comparison-based selection with the following functions: ==, !=, <, >. For unstructured data, it supports substring matching, as in the SQL LIKE operation.

Many of the optimizations introduced in Section 4.4 were inspired by Caribou. In particular, RX_DDR_BYPASS and TCP_NODELAY free up memory bandwidth for the key-value store deployed on the same FPGA board. Moreover, the lower communication latency reduces the required state that has to be maintained and stored by the ZAB module. TCP/IP is also used to expose a key-value store interface to the clients. Using TCP/IP has the benefit that commodity software and hardware can be used on the client side. In terms of performance, the system can reach up to 11 Mio. requests per second, only being bound by the network link. Thanks to the tight integration (Figure 4.16) of the network stack, the key value store, and the ZAB module, the latency observed at the client is as low as 30µs.

4.6.3 Memcached Accelerator

In the commercial space, LegUp Computing [CLL+18] developed a memcached accelerator targeting Amazon’s F1 FPGA cloud instance. Memcached implements an in-memory hash table to cache data objects. When the hash table is full, it evicts the least recently used (LRU) item on subsequent inserts. Memcached is commonly used as a cache for database
systems. When the application wants to retrieve an object, it queries first memcached, and only in case of a miss does the database have to be accessed. Since memcached holds the data in memory, it can return the object rapidly, avoiding having to query the database. This reduces the latency involved in answering the request and the load on the database.

The accelerator of LegUP Computing builds on existing work that presented efficient hash-table implementations for FPGAs [IABV13, CLW+13] and combines it with the open source network stack presented in this chapter. As shown in Figure 4.17, the accelerator is not directly connected to the network. However, by offloading network processing, they avoid the inefficiencies of the OS network stack and achieve a tight integration between the network stack and the memcached accelerator, resulting in low latency.

To efficiently move the raw Ethernet frames received by the network card to the FPGA accelerator and to move the generated response frames back, they implement a framework on top of DPDK [inta]. By using DPDK, which runs in user space, the OS software stack is bypassed. They report that passing Ethernet frames from the network to the FPGA or vice versa takes between 20 and 50\(\mu s\) per direction. Matching the throughput of existing work, their accelerated memcached implementation achieves up to 11 Mio. requests per second, which they claim to be the fastest public cloud memcached implementation.
Related Work

There have been both commercial and academic TCP/IP implementations for reconfigurable logic in the past. On the academic front, Dollas et al. [DEK+05] presented an open TCP/IP architecture in 2005 with limited throughput (estimated at 350 Mbps) and limited connection support (31 active and 16 passive). In their implementation, buffers are shared between all protocols and connections, which limits parallelism. Wu et al. [WC06] and Chung et al. [CLL+07] presented TCP Offload Engines (TOEs) using a combined hardware-software solution. A PowerPC CPU handles most of the TCP processing, but offloads the checksum calculation. Address Resolution Protocol (ARP), Internet Control Message Protocol (ICMP) and IP packets are processed by the FPGA fabric. A complete TOE for FPGAs without processor assist was presented by Uchida [Uch08]. Targeting embedded devices, resource usage was paramount, resulting in a limited feature set. Most recently, Ji et al. [JH11] presented a TOE with a centralized scheduler, which is estimated to sustain 4 Gbit/s for receiving and 40 Gbit/s for transmitting data for payload sizes, corresponding to the maximum segment size of 1460 B. This implementation targets an asymmetric workload such as video on demand where large-sized packets are transmitted. In contrast, our data-flow-based approach supports 10 Gbit/s full-duplex, even with worst case minimum-size packets of 64 B, thereby enabling much higher packet rates. Furthermore, we support a more complex and deeper packet buffering system, which offers 64 KB per session as opposed to the single buffer used for all connections. This provides more flexibility and less tight coupling to the application. Out-of-order (OOO) processing is equally supported with minor differences on the data structures; we store only offset and length for OOO blocks, instead of all SEQ numbers explicitly, and write the OOO blocks directly to their position within the receive buffer instead of sorting through received segments when access is required. Some other work has focused on parts of a TCP/IP stack. Perrett et al. [PD11] presented an ARP module. Herrmann et al. [HPdF+09] showed a User Datagram Protocol (UDP) stack targeting 1 Gbit/s.

In regards to commercial systems, multiple TCP/IP stacks for FPGAs are available from vendors such as Intilop, Fraunhofer HHI, Dini Group and PLDA [pld, LBT+, intb, din]. Up until 2014, all of the available TOEs were ultra-low latency with low session count, typically well below 256 concurrent sessions, with the key driving application being high-frequency trading (HFT). To minimize latency, these stacks are typically constrained in session support, as high session counts directly impact latency two-fold: Firstly, for every
session, packets need to be buffered for both packet reception and transmission. Assuming a typical TCP window size of 64 KB and given that standard FPGAs are typically limited to tens of megabits in on-chip memory (for example, 52.9 Mbits for the Virtex 7 used in this work), packet buffering for anything more than a few dozen sessions needs to be moved off-chip into external DRAM. This has an adverse effect on latency (we have measured the overhead of external packet buffering to be over 600 ns and increasing with segment size. Secondly, to associate an incoming or outgoing packet with a session, one needs to perform a lookup using the four-tuple of source and destination IP address, source and destination TCP port. Lookup problems can be concluded within one clock cycle when ternary content addressable memory (TCAM) style architectures are deployed. In essence, one stores every entry of the session table inside the chip together with a comparator to the incoming four-tuple that enables a comparison of all entries within a single cycle. However, these architectures become prohibitively resource-expensive for high session counts.

At the beginning of 2014, Intilop announced a high session count variant with support for up to 16,000 sessions, which is probably the closest-related available TCP/IP stack in terms of feature set. A detailed architectural comparison is at this stage not possible as Intilop has not disclosed the design. However, one obvious key differentiator is that our implementation is open source and, with the use of high-level synthesis, it can be easily adapted or extended with new functionality.

4.8 Summary and Discussion

In this chapter, we presented a TCP/IP implementation based on the scalable architecture introduced in Chapter 3. The implementation is able to process packets at 10 Gbit/s full-duplex, while handling thousands of concurrent connections. By design, the on-chip memory footprint scales linearly with the number of supported connections. While we showed up to 10,000 connections on a two-generation-old device, we estimate that the latest generation Ultrascale+ devices – which introduce a new type of on-chip memory, called UltraRAM – are able to support up to 100,000 connections. While the core of the design is based on our scalable architecture, the implementation also encompasses functionality such as checksum calculation, flow control and out-of-order processing to fully support TCP. The design was implemented almost entirely using C++, which shortened development significantly, simplified verification, and provided greater design flexibility.
Initially, we evaluated the performance, compatibility and robustness of our design on a Xilinx VC709 development platform using software tests and microbenchmarks. However, the integration with actual applications has further strengthened the robustness of our implementation and inspired the optimizations described in Section 4.4. As a result, our implementation evolved from a research prototype to a reliable building block that enables a multitude of accelerators that are directly attached to the network, benefiting from the low-latency and high-bandwidth communication.

Future work includes adding support for TCP options, especially the TCP window scale option. This option becomes more relevant with increasing bandwidth. Essentially, the receive window determines and limits the amount of unacknowledged data in-flight. With the default size of 64 KB and a 10 Gbit/s link, a single connection can saturate the link if the round trip time is below 50 $\mu$s. However, for higher bandwidths, the round trip time has to be even lower; therefore, it is not possible to saturate a high bandwidth (100 G) link with a small number of connections. The window scale option addresses this issue by supporting and advertising receive windows up to 1 GB in size.

With increasing data path, for instance 64 B, the smallest possible packet can arrive in a single cycle. As we have seen, currently it takes multiple cycles to update the state for each packet. This means that, at a higher bandwidth (data path width), the rate at which state can be updated can potentially limit the packet rate for small payloads. To reduce the access latency to the State Tables, the most recently used entries could be cached in the RX or TX Engine as long as consistency can still be guaranteed.
FPGA-based RDMA Network Card

One of the key benefits of Remote Direct Memory Access (RDMA) is its low latency, which is achieved by removing the CPU from the data path and offloading all network processing to the network card, which has direct access to the host memory. The bandwidth of RDMA is approaching the bandwidth of inter-socket links, such as QPI (around 25 GB/s). This has prompted the redesign of algorithms and applications to scale beyond multi-socket systems [BLAK15]. Initially designed for specialized networks like Infiniband, RDMA is becoming increasingly important over Ethernet, especially as network capacity grows from today’s 10 G and 40 G to 100 G and 400 G in the near future. Processing packets at these rates in software (involving the CPU) is difficult; hence the need to offload network processing entirely to the network card.

In practice, from the application perspective, RDMA offers the attractive option of reading from or writing to remote memory with very low latency and high bandwidth. RDMA has thus been used to build distributed in-memory storage systems, e.g., RAMCloud [OAE+10], FARM [DNCH14], RStore [TSM+15], and key-value stores [MGL13, KKA14]. It is also used as an interconnect in tightly integrated appliances, such as Oracle’s ExaData, as well as in the cloud [LRX+17]. However, it remains a challenge to fully leverage the benefits of RDMA in distributed systems. As a result, it is necessary to carefully design such applications [FA09, BCG+16] and rethink algorithms [PH15, BMS+17].

To some degree, RDMA is the result of the continuous effort to offload network processing, culminating in the full offload of the protocol stack including direct access to the host.
memory. While this continuous increase in offloaded functionality has improved many key metrics, it has also eliminated the opportunity to implement new functionality or optimizations in the protocol stack. On the contrary, it has left the developer with the task of tailoring his application towards the underlying hardware. In this chapter, we introduce, Wisent, an open source\textsuperscript{1} RDMA network card enabling researchers and developers to explore optimizations and extensions at the protocol level.

In detail, Wisent is an FPGA-based RoCE network card consisting of a RoCE stack for FPGAs and a driver so that user applications running on the CPU can interact with it. The RoCE stack supports the one-sided \texttt{RDMA READ} and \texttt{RDMA WRITE} commands, operates at a rate of 10 Gbit/s and can handle up to 16,000 concurrent connections, called queue pairs in RDMA. Wisent can access up to 32 GB of host memory directly through its DMA engine, which can be made accessible to remote nodes over the network. Through our driver, user applications can configure the network card and issue RDMA read and write commands. The implementation presented here follows the RoCE specifications as closely as possible, but thanks to the use of a high-level language (Vivado HLS \cite{Xila}), the code can be modified with reasonable effort.

As we will show in Chapter 6, Wisent can be used as a research platform to explore compute offload to the network card. Additionally, it provides FPGA-based applications with the capability to directly access and process data stored on (multiple) remote machines, thereby opening up many possibilities for using FPGAs in a wider context.

As our experimental evaluation shows, user applications can access remote data in as low as 6.8 $\mu$s for read and 4.5 $\mu$s for write operations. In comparison, a commercial RDMA network card is able to execute the same operations in around 3.5 $\mu$s but without the ability to change or add extra functionality that our design provides.

5.1 Background

5.1.1 Remote Direct Memory Access (RDMA)

Remote Direct Memory Access (RDMA) is a mechanism to directly access data in the main memory of a remote machine. The accessible memory has to be pinned and registered in the form of a buffer with the NIC before it can be accessed by the NIC \cite{FA09}. Access\textsuperscript{1}

\textsuperscript{1}https://github.com/fpgasystems/fpga-network-stack
to memory takes place through DMA from the NIC without CPU or OS involvement for the data transfer (there could be CPU involvement for setting up or completing the operation). When RDMA operations involve the CPU at both the sender and the receiver, they are called two-sided (RDMA SEND and RDMA RECEIVE). Two-sided operations reflect many common protocols such as TCP or UDP, which have a similar send-receive model. One-sided operations, RDMA WRITE and RDMA READ, access memory directly from the NIC by performing DMA to a remote virtual memory address and, in case of RDMA READ, also to a local address to store the retrieved data. One-sided communication is made possible through: (1) offloading the network processing completely to the network card and (2) allocating memory buffers for each queue pair a priori, pinned in memory, and accessible by the network card through its DMA engine.

![Protocol Layers](image_url)

**Figure 5.1:** InfiniBand and RoCE protocol layers. RoCE v1 is indicated in the Ethernet frame by the EtherType 0x8915, RoCE v2 is indicated by the UDP destination port 4791.

### 5.1.2 RDMA over Converged Ethernet (RoCE)

While originally RDMA was limited to specialized network fabrics such as InfiniBand, it is now also possible to use it over commodity Ethernet networks. *Internet Wide-area RDMA Protocol (iWARP)* specifies how RDMA packets can be encapsulated into TCP/IP. However, this encapsulation requires the processing of TCP, which introduces additional overhead and inefficiencies. More recently, *RDMA over Converged Ethernet (RoCE)* was introduced where RDMA packets are encapsulated into Ethernet frames or UDP packets. The key advantage of using Ethernet is the reduced cost and the possibility to deploy it on the existing network infrastructure. The UDP-based version, called RoCE v2, is routable by encapsulating IB packets into IP/UDP packets. The protocol layers in InfiniBand and the two RoCE versions are illustrated in Figure 5.1. Converged Ethernet at scale can suffer
from head-of-line blocking; Zhu et al. [ZEF+15] proposed modifications to the protocol to address this. SoftiWARP [TMS11] and SoftRoCE\(^2\) are software implementations of iWARP and RoCE, respectively. They provide lower performance than their hardware counterparts, but do not require a specialized NIC.

The Infiniband protocol implements reliable data transfer, which means it keeps track of each packet transmitted through a packet sequence number (PSN) and acknowledges received packets to the sender using this PSN. Unlike TCP, it does not provide any congestion control and relies on the control flow of Converged Ethernet. Although RoCE is intended for Converged Ethernet, it can also be used over a standard Ethernet network, which is how it is used in this work. Further, in this work we use RoCE v2 over UDP/IPv4 and set the MTU (Maximum Transmission Unit) to 1500 B.

### 5.1.3 DMA on FPGA

There is a wide range of existing PCIe DMA engines originating from academia or available as commercial products [Xile, Log]. All the academic solutions are open source and provide a range of different features. EPEE [GWC+14] and DyRACT [VF14] are limited to PCIe Gen2 while ffLink [dlCKK16], RIFFA [JRHK15], and jetstream [VKVF16] support PCIe Gen3 with up to eight lanes.

All these DMA engines are host-driven, meaning the host initiates and synchronizes data transfers. The FPGA is not able to read/write host memory without involving either the application or the kernel driver. Moreover, the transfer buffers have very limited capacity, e.g., 4 MB in jetstream and less than 64 KB in RIFFA. Either they are pre-allocated in the kernel driver or allocated on-demand (e.g., in the case of zero-copy DMA). In both cases, the amount of free and physically contiguous memory that can be requested from the OS is limited, leading to limited transfer buffer capacities. The benefit of this approach is that the logic on the FPGA has to handle only a handful of buffers, thereby avoiding high complexity.

In the case of RDMA, we have two requirements which are not compatible with any of these solutions: (1) the FPGA needs to be able to read and write host memory without any involvement from the CPU (i.e., it cannot involve the host application or an OS driver) and (2) the FPGA should be able to address large amounts of memory, ideally

\(^2\)https://github.com/SoftRoCE

80
tens of gigabytes, instead of a few megabytes, or even kilobytes. Given that there is a fixed overhead per data transfer, e.g. issuing the request, memory latency, etc., being able to address and transfer large amounts of data results in more efficient transfers.

Section 5.2 describes how our solution addresses both issues by providing direct memory access from the FPGA, similar to systems like Intel’s Xeon+FPGA or IBM’s Power 8 with CAPI, although without cache-coherency, but coherent access to the main memory on our x86 setup.

## 5.2 Architecture

Our prototype platform consists of a PCIe-attached FPGA acting as an RDMA capable NIC. The NIC consists of two main components: the RoCE network stack and the DMA engine accessing the host memory over PCIe, see Figure 5.2. The RoCE network stack can process and generate packets at a rate of 10 Gbit/s and is directly connected to the 10 G network interface on the FPGA board. Apart from the RDMA protocol, the stack implements IPv4 and UDP. While we deploy Wisent on a traditional Ethernet network, the used Ethernet IP core\(^3\) supports Priority-based Flow Control (PFC) necessary for Converged Ethernet. For seamless integration into the network infrastructure, we use two modules introduced in Chapter 4 to handle the Address Resolution Protocol (ARP) and parts of the Internet Control Message Protocol (ICMP) to support ping messages.

![Figure 5.2: Hardware modules deployed on the FPGA, connected over PCIe to the host machine and over 10 G Ethernet to the network, the figure indicates commands issued (dashed lines) and the data-flow.](http://www.xilinx.com/products/intellectual-property/ef-di-25gemac.html)

\(^3\)http://www.xilinx.com/products/intellectual-property/ef-di-25gemac.html
Chapter 5. FPGA-based RDMA Network Card

To make Wisent a viable RDMA network card, it must be able to address tens of gigabytes of host memory. This is achieved through three components: (1) huge memory pages in combination with our custom driver, (2) a TLB on the FPGA to translate virtual to physical memory addresses, and (3) a DMA engine to directly access the host memory.

5.2.1 DMA Engine and Driver

For Direct Memory Access (DMA) over PCIe, we deploy the Xilinx DMA/Bridge Subsystem for PCI Express\textsuperscript{4} IP core, running at 8 GT/s and clocked at 250 MHz. As mentioned in Section 5.1.3, all existing DMA engines deploy a host-driven design where the CPU initiates all data transfers. This is also the case for the driver\textsuperscript{5} provided by Xilinx for the DMA/Bridge Subsystem IP core. It exposes multiple streaming interfaces to the application, each implemented with a fixed-size circular buffer in the kernel. Their approach requires for every transfer an interaction with the kernel, leading to a context switch and a data copy from the user space to the kernel space. This increases latency manifold and limits throughput for small transfers. To achieve the functionality required for Wisent, we have implemented our own Linux kernel driver that runs on the latest Linux versions (4.4 - 4.15).

For control and access to the status registers, the DMA IP core is configured with an AXI4-Lite interface that is addressable through a PCIe bar from the host. Our driver exposes this PCIe bar as a device /dev/roce. This device can be directly mapped into the user space of the application through \texttt{mmap}. This allows the software application to directly interact with the FPGA at low latency without involving the operating system. On the hardware side, the AXI4-Lite is connected to a Controller module that converts the register accesses into commands that are issued to the RoCE stack, to custom logic, or to populate the TLB (Figure 5.2). Additionally, through this interface, the host can also retrieve status and performance metrics.

For data transfers, the DMA IP core is configured with two 32 B streaming interfaces: one stream for writing data to host memory and the other to retrieve data from host memory. Further, we enable the Descriptor Bypass interface on the DMA IP core. This interface allows the FPGA to issue requests directly to the DMA engine without synchronization.

\textsuperscript{4}http://www.xilinx.com/support/documentation/ip_documentation/xdma/v3_1/pg195-pcie-dma.pdf
\textsuperscript{5}http://www.xilinx.com/support/answers/65444.html
with the CPU. Each descriptor describes a data transfer from the host memory to the
card or vice versa. To enable direct access to the host memory from Wisent, memory
has to be pinned in advance. To do so, the application passes a memory region to the
driver, which pins every page and also returns its physical addresses. The current version
of the driver does not support interrupts as such applications use polling for low-latency
communication.

5.2.2 Memory Management

RDMA operations specify virtual memory addresses, but to access host memory over PCIe,
physical addresses are used. To translate from virtual to physical addresses, a Translation
Lookaside Buffer (TLB) is deployed on the NIC. Each entry in the TLB stores one 48 bit
physical address corresponding to a 2 MB huge page, which is contiguous in the physical
address space. In our setup, the TLB is deployed with 16,384 entries. Using 2 MB huge
pages, the FPGA can directly address up to 32 GB of host memory. This seems to be
enough for most uses cases, but can be further increased by allocating more BRAMs to
the TLB if necessary.

The TLB module is populated once and does no support page misses, which requires
that the physical pages are pinned by the kernel driver (such a requirement also holds for
RDMA buffers). In detail, the application maps 2 MB huge pages, which are allocated by
the operating system at boot time, into the user space. It then calls the kernel driver to
pin these pages and return their physical addresses. The virtual to physical mapping for
each page is then loaded into the TLB on the FPGA.

Even though all huge pages combined build a single contiguous virtual address space, as
illustrated by Figure 5.3, physically they might not be contiguous. This means the TLB
has to check if a read or write operation is crossing a 2 MB page boundary. If this is the
case, the TLB resolves these accesses by splitting the command into multiple commands,
none of them crossing page boundaries.

5.2.3 Software-Hardware Interaction

The following sequence shows how the operating system, the driver, and a user application
interact. Enabling huge pages aside, we use a default Linux installation.
Figure 5.3: Accesses in virtual memory space leading to memory page crossing in the physical address space, illustrated by the grey data block, which is contiguous in virtual space but non-contiguous in physical space.

1. At boot time, the operating system reserves a predefined number of huge pages (2MB) and makes them accessible as a file, e.g. /mnt/hugepages.

2. After booting the operating system, the kernel driver can be loaded. The driver creates the /dev/roce device that exposes control and status registers.

3. When the application starts, it maps the roce device and the file with the huge pages into its memory space.

4. The application makes an ioctl call to the kernel driver to pin the huge pages to the memory and get the physical address for each of them.

5. After retrieving the physical addresses, the application populates the TLB on the FPGA through the control registers exposed by the /dev/roce device.

6. The contiguous virtual memory space backed by the huge pages is managed by a custom memory allocator, which is part of the application.

7. Finally, the FPGA is initialized by setting configuration parameters such as the IP address.

5.3 RoCE Network Stack

For our prototype NIC, we have implemented a subset of the RoCE v2 protocol supporting the two one-sided IB verbs: RDMA WRITE and RDMA READ. We adopted the scalable architecture introduced in Chapter 3, see Figure 5.4. Protocol processing in the two
5.3. RoCE Network Stack

Figure 5.4: Architecture of the RoCE v2 network stack, with clear separation between data paths and state-keeping data structures. Protocol processing is fully pipelined.

data paths is pipelined to achieve line-rate, and the following protocol headers are processed: IP, UDP, BTH (Base Transport Header), and RETH (RDMA Extended Transport Header) and AETH (ACK Extended Transport Header), respectively.

5.3.1 Receiving Data Path

The different protocols of the network packet are processed at different pipeline stages. At each stage, the current protocol header is parsed to extract all relevant metadata. Then, the packet header is removed and the packet is re-aligned to the width of the data path. After checking the IP checksum and UDP port, the Process IP and Process UDP module extract metadata, e.g., IP addresses, UDP ports, and packet length, and forward it on a separate bus to the Process BTH module. This module extracts the RDMA op-code, the packet sequence number (PSN) and the queue pair number (QPN) from the header. Next, a finite state machine (FSM) checks the extracted metadata against the state stored in the PSN Table and updates it if necessary. For instance, the PSN in the packet is checked against the PSN stored in the table and classified as either an expected, duplicate, or
invalid packet. Based on this classification, the packet is either forwarded or dropped. The final stage processes the RETH and AETH headers and implements an FSM that makes decisions based on the RDMA op-code and, if required, updates the MSN Table, dequeues an address from the Read Request Queue, issues DMA commands, or triggers the generation of a response packet.

5.3.2 Transmitting Data Path

The generation of packets can be triggered in two ways, either as a response to a received packet (e.g., an acknowledgment or a read response) or from a user application which passes a command over the DMA engine to the RoCE stack. The former are handled by the Response Handler module and the latter by the Request Handler module. In the case of a write or a read response, the packet contains payload which is fetched by the RoCE stack by issuing a command to the DMA engine to retrieve it from the host memory. The request is then forwarded to the Generate RETH/AETH module, which generates the corresponding headers and appends payload if applicable. Similar to the receiving data path, before the packet is forwarded to the next stage, it is re-aligned such that the next packet header can be prepended. The Generate RETH/AETH and Generate BTH modules both deploy an FSM to retrieve and update metadata stored in the data structures. They also forward metadata to the Generate UDP and Generate IP modules to generate the corresponding headers. The constant queue pair values – remote queue pair number, remote IP address, and remote UDP port – are initialized by the host during connection establishment and stored in the QP Table.

5.3.3 State-keeping Data Structures

There are three data structures which store metadata for each Queue Pair: the Queue Pair Table, the PSN Table, and the MSN Table. The Queue Pair Table stores the mapping between the local and remote queue pair number as well as the remote IP address and UDP port. The PSN Table stores all packet sequence numbers (PSNs) defining the valid, invalid, and duplicate PSN regions. This information is stored for two cases when the NIC acts as a responder and when it acts as a requester. The MSN Table stores the message sequence number (MSN) and the current DMA address. The DMA address has to be stored for write operations spanning multiple packets, since the target address is only in
the header of the first packet. These three data structures are all implemented as a State Table using BRAMs (Section 3.3). By carefully restricting access to the different fields in the table, we are able to avoid any locking mechanism.

The Read Request Queue keeps tracks of all outstanding read requests and stores for each request the local target memory address. It is implemented as a Multi-Queue (Section 3.3) to support multiple outstanding RDMA read operations per queue pair. Similarly, the Packet Queue keeps track of all transmitted but not yet acknowledged packets. For this data structure, we extended the Multi-Queue with a scan operation to retrieve a range of PSNs in case of a retransmission triggered by a NAK. In case the retransmission is triggered by the Timer, only the longest unacknowledged packet is retransmitted. Its metadata can be retrieved through a front() operation.

The Retransmission Timer implements one timer per queue pair to detect packet loss. If any timer reaches zero, an event is triggered and forwarded to the transmitting data path to retransmit the lost packet.

5.3.4 Scalability

As expected from the scalable architecture, the implementation of the stack allows scalability in two dimensions: First, the state-keeping data structures can be scaled independently to support the desired number of queue pairs. The number of supported queue pairs is a compile-time parameter and has a linear impact on the required on-chip memory usage, see Section 5.4. Second, the width of the data path is parametrizable in power of two steps. In particular, the width can be varied from 8B to 64B resulting in a bandwidth of 10-80 Gbit/s at 156.25 MHz. Similarly, the Multi-Queue data structure can be parametrized by the number of queue pairs, as well as the total number of unacknowledged packets or outstanding RDMA read operations.

5.4 Evaluation

Our evaluation setup consists of two machines equipped with an Intel Core-i5-6600 clocked at 3.3 GHz and 16 GB of main memory. Each machine was equipped with an Alpha Data ADM-PCIE-7V3 FPGA card acting as an FPGA-based network card. The FPGA card has a Xilinx Virtex 7 XC7VX690T and 10 G interfaces of which one was used and connected
to the RoCE stack. The RoCE stack is clocked at 156.25 MHz and the DMA engine at 250 MHz. The board is connected to the host machine over PCIe Gen3 x8. Our baseline is a Mellanox ConnectX-3 configured to operate at 10 G using RoCE v1. For our evaluation, we directly connected two Wisent NICs, respectively Mellanox cards, to each other to remove the potential noise introduced by a switch. Experiments with a 10 G Ethernet switch between the two FPGAs showed that the additional hop adds a latency of 1-2 µs.

Figure 5.5: Throughput of the DMA engine for read and write operations issued by the FPGA through the Descriptor Bypass interface.

5.4.1 Microbenchmark: DMA

As discussed, the bandwidth over PCIe can be significantly affected by the transfer size. To quantify this effect, we run a microbenchmark that evaluates the throughput when issuing requests from the FPGA fabric to the Descriptor Bypass interface. By using this interface, the CPU is not involved in the data transfer at all. For 64 B reads, the throughput is at 2 GB/s, and the peak bandwidth can be reached with transfers of 256 B and larger, see Figure 5.5. For writes, 90% of the peak bandwidth can be reached with a transfer size of 512 B, and for 64 B transfers, the throughput is at 2.6 GB/s. In comparison, the approach by Xilinx requires 16 KB and 8 KB to achieve the peak throughput for read or write.
5.4. Evaluation

operations, respectively\(^6\). In their case, every transfer involves the driver introducing a huge overhead, which is especially visible for small transfers; as an example, 64 B transfers result in a throughput of around 100 MB/s.

5.4.2 Latency

Write latency is measured through a ping-pong benchmark involving two machines. In the ping-pong benchmark, the initiator machine writes data to the remote machine at a predefined address. The remote machine polls on this address until the data is accessible in the main memory. Once this is the case, it immediately writes the data back to the initiator machine that likewise polls on the corresponding memory address. This round trip time is measured on the initiator machine and the corresponding latency (\(\text{RTT}/2\)) is reported. The latency for Wisent increases with the payload size from 4.5\(\mu\)s to 10\(\mu\)s. The commercial card shows a similar increase in latency, however, starts at a lower latency of 3.4\(\mu\)s for the smallest payloads and increases faster than for Wisent.

\(^6\)https://www.xilinx.com/support/answers/68049.html
Chapter 5. FPGA-based RDMA Network Card

To measure the read latency, the application issues a read request and polls the memory address where the requested data is written to by the network card. Figure 5.6 compares the latencies of Wisent with the ConnectX-3. Since this command includes a round trip on the network, the latency is generally higher than in the case of a write operation. The latency again increases with payload size and ranges from 6.8\(\mu\)s to 12.4\(\mu\)s. The latency of the commercial card ranges from 3.2\(\mu\)s to 10.7\(\mu\)s.

The results show that the latency of the programmable NIC is within 2\(\times\) of the commercial card for very small messages. For larger message sizes, the read latency of Wisent is comparable to that of the commercial NIC. We believe that the overall performance of the design is more than acceptable as we are comparing it with that of an ASIC-driven system.

![Latency Breakdown](image)

Figure 5.7: Latency breakdown of RDMA WRITE and RDMA READ operation on 64 B of data. In case of RDMA WRITE, the software issues a request, the FPGA requests data from the DMA engine, the FPGA transmits the data, the remote FPGA receives the data from the network and writes it to the memory through the DMA. In case of RDMA READ, the software issues a request, the FPGA generates a read request packet, the remote FPGA receives the read request, issues a DMA read command and transmits the data over the network. The requesting FPGA receives the data and writes it to the memory through the DMA engine.
5.4.3 Latency Breakdown

To determine how different components contribute to the total latency, we did a breakdown of the latency for the read and write operation with a payload of 64 B. In Figure 5.7, the latency is split up into time spend in software, on the FPGA, on the network, and for DMA transfers. In the case of the write operation, 39% of the latency stems from the FPGA (including DMA), while for read this share is 45%. As mentioned, the FPGAs are directly connected over a 10 G Ethernet link.

![Throughput of RDMA read and write operations](image)

Figure 5.8: Throughput of RDMA read and write operations. The dotted lines indicate the theoretical peak throughput of 10 G RoCE v2 (MTU 1500) and 10 G RoCE v1 (MTU 1024), respectively.

5.4.4 Throughput

The bandwidth of Wisent is evaluated similarly to the latency. For writes, the local machine sends data with increasing payload to a remote machine, and the remote machine replies with a write of 64 B to acknowledge that the data was successfully received. The time from issuing the write command until reception of the 64 B acknowledgment is used to calculate the throughput. For reads, the time from issuing the read command until reception of the requested data is measured to calculate the throughput.
Chapter 5. FPGA-based RDMA Network Card

Figure 5.9: Message rate of RDMA read and write operations. The dotted lines indicate the theoretical peak throughput of 10 G RoCE v2 (MTU 1500) and 10 G RoCE v1 (MTU 1024), respectively.

The throughput for varying payload sizes from 64 B to 1 MB is shown in Figure 5.8. With larger payload sizes, both NICs reach their theoretical peak bandwidth. The lower throughput of the Mellanox card for small payload sizes stems from its lower message rate, as seen in Figure 5.9. The discrepancy of large payload sizes originates from the different headers in RoCE v1 and RoCE v2 and Maximum Transmission Unit (MTU), which cannot be set to 1500 B on the ConnectX-3. Our DMA microbenchmark showed that, even for small transfers, more than 30 Mio. operations/s can be executed. This means that it is not the data transfer that is limiting the throughput for small payloads, but the overhead of issuing a command from the application. The hardware pipelines on the NIC to process and generate packets are able to operate at 10 G line-rate even for small packets.

5.4.5 Resources

The resource utilization by component is listed in Table 5.1. This configuration supports up to 500 queue pairs (QPs). As it can be seen, the resource utilization is around 25%, allowing the deployment of an application or accelerator logic on the same FPGA. The
Table 5.1: Resource usage by component and in percentage of total resources available on the device.

<table>
<thead>
<tr>
<th>Component</th>
<th>CLB</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network if.</td>
<td>1,750</td>
<td>7</td>
</tr>
<tr>
<td>RoCE stack</td>
<td>7,363</td>
<td>36</td>
</tr>
<tr>
<td>DMA</td>
<td>8,013</td>
<td>34</td>
</tr>
<tr>
<td>TLB</td>
<td>477</td>
<td>24</td>
</tr>
<tr>
<td>Control logic</td>
<td>399</td>
<td>3</td>
</tr>
<tr>
<td>Interconnect</td>
<td>7,860</td>
<td>28</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>25,862</strong></td>
<td><strong>130</strong></td>
</tr>
</tbody>
</table>

Table 5.2: Resource scaling with increasing number of supported queue pairs.

<table>
<thead>
<tr>
<th>#QP</th>
<th>500</th>
<th>1 K</th>
<th>2 K</th>
<th>4 K</th>
<th>8 K</th>
<th>16 K</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRAM</td>
<td>36</td>
<td>41</td>
<td>50</td>
<td>73</td>
<td>119</td>
<td>204</td>
</tr>
</tbody>
</table>

TLB with its 16,384 entries requires 24 36 K BRAMs. Similarly, the RoCE stack, which keeps most of the state in BRAMs, uses 36. Given that the state-keeping data structures linearly scale with the number of supported QPs, only the number of BRAMs increases when going from 500 to 16,000 QPs, see Table 5.2. The usage of all other resources varies by less than 1%.

5.5 Summary and Discussion

We have presented Wisent, an open FPGA-based RDMA network card, to facilitate research in the context of RDMA and compute offload to the network card. The card consists of three main components: the RDMA over Converged Ethernet (RoCE) stack, the DMA engine with the TLB, and the Linux kernel driver. The card fully supports the RDMA READ and RDMA WRITE commands with latencies as low as 5 μs, thereby being within range of commercial solutions while remaining fully programmable. Although Wisent was evaluated at a bandwidth of 10 G Ethernet, it can support higher bandwidths thanks to a scalable data path. By making our work open source, we hope that Wisent will enable new FPGA-based applications directly operating on the main memory of remote machines.
Given that Wisent is a prototype platform, there are plenty of opportunities to extend its functionality including support for two-sided commands, removing the requirement to reserve huge memory pages by the OS, and extending the driver with multi-thread/process support.
In this chapter, we introduce Smart Remote Memory (STROM), a mechanism to move data processing to the network card. Unlike most accelerators, by accelerating data processing on the network card, no additional data movement is required. On the contrary, this approach benefits from the data-flow architecture of the network card that allows us to process data at a high rate with minimal overhead in regards to latency. Assuming the acceleration leads to reduced load on the CPU, the overall latency should in fact decrease.

As the name suggests, STROM is an extension to RDMA. One advantage of building STROM on top of RDMA is having direct access from the network card to the host memory to operate on data. Additionally, since the network stack is completely offloaded, the CPU is not involved in data movement and cannot negatively affect the acceleration. Finally, distributed systems that are able to cope with the increasing amount of data rely on low-latency communication to synchronize across nodes, and on high bandwidth interconnects to transfer data between nodes. RDMA is able to deliver on both fronts and, as a result, has been adopted in recent years for many distributed applications [MGL13, DNCH14, KKA14, BCG⁺16, YWX⁺15, TSAC18]. When tailored to the underlying network infrastructure, these systems show large performance gains. However, building applications on top of RDMA is non-trivial. One of the reasons for this is the mismatch between simplistic RDMA verbs and the more complex application logic involved in distributed settings.

STROM provides a mechanism to accelerate distributed applications by offloading com-
compute kernels to the NIC. These kernels can perform data access operations directly from the NIC such as traversal of remote data structures, e.g. lists, indexes, graphs, in addition to data processing operations such as filtering or aggregation of data streams from/to the remote memory. Since it is built on top of RDMA, kernels have direct access to the exposed memory regions and can be invoked remotely over the network. By accelerating data access operations as well as data processing, STROM addresses both the low-latency and high-bandwidth requirements of distributed systems.

A GET operation in a distributed key-value store is an example of a data access operation (see Section 6.3.2) that currently either requires multiple network round trips when implemented with one-sided RDMA verbs [MGL13, DNCH14] or involves the remote CPU when using two-sided semantics [KKA14]. In STROM, the GET operation can be implemented as a kernel on the remote NIC that can be invoked with a single network round trip to retrieve the requested value without interrupting the remote CPU. Enabling more complex operations also addresses the mismatch between operations at the application level and the RDMA verbs. Further, our experiments show that we can, for instance, retrieve an object and check whether it is in a consistent version with minimal overhead in comparison to a conventional RDMA read operation. In regard to data streams, there exists a wide range of operations, such as filtering, aggregation [WIA14], and partitioning [WHZ15], that could be deployed as a STROM kernel. Since many of these operations involve a lot of data movement but are not that compute-intensive, the data-flow architecture of the NIC is suitable to implement them.

STROM is enabled through the open source RoCE implementation in Wisent (Chapter 5), which we extend with STROM-specific verbs for invoking kernels over RDMA. Further, the programmable fabric in Wisent allows us to deploy application-specific STROM kernels that can support line-rate processing. STROM kernels adhere to a given hardware interface to facilitate portability and exchangeability. Kernels can be implemented using high-level synthesis to ease the development and integration process. We implemented STROM on Wisent and demonstrate its capabilities through three example kernels and their use cases: 1) traversal of remote data structures, 2) consistency verification when accessing remote data blocks, and 3) on-the-fly data shuffling when writing to remote memory.
6.1 Background

6.1.1 Communication Models

There are a number of communication models and mechanisms predated RDMA that allow the sender to execute or trigger operations on the receiver machine. A. Z. Spector [Spe82] introduced a communication model to perform remote operations efficiently on local network-based computer systems. Remote operations are executed by a dedicated communication process with low overhead on the remote machine. Von Eicken et al. [vECGS] introduced Active Messages, an asynchronous communication mechanism providing the ability to overlap communication and computation. Each message contains an address pointing to a user-level handler on the remote machine. This handler is executed upon message arrival to extract the data from the network message. Buzzard et al. [BJM+96] presented Hamlyn, a sender-managed interface architecture. Hamly avoids buffer overruns on the receiver side by having the sender specify the receiver-side memory address where the packet has to be stored. Further, the application has direct access to the network interface hardware, bypassing the operating system. It also separates the data movement from the message arrival notification. Some of these mechanisms were adopted by RDMA; for instance the separation of data movement and notifications is achieved by the completion queues in RDMA. Portals [BBG+17] is a network programming interface providing one-sided data movement operations. In Portals, the target address of the remote operation is not defined by the sender; instead the address is determined at the receiver by extracting fields from the message header and matching them against pre-loaded rules on the remote NIC.

6.1.2 Programmable NICs and Switches

There is a wide range of research [Son13, BGK+13, JYVM15, DCPS16] and products [Net19, FPM+18] around programmable NICs and switches in the context of software-defined networks (SDN). In SDN, decisions are made based on packet headers and operations are applied to the packet or its header, e.g., dropping the packet or rewriting the header. P4 [BDG+14] is a high-level language to program the packet forwarding plane and specify what actions to take for a given packet header. P4 can also operate on the payload. A fundamental difference between STROM and programmable switches is that STROM
builds on RDMA’s ability to access the host memory directly from the local NIC, something not possible from a switch. Thus, the in-network processing functionality provided by STROM is complementary and quite different from that of programmable switches.

Programmable switches can be used beyond the scope of SDN, for instance to push data processing into the network [LHCM19, JLZ+17, LSK+16], or accelerate coordination among nodes [DCPS16, JLZ+18]. In case of data processing, a main drawback of programmable switches is the limited memory that can be used to maintain state. This restricts state-full operations, imposes assumptions on the data set, or requires fall-back mechanisms in case the available memory is exceeded. In addition to this, state might have to be migrated or recovered in case flows are re-routed or the switch fails. Another challenge are reliable network protocols, which keep track of transmitted bytes or packets. This makes data reduction operations, such as aggregation, compression, and filtering at the switch, highly complex or unfeasible. By implementing STROM on the NIC, we avoid these limitations since the processing kernels are not exposed to network packets. Although on-chip memory is also limited, STROM kernels can access the host memory to store partial results or even maintain state.

6.2 STROM - Design Decisions

6.2.1 Extending the capabilities of RDMA

STROM has two goals: 1) to reduce the load on the CPU by moving data processing, which is often memory-intensive but not compute-heavy, to the NIC, and 2) to move processing close to the data by extending the functionality of one-sided operations to reduce the number of round-trips or replace two-sided operations. Frameworks [KKA16, KKA19, SZC+17] that implement RPC functionality on top of RDMA use one-sided or two-sided operations to implement a fast message-passing mechanism to execute RPCs on the remote CPU. In STROM, we want to provide similar functionality but execute the RPCs on the remote NIC so as to not interfere with the CPU.

One approach would be to introduce a new verb for each new functionality, however, this does not scale and would increase the complexity of the Infiniband (IB) protocol manifold. Further, with this approach, each new verb has to address the trade-off between wide-applicability and specialization to maximize potential performance gains. Finally, in
contrast to the current basic but robust IB verbs, the application developer would have to program against a more complex and potentially evolving interface.

With STROM, we want to minimize the changes to RDMA and instead extend it with a generic mechanism to invoke a function (STROM kernel) on the remote NIC. A function might require arguments (hereafter referred to as parameters), which have to be transmitted and passed to it upon invocation. One of the most efficient and least intrusive ways to accelerate data processing is on the data path, which in our case is between the network and the host memory. Therefore, we want that the functions on the remote NIC can operate on "RDMA" streams, i.e. the data streams passing through the NIC when reading data from the host memory and transmitting it, or when data is received and written to the host memory.

6.2.2 Low Power CPU vs Programmable Fabric

Recently, NIC manufacturers have started to integrate powerful co-processors on their smartNICs, such as 64bit ARM processors (e.g., Mellanox BlueField, Broadcom PS225) and FPGAs (e.g. Mellanox Innova-2 Flex). ARM cores provide a familiar development environment to many users, but in current solutions the bandwidth is limited to 25 Gbit/s and the latency is significantly affected when the ARM cores are involved. A simple ping pong microbenchmark on a Broadcom PS225 NIC has shown a latency increase of \(3\,\mu s\) when the ARM cores are on the data path. The reason for this latency increase is that the ARM cores are not actually on the data path; instead, they are implemented as an additional endpoint on the NIC connected over an on-board PCIe switch to the NIC and the host. This means involving the ARM cores adds an additional hop. On the other hand, programmable logic in the form of FPGAs is already widely used for high-bandwidth stream processing, such as packet inspection [QFJ+10], and, as a result, also deployed on some commercial smartNICs as a co-processor. For these commercial solutions, the interface between the FPGA and the network stack and the DMA engine is barely documented and fairly restricted, limiting the offloading of complex functionality. We use Wisent as our prototype platform for STROM. Wisent provides a RoCE v2 stack, supporting one-sided read and write verbs, and a DMA engine. Given that it is FPGA-based, we have full flexibility, which allows us to place the functionality of STROM directly on the data path avoiding unnecessary data movement.

While the implementation evaluated in this thesis is limited to 10 Gbit/s, the latest gen-
eration FPGAs support higher clock frequencies, have more logic resources and more on-chip memory capacity. These improvements should facilitate deploying Wisent and STROM with a wider data path and at a higher clock frequency resulting in an overall higher bandwidth. While we present STROM on an FPGA-based NIC, we reckon that with some modifications it could be ported to a commercial smartNIC with an FPGA co-processor.

6.2.3 Programmability

Apart from performance, programmability is a key feature of an accelerator. A number of smartNICs deploy ARM cores, which provide high flexibility, but impose a limit on the bandwidth. Another approach would be to introduce a domain-specific programming language, such as P4, which has a fixed set of operations (actions) and can be applied to pre-defined data structures (packets). In STROM, we do not want to impose limitations on the type of functionality that can be implemented. Therefore, we opt to use high-level synthesis [Xila] to directly program the programmable fabric of Wisent. High-level synthesis raises the programming abstraction from a hardware description language to C/C++. To guarantee that the offloaded functionality is portable and interchangeable at run-time (e.g. through partial-reprogramming), we clearly define and restrict the hardware interface between the offloaded logic and the NIC. This approach is similar to OpenCL kernels for FPGAs, which also adhere to a pre-defined hardware interface but give the programmer the freedom to implement any functionality; hence, we call the offloaded accelerator logic a STROM kernel.

6.2.4 Capabilities

STROM kernels can accelerate data processing and data access. For data processing, the kernel operates on a stream of data that is read from remote memory or written to remote memory when payload is attached to the RPC. There are various operations that can be applied to this type of data or tuple streams, some of which are filtering, pattern detection [WTA10], aggregation [WIA14], compression [FKBH15], and encryption [HV04]. For data access, the kernel can implement pointer chasing to extract data from a data structure or even update simple data structures, such as a circular buffer. Data access and processing can also be combined. For instance, can a data stream be composed/de-
composed on-the-fly from/to multiple memory locations on the host. As an example, data could be stored in the host memory in a column data layout where every column is at a different memory location. In this case, the kernel can compose tuples on-the-fly that are then processed.

Further, STROM kernels are not limited to invocation as RPCs on a remote NIC. They can also be triggered by the local host, thereby enabling, for instance, offloading an on-the-fly data reduction operation to the local NIC while simultaneously transmitting the reduced output to a remote node.

6.3 STROM - Implementation

To implement STROM, changes at the protocol, hardware, and software level are required. In particular, we extend the Infiniband Transport protocol with STROM-specific verbs, integrate STROM kernels into Wisent, and expose the functionality of STROM to the software application.

6.3.1 Protocol

Despite having the ability to change any part of the RoCE implementation of Wisent, when implementing STROM we aim to minimize the changes to the IB protocol to avoid an increase in complexity and a negative impact on existing RDMA verbs. To invoke an RPC and send the corresponding parameters to the remote NIC, we introduce the RDMA RPC verb. This verb maps to a single Base Transport Header (BTH) op-code RDMA RPC Params. Packets with this op-code are treated similarly to an RDMA WRITE Only, which means the payload size is at most one Maximum Transmission Unit (MTU). The payload contains the parameters to the STROM kernel. Instead of replacing the Reliable Extended Transport Header (RETH) on top of the BTH, we reuse the address and length field. In case of an RPC, the address field encodes an RPC op-code that is used to match the request against the deployed STROM kernels on the remote NIC. This mechanism resembles the matching used in Portals [BBG+17] and enables multi-kernel deployments on the remote NIC. If the RPC op-code does not match any of the deployed kernels, either a fallback implementation on the remote CPU is triggered (if configured a priori by the remote CPU) or an error code is written back to the requesting node.
Table 6.1: Reliable Extended Transport Header op-codes to support STROM kernels.

<table>
<thead>
<tr>
<th>Code[7-5]</th>
<th>[4-0]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>11000</td>
<td>RDMA RPC Params</td>
</tr>
<tr>
<td>000</td>
<td>11001</td>
<td>RDMA RPC WRITE First</td>
</tr>
<tr>
<td>000</td>
<td>11010</td>
<td>RDMA RPC WRITE Middle</td>
</tr>
<tr>
<td>000</td>
<td>11011</td>
<td>RDMA RPC WRITE Last</td>
</tr>
<tr>
<td>000</td>
<td>11100</td>
<td>RDMA RPC WRITE Only</td>
</tr>
<tr>
<td>000</td>
<td>11101 - 11111</td>
<td>reserved</td>
</tr>
</tbody>
</table>

As we have described, kernels should be able to operate on data streams. When retrieving data from the remote memory, this can be easily achieved using the RDMA RPC verb. Though, it is not possible to operate on incoming data streams, since the payload is written directly to the host memory by the RoCE stack. Therefore, we introduce the RDMA RPC WRITE verb, which has the same semantics as RDMA WRITE, but instead of writing the payload to the host memory, it is passed to the STROM kernel using the address field in the RETH as an RPC op-code. To support this, the RoCE stack is extended with 4 BTH op-codes RDMA RPC WRITE First, RDMA RPC WRITE Middle, RDMA RPC WRITE Last, and RDMA RPC WRITE Only.

Both new IB verbs are semantically similar to RDMA WRITE, which is used to transmit data from the remote NIC back to the requesting node. Using write semantics for the new IB verbs, instead of read semantics, has the advantage that the size of the response does not have to be known in advance. In contrast, an RDMA READ operation specifies the length of the response already in the request, which is required to pre-calculate the number of expected packets and their sequence numbers. In the context of STROM, this constraint would inhibit many operations, e.g. (data reduction), where the response size is determined at run-time and not known a priori.

In summary, we extend the RoCE stack with five new op-codes in the BTH and two new Infiniband verbs, resulting in a code change of less than 50 lines in the RoCE stack implementation of Wisent. The BTH op-codes use the numbers 24-28, which are currently unused, see Table 6.1.
6.3. STROM - Implementation

A critical factor in the integration of STROM with Wisent is the placement of the kernels. We place the kernels on the data path between the RoCE stack and the DMA engine. This placement especially benefits stream-oriented operations that can operate at a high bandwidth while incurring minimal latency. Further, the existing direct data path between the RoCE stack and the DMA engine remains and is only extended with arbitration logic that adds negligible latency.

To simplify the deployment of STROM kernels and make them run-time interchangeable, we strictly define the hardware interface, as illustrated in Figure 6.1. The data paths between the kernel and the RoCE stack as well as the DMA engine are 64 B wide. The RPC parameters are received from the RoCE stack through a 32 B bus, and metadata to issue an RDMA write operation from the kernel occurs through a 20 B bus. Similarly, DMA read and write commands are issued to a 12 B bus.

Figure 6.1: STROM kernel and its hardware interfaces.

6.3.2 STROM Kernel

Listing 6.1: Function interface of STROM kernel.

```c
void strom_kernel( stream<ap_uint<24>> & qpnIn,
                  stream<ap_uint<256>> & paramIn,
                  stream<net_axis<512>> & roceDataIn,
                  stream<memCmd>& dmaCmdOut,
                  stream<net_axis<512>> & dmaDataOut,
                  stream<net_axis<512>> & dmaDataIn,
                  stream<roceMeta>& roceMetaOut,
                  stream<net_axis<512>> & roceDataOut );
```

A critical factor in the integration of STROM with Wisent is the placement of the kernels. We place the kernels on the data path between the RoCE stack and the DMA engine. This placement especially benefits stream-oriented operations that can operate at a high bandwidth while incurring minimal latency. Further, the existing direct data path between the RoCE stack and the DMA engine remains and is only extended with arbitration logic that adds negligible latency.

To simplify the deployment of STROM kernels and make them run-time interchangeable, we strictly define the hardware interface, as illustrated in Figure 6.1. The data paths between the kernel and the RoCE stack as well as the DMA engine are 64 B wide. The RPC parameters are received from the RoCE stack through a 32 B bus, and metadata to issue an RDMA write operation from the kernel occurs through a 20 B bus. Similarly, DMA read and write commands are issued to a 12 B bus.
Chapter 6. Smart Remote Memory

The well-defined interface also translates to the software implementation. Listing 6.1 shows the kernel interface in C++, the stream type in Vivado HLS [Xila] maps to FIFOs in hardware. The interface consists of four metadata and four data streams. The first two metadata streams (qpnIn and paramIn) provide the queue pair number (QPN) and the parameters to the kernel, and optional payload is received through roceDataIn stream. To access the host memory, the kernel can issue DMA commands consisting of a virtual address and length over the dmaCmdOut interface, and data to and from the DMA engine is sent over the dmaDataOut and dmaDataIn stream. To transmit data over the network, the kernel can issue metadata and data over the roceMetaOut and roceDataOut streams. The metadata consists of the QPN, the target virtual address, and the length. While not the focus of this work, STROM kernels can also be invoked by the local host by posting an RPC to the local network card and providing the QPN together with the required parameters.

**Example Kernel**

Existing work showed one-sided [MGL13, DNCH14] and two-sided [KKA14] implementations of the GET operation in a key-value store. We use the same operation to illustrate how a STROM kernel can be implemented with the given hardware interface and high-level synthesis. The GET operation consists of two read operations, one to fetch the hash table entry and another one to retrieve the data value. For simplicity, in this example we assume that there is always exactly one matching key in the hash table entry; therefore, our example omits handling of misses and corresponding mechanisms such as linear probing or chaining. A more sophisticated kernel supporting additional data structures is shown and evaluated in Section 6.4.1. Listing 6.2 shows the main function get of the GET kernel. The stream data structure that maps to FIFOs in hardware is used to stream data in and out of the kernel as well as between functions within the kernel. The kernel consists of four functions: 1) fetch_ht_entry reads the hash table entry from the host memory, 2) parse_ht_entry parses the retrieved hash table entry and requests the value data from the host memory, 3) merge_read_cmds merges the DMA read commands from the previous two functions, and 4) split_read_data distributes the data read from host memory to the requesting function. The HLS DATAFLOW pragma means that each of these four functions will map to a hardware module that operates independently and concurrently to the others. Combined with the FIFOs connecting the four modules, the whole
kernel is mapped to a pipelined data-flow on the FPGA that can operate at a high data rate.

Listing 6.2: Main function of the GET kernel.

```c
void get (...) {
  #pragma HLS DATAFLOW
  static stream<readOp> readSrcFifo;
  static stream<memCmd> htCmdFifo;
  static stream<memCmd> valueCmdFifo;
  static stream<internalMeta> metaFifo;
  static stream<ap_uint<512>> htEntryFifo;

  fetch_ht_entry(qpnIn, paramIn, htCmdFifo, metaFifo);
  parse_ht_entry(metaFifo, htEntryFifo, valueCmdFifo, roceMetaOut);
  merge_read_cmds(htCmdFifo, valueCmdFifo, readSrcFifo, dmaCmdOut);
  split_read_data(readSrcFifo, dmaDataIn, htEntryFifo, roceDataOut);
}
```

The `fetch_ht_entry` function (Listing 6.3) implements the fetching of the hash table entry. It checks the two input streams `qpnIn` and `parameterIn` for new metadata. Once new metadata is available, it is consumed and used to issue a DMA request to fetch the hash table entry. Simultaneously, part of the metadata is stored into an internal FIFO that is later consumed by the `parse_ht_entry` function. The `PIPELINE` pragma instructs the compiler to pipeline this function with an initiation interval (II) of 1 meaning that the resulting hardware module can consume data from its input streams every clock cycle.

Listing 6.3: Function in GET kernel to fetch the hash table entry from the host memory.

```c
void fetch_ht_entry(stream<ap_uint<24>> & qpnIn, stream<getParams>& paramIn, stream<memCmd>& htCmdFifoOut, stream<internalMeta>& metaFifoOut) {
  #pragma HLS PIPELINE II=1
  if (!qpnIn.empty() && !paramIn.empty()) {
    ap_uint<24> qpn = qpnIn.read();
    getParams params = paramIn.read();
    htCmdFifoOut.write(memCmd(params.getAddress(), 64));
    metaFifoOut.write(internalMeta(qpn, params.getKey(),
                                   params.getTargetAddr()));
  }
}
```
The `parse_ht_entry` function (Listing 6.4) reads the metadata and the hash table entry containing three buckets. It then concurrently compares the lookup key in the metadata against the key in each bucket and determines the index of the matching bucket. The `UNROLL` pragma specifies that the loop is unrolled in hardware meaning that all iterations are performed concurrently. As a last step, the matching index is used to extract the value pointer and length to generate a DMA read command and the RoCE metadata for transmission of the data value. This function is also pipelined with an initiation interval of 1 using the `PIPELINE` pragma.

Listing 6.4: Function in GET kernel to match the key against the buckets and fetch the value from host memory.

```c
void parse_ht_entry(stream<internalMeta>& metaFifoIn,
                     stream<ap_uint<512>>& htEntryFifoIn,
                     stream<memCmd>& valueCmdFifoOut,
                     stream<roceMeta>& roceMetaOut) {

#pragma HLS PIPELINE II=1

    if (!metaFifoIn.empty() && !htEntryFifoIn.empty()) {
        internalMeta meta = metaFifoIn.read();
        htEntry entry = (htEntry) htEntryFifoIn.read();

        bool match[3];
        for (int i = 0; i < 3; ++i) {
            //pragma HLS UNROLL
            match[i] = (entry.getKey(i) == meta.lookupKey);
        }
        // Check which key matches
        int matchIdx = (match[1]) ? 1 : ((match[2]) ? 2 : 0);

        // Write DMA command
        valueCmdFifoOut.write(memCmd(entry.getValuePtr(matchIdx),
                                    entry.getValueLen(matchIdx)));

        // Write RoCE Metadata
        roceMetaOut.write(roceMeta(meta.qpn, meta.targetAddress,
                                  entry.getValueLen(matchIdx)));
    }
}
```
6.3.3 Software Integration/API

Listing 6.5: Application interface to issue an RDMA RPC.

```c
void postRpc(rpcOpCode op, QueuePair* pair, const void* parameters, uint32_t size);
void postRpcWrite(rpcOpCode op, QueuePair* pair, const void* originAddr, uint32_t size);
```

On the software side, we expose the RDMA RPC functionality of STROM through two simple function calls (Listing 6.5). The user can issue an RDMA RPC by calling the `postRpc` function that takes as arguments: an RPC op-code, the queue pair, the pointer to the parameters and their size. The RPC op-code specifies the STROM kernel type and is required for matching the RPC request to the kernel on the remote NIC. The size of the parameters cannot exceed one MTU. To attach payload to the RPC, the user can call the `postRpcWrite` function which takes the RPC op-code, the virtual address of the data, and its size as an argument.

6.4 Use Cases for STROM

We now present and evaluate several STROM kernels and their corresponding use cases. The idea is not to present full applications; instead we focus on critical operations in existing applications and show how they can benefit from STROM. For the evaluation, we use the same setup as described in Section 5.4 consisting of two machines each equipped with an Alpha Data ADM-PCIE-7V3 FPGA card, which are directly connected over 10 G Ethernet.

6.4.1 Traversing Remote Data Structures

Traversing data structures is a memory-intensive but low-compute operation that, in the case of random memory accesses, cannot benefit from the CPU’s sophisticated memory hierarchy. To address this issue, Kocberber et al. [KGP+13] introduced \textit{Widx}, an accelerator for hash index lookups. \textit{Widx} consists of two modules, one to calculate the hash value and another to traverse the node list. This accelerator can provide up to 50% speedup at
Chapter 6. Smart Remote Memory

Table 6.2: Parameters of the STROM traversal kernel.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>remoteAddress</td>
<td>The address of the initial element in the remote data structure.</td>
</tr>
<tr>
<td>valueSize</td>
<td>The size of the final value to be read.</td>
</tr>
<tr>
<td>key</td>
<td>The lookup key.</td>
</tr>
<tr>
<td>keyMask</td>
<td>This mask specifies where the key(s) is/are located in the data structure.</td>
</tr>
<tr>
<td>compareOp</td>
<td>Operation applied to compare the key in the command and in the data structure.</td>
</tr>
<tr>
<td></td>
<td>This has to be one of the following: EQUAL, LESS_THAN, GREATER_THAN, NOT_EQUAL.</td>
</tr>
<tr>
<td>valuePtrPosition</td>
<td>The position of the value pointer within the data structure element, which can be absolute or relative to the key that matched.</td>
</tr>
<tr>
<td>isRelativePosition</td>
<td>Indicates if the valuePtrPosition is relative to the key or absolute.</td>
</tr>
<tr>
<td>nextElementPtrPos</td>
<td>The position of the pointer to the next element in the data structure. The next element is read in case none of the keys in the current element matched.</td>
</tr>
<tr>
<td>nextElementPtrValid</td>
<td>This boolean indicates if the data structure element contains a pointer to a next element.</td>
</tr>
</tbody>
</table>

The application level, illustrating not only the potential of offloading this task but also the inefficiencies of CPUs and their memory hierarchy.

In a single node system, data structure traversals are commonly limited by the memory latency. When moving to distributed systems that expose data structures over RDMA, the latency of a single memory lookup significantly increases due to the network round trip. As a result, applications using one-sided operations are optimized to minimize the number of round trips required per operation. For instance, a hash table lookup in Pilaf [MGL13] requires on average 1.6 RDMA reads at 75% occupancy. In FaRM’s key value store [DNCH14], it takes 1.04 RDMA reads at 90% occupancy. To complete the GET operation, an additional round trip is required, leading to at least two round trips for the whole operation. More complex data structures, such as B-trees or graphs, would require even more round trips per operation and are therefore commonly implemented with an RPC over two-sided RDMA [YWX+15, RMKN15].
To reduce the complexity and number of network round-trips when accessing remote data structures, we implement a STROM kernel, called traversal kernel that allows the traversal of remote data structures by the NIC.

**Traversal kernel**

The traversal kernel assumes that a data structure consists of multiple elements. The elements can be organized hierarchically, as in a tree, or flat, as in a list. Depending on the data structure, each element has one or multiple keys and points to another element or the data value.

The kernel always starts its traversal from the root/head element. It then compares the key(s) in the element with the lookup key. In case of a match, it extracts the pointer associated with the key, which either points to a data value or the next element. If the key does not match, the traversal is either terminated, e.g. when a leaf/tail element is reached, or depending on the data structure, the next element in the data structure is fetched.

The internals of the traversal kernel are shown in Figure 6.2. The parameters are received by the Command handler, which issues a read request to retrieve the initial element and
parametrizes the modules in the *Compare & Extract* pipeline. Read requests from the *Command handler* and the *Extract Pointer* module are merged and issued to the DMA engine. The data received from the DMA engine is forwarded to both the *Check Keys* and *Extract Pointer* module in case of an element, or forwarded to the RoCE stack in case of the data value.

Data structure elements are processed by the *Compare & Extract* pipeline. First, the *Compare Keys* module compares the lookup key simultaneously with all keys in the element specified by the *keyMask* parameter. The comparison uses one of the following functions: `==`, `!=`, `<`, `>`. If multiple keys match, the key with the lowest position is considered. The *FSM* uses the position of the matching key to calculate the position of the pointer that has to be followed. Using this position, the *Extract Pointer* module extracts the pointer and issues a read command to retrieve either the next element or the data value.

When the read command for the data value is issued to the DMA engine, the *Merge read commands* module also issues the metadata including queue pair number, target address, and length to the RoCE stack. To hide the latency of the DMA, the kernel is pipelined and supports multiple out-standing requests.

The *traversal kernel* accepts the parameters listed in Table 6.2. Through the parametrization of the *Compare & Extract* pipeline, the kernel can traverse different data structures such as linked lists, hash tables, trees, graphs, or skip lists.

The current implementation makes the following simplifying assumptions: 1) each data structure element cannot exceed 64 B (corresponding to the data bus width), 2) and each key has a fixed size of 8 B, and 3) the fields within the element are 4 B aligned. These are aspects of the design that can be easily changed.

**Example: Hash Table**

In our first example, we look at a hash table. Our implementation mimics the implementation and data layout used in Pilaf [MGL13]. The hash table consists of two memory regions; the first one contains fixed sized hash table entries which point to the corresponding data value and the second one contains all the values. A *GET* operation requires in the best case two RDMA READ operations, one to read the hash table entry and another one to fetch the data value. By using the *traversal kernel*, the *GET* operation can be executed with a single network round-trip. The *traversal kernel* deployed on the remote NIC will fetch the hash table entry specified by the *remoteAddress* parameter. It will then extract
Figure 6.3: Median latency of hash table lookup using RDMA READ vs a traversal kernel RPC, while varying the value size. Error bars indicate the 1st and 99th percentile.

6.4. Use Cases for STROM

the keys and match them against the given key. If successful, the data value of size `value-Size` will be read and transmitted to the client. Otherwise, the remote NIC could either return an error code or fetch the next hash table entry in case the implementation uses chaining for collision resolution.

In Figure 6.3, we compare the latency of retrieving a value by using either two RDMA READ operations or a single traversal kernel RPC. For this evaluation, we assume that the hash table entry always matches the given key, resulting in the best case of two read operations to retrieve the value. Using the traversal kernel, the latency can be reduced by around 5\(\mu s\) per lookup by saving one network round trip.

Example: Linked List

In our second example, we show how the traversal kernel can be used to traverse a linked list. We consider a simple linked list as shown in Figure 6.5 where each element has the memory layout shown in Figure 6.4. The key in each element is unique. To look up a given key, the list is traversed starting from the header until the lookup key matches the key in the list element. Once the matching element is found, the data value pointed to by
Chapter 6. Smart Remote Memory

Figure 6.4: Element of the linked list with 4 B-aligned fields, each of size 8 B.

Figure 6.5: Linked list in remote memory

the value pointer is read. Given the layout of the list element, we set the keyMask to 1, the valuePtrPosition to 4, and the nextElementPtrPosition to 2.

We evaluate the latency of retrieving a value in the linked list by randomly picking a key and then retrieving its corresponding value by traversing the remote linked list, see Figure 6.6. For each experiment, this operation is executed 1 Mio. times. The length of the list is varied for the different experiments. Given that we retrieve a random element in the list, the expected number of elements that has to be accessed to find the key equals half the length of the list.

When using conventional RDMA READ, each element access when traversing the linked list requires an RDMA READ operation involving a network round trip. Since the elements are accessed sequentially, the latency increases linearly with the length of the list. In case of the traversal kernel, the list can be traversed with a single network round trip, and each element access, requiring a read over PCIe, takes around 1.5 µs. Since the network round trip is the main cost, reducing the lookup to a single round trip leads to a sublinear increase of the latency with increasing list length. The variance in the experiment reflects that the number of element accesses depends on the key. Since the cost of accessing an element is higher for RDMA READ, the measured variance is also higher. We further increased the value size from 64 B to 4 KB (Figure 6.7), however, this has a marginal impact on the latency given that it is dominated by the traversal of the linked list.
6.4. Use Cases for STROM

Figure 6.6: Traversing remote linked list using conventional RDMA READ vs a traversal kernel RPC. Whiskers indicate the 1st and 99th percentile. Value size 64 B.

Figure 6.7: Traversing remote linked list using conventional RDMA READ vs a traversal kernel RPC. Whiskers indicate the 1st and 99th percentile. Value size 4 KB.
Example: B-tree

Even more complex data structures such as B-trees can be traversed by the traversal kernel due to its versatile parametrization. Figure 6.8 illustrates the memory layout of an internal B-tree node. The node contains keys and pointers to child nodes. The lookup key is compared with the keys in the node and, based on this comparison, the corresponding child pointer is extracted. Through the mask parameter, multiple keys in the node can be specified. Using the compareOp parameter (Table 6.2), the comparison can be parametrized to check if the lookup key is smaller than the key(s) in the node. The pointer to extract for the child node can be relative to the key that matched. In the case that multiple keys match, the key at the lowest memory address is picked. With these parameters, the STROM kernel on the remote NIC can traverse a B-tree starting from its root down to the leaf node, which points to the data value, without incurring network round trips.

Figure 6.8: Inner node of a B-tree.

Figure 6.9: Data objects in FaRM [DNCH14] having a version number in each cache line and in Pilaf [MGL13] having a CRC64 checksum over the whole object.
6.4.2 Data consistency check

The x86 memory system only provides atomic operations at the granularity of cache lines. However, many data objects exposed through remote memory are larger than a single cache line. Additionally, atomic operations over RDMA incur a high latency and are avoided whenever possible. In fact, optimistic execution with a way to recover in case of inconsistency is often a faster option. When accessing a data object through a one-sided read operation, the retrieved object can be inconsistent if the object was modified at the same time by the remote host. In FaRM [DNCH14], data objects have a version number stored in every cache line of the object (Figure 6.9a). A client issues a one-sided read operation to retrieve the object. It then checks if the version number in the cache lines is consistent. If such is not the case, the object must be read again resulting in an additional round trip. Similarly, Pilaf [MGL13] calculates a checksum for each object and stores it in the object (Figure 6.9b). When a client reads an object over RDMA, it will check if the checksum is correct and must read the object again otherwise.

We can use STROM to implement the data consistency check as a kernel on the remote NIC.
Chapter 6. Smart Remote Memory

Consistency kernel

The *consistency kernel* receives four parameters: the queue pair number, the address of the object, the length of the object, and the origin address specifying an address in the requester’s memory. The basic functionality of the kernel is to read the data object as specified by the parameters, calculate the CRC checksum over the object, and compare it with the CRC stored in the object. If the two CRCs match, the object is transmitted to the requester, otherwise the object has to be read again from the host memory.

Internally, the *consistency kernel* consists of a pipeline of three modules to check the CRC checksum and two modules to handle commands, see Figure 6.10. The *Command handler* module receives the parameters from the RoCE stack and issues a read request to the DMA engine. When the data object is received from the DMA engine, the *Extract CRC* module extracts the CRC stored in the object and forwards it to the *Check CRC* module. The data object is then passed to the *Calculate CRC* module that calculates the CRC64 checksum over the object and stores the object in a FIFO. Finally, the *Check CRC* module compares the calculated and the extracted CRC value. In case they match, it reads the data object from the FIFO and forwards it together with the corresponding metadata to the RoCE stack. Otherwise, it discards the data object and issues a read command to read the object again.

Evaluation

We have evaluated the overhead of offloading the CRC64 check to the remote NIC in comparison to an RDMA read operation without verifying the checksum and with verifying the checksum in software by the requester (Figure 6.11). For this experiment, we assume the optimistic case where the checksum always holds and therefore the object is read exactly once. For small object sizes, the overhead of checking the consistency in either software or hardware is marginal. With increasing object size, the CRC64 calculation in software introduces up to 40% overhead. For the same object size, the *consistency kernel* only introduces an overhead of 1µs (<8%). The overhead of the kernel stems from the fact that the complete object has to be read and checked before it can be transmitted to the requester. Thus, the execution time of the CRC64 check increases linearly with the object size.

In Figure 6.12, we evaluate the impact of inconsistent reads on the overall latency to retrieve an object. The failure rate is the probability that the consistency check fails when
6.4. Use Cases for STROM

Figure 6.11: Median latency of reading a remote value without a consistency check, with a local CRC64 check in software, and with the CRC64 check offloaded to the consistency kernel on the remote NIC. Error bars indicate the 1st and 99th percentile.

an object is read. Note that in this evaluation it does not affect consecutive retries that always succeed. For a failure rate of 1% or less, the average latency is barely affected. With a 10% failure rate, the required retries in case of RDMA READ incur a measurable overhead, while the overhead from the consistency kernel is minimal up to a failure rate of 50% when the re-reads on the remote NIC also impact the overall latency.

6.4.3 Data Shuffling

In database systems, data partitioning is a common approach to accelerating high-level operations such as joins [BTAÔ, BLP11], aggregations, and sorting [PR14]. Data partitioning enables data parallelism in multi-core systems and splits the data into cache-sized pieces to improve cache-locality. In distributed database systems [BMS+17], data is also partitioned across nodes to increase parallelism further.

Liu et al. [LYB17] evaluated six different data shuffling operator designs targeting parallel database systems. Two designs use one-sided RDMA operations over a reliable connection (RC), two use two-sided (Send/Receive) over a reliable connection, and two use two-sided
Figure 6.12: Average latency of reading a remote data object with varying failure rates and sizes. In case of a failure, the following re-read will always succeed.

(Send/Receive) using unreliable datagrams (UD). In their one-sided implementation, the receiver issues read operations to retrieve data from the sender and then shuffles data locally. The sender is not involved in the data transfer. The sender and receiver exchange control data through two circular buffers. They concluded that the design using unreliable datagrams provides the highest bandwidth and has the lowest overhead on the CPU. However, unreliable communication is not a feasible option for a parallel database system and therefore not considered in our work.

A different approach for data shuffling was taken by Barthels et al. [BMS+17] using one-sided write operations. In their implementation, the sender first shuffles the data locally and then writes each data partition to its corresponding remote memory location. The memory locations are determined by a histogram, which has to be calculated in advance. In this approach, the receiver is passive and not involved in the data transfer.

**Shuffling kernel**

We implement a *shuffling kernel* based on the approach by Barthels et al. [BMS+17], but move the partitioning operation to the remote NIC. This reduces data copies and CPU
load on the sender. This kernel is an example of an RPC that operates on an incoming data stream. Figure 6.13 illustrates the execution of the kernel: (1) the sender sends an RDMA RPC message containing the addresses for each partition, (2) the sender issues an RDMA RPC WRITE to send the data to the remote NIC, and (3) the shuffling kernel on the remote NIC partitions the incoming data on-the-fly and writes the partitions to the host memory.

The kernel consists of a pipeline of three modules, see Figure 6.14. In this example, the data received from the RoCE stack is treated as 8 B tuples, which are hashed in the Partition module to determine the partitionID for each tuple. The module implements the radix hash function, which simply takes the $N$ least significant bits of the value as the hash. However, more robust hash functions could be deployed and are suitable for FPGAs, as shown by Kara et al. [KA16]. The tuple and its partitionID are then forwarded to the Write Combiner, which combines for each partition 16 tuples before they are written to the host memory. By combining multiple tuples together, the transfer size over PCIe to the host memory is increased, which reduces the per transfer overhead and leads to an overall better utilization of the link. For 16 tuples, the transfer size is 128 B, for which a bandwidth of 4.5 GB/s can be achieved (Figure 5.5), more than enough to maintain 10 Gbit/s line-rate.

Internally, the Write Combiner consists of two memories. The first one stores the tuples and has 16 slots for each partition. The second one stores for each partition the next slot index. When the last slot of a partition is filled, the whole memory line is flushed and written to the host memory. Whenever a partition is flushed, the partitionID is forwarded to the Map module, which stores the addresses of all partitions in BRAM. When it receives a partitionID, it looks up the address, issues a write commend to the DMA engine, and updates the address stored in BRAM. The Map module is initialized by the Command
Chapter 6. Smart Remote Memory

Figure 6.14: Internals of the shuffling kernel

 handler, which receives a RDMA RPC message containing a histogram indicating the size and memory location of each partition.

The kernel can support up to 1024 partitions. This results in a total BRAM usage of 112 (Table 6.3), since both the Write Combiner and the Map module have to store this many entries.

Evaluation

To evaluate the shuffling kernel, input data consisting of 8 B tuples is partitioned into 1024 partitions and each partition is written to the remote memory. As a baseline we use the implementation by Barthels et al. [BMS+17] that first partitions the data locally and then writes each partition separately to the remote memory. Figure 6.15 also shows the time it
6.4. Use Cases for STROM

Figure 6.15: Average execution time for partitioning and transmitting data consisting of 8 B tuples.

takes to transmit the data without partitioning as a reference. Doing the partitioning on the CPU requires a pass over data and each tuple has to be copied to its partition buffer. Once the size of a partition buffer reaches a threshold it is written to the remote memory. Since the radix hash function is inexpensive, the overhead of partitioning stems from the additional data pass and copy. The shuffling kernel partitions the data on-the-fly upon reception avoiding data copies. The overhead in comparison to the RDMA WRITE is around 1% and can be explained by the buffering of the partitioned tuples into 128 B chunks and the random writes by the DMA engine.

The shuffling kernel can also be invoked on the local network card such that data is partitioned among different queue pairs and correspondingly different remote machines. However, data shuffling before transmission requires more buffering, up to MTU size, to achieve high bandwidth over the network. This limits the number of partitions, further increases the latency introduced by buffering, and requires more on-chip memory per partition.
Table 6.3: Resource usage of STROM kernels.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>CLB</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>traversal</td>
<td>8,157</td>
<td>15.0</td>
</tr>
<tr>
<td></td>
<td>7.5%</td>
<td>1.0%</td>
</tr>
<tr>
<td>consistency</td>
<td>6,305</td>
<td>22.5</td>
</tr>
<tr>
<td></td>
<td>5.8%</td>
<td>1.5%</td>
</tr>
<tr>
<td>shuffling</td>
<td>17,723</td>
<td>135.5</td>
</tr>
<tr>
<td></td>
<td>16.4%</td>
<td>9.2%</td>
</tr>
</tbody>
</table>

### 6.4.4 Resources

Since STROM kernels are implemented on reconfigurable fabric, the developers have a lot of freedom in terms of their functionality. Yet, they are bound by the available resources, since every feature of the kernel is mapped to hardware resources. Table 6.3 lists the resource usage for the three previously described kernels. The *traversal* and *consistency* kernel use less than 10% of the available logic resources. The *consistency kernel* requires BRAMs for the FIFO that buffers the data object when the CRC is calculated. Similarly, the *traversal kernel* contains FIFOs to hold the data structure elements received from the DMA engine before they are processed by the *Compare & Extract* pipeline. The *data shuffling* kernel is more resource-heavy due to the need to buffer tuples for each partition, which requires on-chip memory and logic resources.

### 6.5 Related Work

SmartNICs equipped with network processors that provide some form of programmability for custom packet processing and network function virtualization (NFV) have been available for quite a while. More recently, NIC manufacturers started to integrate more powerful co-processors; for instance, Mellanox and Broadcom offer smartNICs [Mel, Bro] equipped with 64 bit ARM processors. ARM cores as co-processors allow the developer to run almost arbitrary C code, thereby providing a much higher versatility than STROM. However, to guarantee line-rate processing, the co-processor has to adhere to a very strict time budget per network packet. Additionally, there is no standardized way of interacting with the co-processor. The lack of a programming interface is addressed by Hoefler et al. [HDGT+17] who introduce a programming model to offload simple packet processing to the NIC.
Li et al. \cite{Li2017} propose KV-Direct, which makes use of an FPGA-based programmable NIC to extend RDMA with a PUT and GET verb to support key-value store operations natively. The work focuses on the key-value store processors that access the host memory over PCIe. How the Infiniband protocol is extended and how the new verbs are implemented is not described in detail. We have shown that STROM kernels could also implement this type of functionality without the need to introduce specific PUT and GET verbs. In addition, our approach offers higher flexibility and can support a wider range of applications.

6.6 Summary and Discussion

In this chapter, we have introduced, STROM, our approach to move and enable data processing on the network card. STROM deploys data access and data processing kernels on the network card to accelerate distributed systems. The three example use cases have illustrated that even simple operations on the NIC can have a significant impact on operations at the application level and that STROM is able to reduce the mismatch between these operations and RDMA verbs. Given the flexibility of programmable logic, STROM kernels offer the ability to either accelerate or reduce the complexity of a wide range of applications.

Given the increasingly wide deployment of custom and specialized hardware in data centers, we see our design as an efficient way to offload computation to the network card. The architectural design we propose is also not hypothetical. Our design fits well in recent deployments in commercial clouds. In Microsoft’s Catapult, the FPGA sits on the data path in front of the network card acting as a smartNIC \cite{FPM2018}, which can be extended with the features discussed in this chapter.

One of the strengths of STROM is that the kernel implementations are only limited by the available programmable fabric and the kernel interface. However, this freedom also poses some risks since kernels are able to access the exposed memory regions and generate traffic on the network and PCIe bus. In a production environment, it would be necessary to monitor the kernels and detect misbehavior from which the network and the host should be protected.

One way to reduce the potential for faulty STROM kernels would be a higher level abstraction such as instructions or microcode to program them. Naturally, this abstraction
would be more restrictive and limited in its expressiveness; for instance, it might only express data access operations.
Conclusions

7.1 Summary

In this dissertation, we have presented the first TCP/IP stack for reconfigurable hardware that can operate at 10 Gbit/s line-rate independent of the payload size and support up to 10,000 concurrent connections. We also describe to our knowledge the first open source hardware implementation of RDMA. Based on this, we introduced Wisent, an open research platform for RDMA network cards. Finally, we presented STROM, a mechanism that extends RDMA with data processing and data access capabilities to accelerate distributed systems.

Chapter 1 motivates this work: In recent years, network bandwidth has been increasing steadily and data growth has been exponential, while at the same time single-threaded CPU frequencies have been stagnant. By leveraging the data-flow architecture of network devices that operate at a high rate, there is an opportunity to push data processing functionality to the network. In this work, we propose two approaches to enable in-network data processing: 1) by moving data processing into the network and 2) extending the functionality of network protocols with data processing capabilities to offload compute to the network card.

Chapter 3 describes the scalable architecture for network stacks. The main contributions of the architecture are the move to slower but higher capacity memory while maintaining
line-rate processing. Scalability in terms of the number of connections and the width of the data path is achieved through the separation of the state-keeping data structures, the control plane, and the data plane. Further, we introduce common and reusable data structures for stateful and reliable network processing: State Table with support for mutual exclusion, Hash Table, Multi-Queue, and Timers.

Chapter 4 presents a complete TCP/IP stack for reconfigurable hardware with the aim of enabling network-attached FPGA-based data processing accelerators. The implementation is based on the scalable architecture and can support up to 10,000 concurrent connections at 10 Gbit/s line-rate. The stack enabled FPGA-based key-value stores [ISAV16, ISA17] directly attached to the network and IBM’s cloudFPGA [WPAH16, AWH+17], a network-attached accelerator for data analytics.

Chapter 5 introduces Wisent, our open FPGA-based RDMA network card, to facilitate research in the area of adapting RDMA and extending it with new capabilities. Wisent implements the scalable architecture, operates at 10 Gbit/s line-rate, and can execute one-sided RDMA operations with latency competitive to commercial solutions.

Chapter 6 describes Smart Remote Memory (STROM), which extends RDMA with the capabilities of offloading acceleration kernels to the network card. These kernels can accelerate data access operations by reducing the number of required network round trips and data processing by executing stream-based operations on data passing through the RDMA network card. Three use cases from distributed systems illustrate the functionality of the so-called STROM kernels and evaluate their benefit.

### 7.2 Research Outlook

To address the slow down of Moore’s Law, industry and academia are exploring different types of accelerators. As a result of this trend, specialized hardware is becoming ubiquitous, more powerful, and easier to use. In data centers, this has led to a more heterogeneous environment where certain racks or machines are tailored towards specific domains or workloads. At the same time, data center networks are evolving fast, initiated by programmable switches [BDG+14] and software-defined networking (SDN). Nowadays, the complete network stack is being rethought and optimizations across all layers are being considered.
7.2. Research Outlook

There is a number of hardware innovations and trends in the industry that will open up new research directions based on the work presented in this thesis.

Innovation in the Hardware Landscape

We see two major trends that will benefit systems that use FPGAs: 1) higher memory capacity and bandwidth and 2) tighter system integration thanks to emerging interconnects. Data processing operations, such as joins and group-by aggregations in databases or windowed joins and reductions over data streams, can have a significant memory footprint depending on the input data. Thus, especially for large datasets and high cardinality, their performance can be constrained by the available memory capacity and bandwidth.

At the same time, we consider these operations suitable to the FPGA, since they involve a lot of data movement and little compute. However, we find that the CPU performs well for many of these operations despite its inefficiencies regarding data movement. One reason for this is the sophisticated caching hierarchy with large last level caches (LLC) and multiple memory channels resulting in a high memory bandwidth. In terms of numbers, server grade CPUs have 20-30 MB of LLC and around 60 GB/s bandwidth to the main memory. Another reason is that many implementations [MSL+15, BTAÖ, MBK02] are tailored towards the CPU’s cache hierarchy and therefore deliver phenomenal performance results.

In comparison, FPGA boards until recently rarely had more than two DDR channels and the on-chip memory capacity was limited to a few megabytes, e.g., a Xilinx Virtex7 device has 6.6 MB. As a result, despite having the ability to implement specialized caches using the on-chip memory, FPGA implementations have to address these limitations by either making assumptions about the input data (size, cardinality) or by implementing a fallback mechanism in case the available memory capacity is exceeded. Fortunately, both metrics significantly improved in the latest generation of FPGA boards, bringing them on par with the CPU. As an example, the Xilinx Alveo U250 board has on-chip memory in the form of BRAM and Ultra RAM with a combined capacity of 54 MB and four DDR channels resulting in up to 77 GB/s bandwidth. These improvements have a clear impact on memory-intensive operations and close the gap between CPUs and FPGAs. Moreover, both vendors also recently announced FPGA boards with high bandwidth memory (HBM). For instance, the Xilinx Alveo U280 card is specified to provide a bandwidth of 460 GB/s to its 8 GB HBM. HBM is providing a new point in the trade-off between latency, capacity,
and bandwidth. Since many data processing operations can be parallelized well through partitioning, they will benefit from the massive increase in bandwidth provided by HBM as long as the capacity is sufficient.

In terms of system integration, a number of emerging interconnects, such as CCI-X, OpenCAPI, or Gen-Z, are aiming to address the inefficiencies of PCIe, e.g., high latency and high per transfer overhead. The latter we had to address when implementing the *shuffling kernel* for STROM. The new interconnects will provide a tighter integration between the FPGA and the CPU and enable access to host memory at cache line granularity without a significant penalty in performance. Work [MKN+18, KGA17] on hybrid CPU-FPGA architectures like the Intel Xeon+FPGA showed the benefit of tight integration between CPU and FPGA.

### Data Center Aware Network Protocols

Data center networks have different characteristics than the wide-area networks for which protocols such as TCP/IP were originally intended. Recent work [PLL+15, DSC+15] has shown that out-of-order delivery and packet loss is uncommon in data center networks. Similarly, the type of failures differ. Packet loss in a data center is rarely caused by an unreliable link. Instead a link or even a complete switch fails. Congestion control and recovery mechanisms in TCP are not tailored to this type of failure and, in certain cases, might react in a detrimental way. For instance, the fast retransmit algorithm in TCP can detect packet loss very quickly through duplicate acknowledgments and then initiate a retransmission for a fast recovery. However, if a link or switch fails, it can take seconds until an alternative path is established, which is significantly longer than the detection and recovery time of fast retransmit, e.g. within microseconds. This means retransmission occurs before the new path is established putting unnecessary load on the network and causing additional congestion.

Under the premise that hardware offload is a necessity for high bandwidth networks, commodity network cards provide limited abilities to customize the offloaded functionality. In contrast, the combination of reconfigurable hardware and the scalable architecture presented in this work provides the ability to customize existing protocols, e.g. by introducing data center aware congestion and recovery algorithms. Moreover, this allows for the development and deployment of new protocols tailored to the underlying network infrastructure without relying on commodity hardware to offload network processing. In a data center
7.2. Research Outlook

environment, such a protocol could transparently replace existing protocols such as TCP or UDP as long as it is exposed through the same socket interface to the application.

Disaggregation of Resources

In this area, we see two future directions: 1) reduce data movement by enabling near-data processing for disaggregated memory and storage and 2) enable the disaggregation of FPGAs in the data center to increase their utilization.

Disaggregation of Resources in data centers aims to address the issue of imbalanced resource requirements in server machines and has been proposed by Han et al. [HEP+13]. Since applications have different and often dynamically varying resource requirements, it is difficult to build server machines with the right balance of CPU, memory, and storage. Although the disaggregation of storage [KKT+16] and memory [LCM+09] provides more flexibility, it inherently puts a large burden on the network by moving data that previously occurred within a single machine to the data center network. In STROM, we have shown how data movement can be reduced by moving compute to the remote memory. The same techniques can also be applied to remote (flash) storage. Near-data processing as in STROM can alleviate the burden on the network twofold: 1) reducing the overall latency by reducing the number of required round trips and 2) by pushing data reduction operations, such as filtering or aggregation, to the remote node to reduce the amount of data to be transmitted.

With an increasing amount of accelerators in the data center, the question can be raised of how accelerators can be disaggregated. Similar to other resources, the tight coupling of PCIe-attached accelerators to a host leads to inefficient resource usage. IBM's cloudFPGA [WPAH16] proposes one approach to disaggregate FPGAs by attaching them directly to the network. However, in this approach, FPGAs are explicitly implementing services or functions that can be exposed over the network. Given the increase in network bandwidth and the networking capabilities (RDMA, TCP/IP) introduced in this work, we envision an approach where it is completely transparent to the accelerator logic on the FPGA if it is deployed on a local, PCIe-attached FPGA or on a remote, network-attached FPGA. This can be achieved by mapping the interfaces of the accelerator logic dynamically to either remote or local resources. More specifically, there are two types of interfaces that allow the accelerator logic to interact with other resources: streaming and memory-mapped. The streaming interface is used to pipeline accelerators, to process data received
from another device, or to produce data on the fly. The memory-mapped interface is used to access different types of memories, such as on-chip BRAM, off-chip DRAM, and host memory over DMA. Leveraging this common and well-understood abstraction, we can completely de-couple FPGA accelerators from the host. Hence, if the accelerator is accessing a memory-mapped resource, this can be mapped transparently at run-time to any of the aforementioned memories, including remote memory accessible over RDMA. The same applies to stream interfaces; for instance, if two accelerators are pipelined and connected over a data stream, the two accelerators can be deployed on the same FPGA fabric and be directly connected through a hardware FIFO or they can be deployed on different FPGA devices and transparently be pipelined over a TCP/IP connection. For both memory-mapped and streaming interfaces, the accelerator logic no longer needs to be aware of its environment; instead a management layer would provide the mapping of the abstractions to actual resources. This disaggregation would enable a more flexible deployment and allocation of FPGA resources in the data center.
List of Tables

3.1 Space overhead of the Multi-Queue data structure. 30

4.1 Event types defined in the TCP module. 44

4.2 Concurrent access to the most contentious data structures 53

4.3 Resource usage for different configurations of the TCP/IP stack supporting 10,000 connections. 67

4.4 Resource usage by component and in percentage of total resources available on the device. 67

4.5 Resource scaling with increasing number of supported connections. 68

5.1 Resource usage by component and in percentage of total resources available on the device. 93

5.2 Resource scaling with increasing number of supported queue pairs. 93

6.1 Reliable Extended Transport Header op-codes to support STROM kernels. 102

6.2 Parameters of the STROM traversal kernel. 108

6.3 Resource usage of STROM kernels. 122
List of Figures

1.1 Relative growth of single-threaded CPU frequency and network bandwidth since 1982. ................................................................. 2

2.1 Internals of a Field Programmable Gate Array (FPGA) ..................... 12
2.2 FPGA Design Flow ...................................................................... 13
2.3 Comparison of FPGA integration on different platforms. .................. 15
2.4 A simple data-flow architecture in Vivado HLS. ............................... 17

3.1 Generic architecture that supports only a single or limited number of connections. ................................................................. 20
3.2 Scalable architecture that is heavily pipelined and supports thousands of connections. ................................................................. 21
3.3 Data path scalability through separation of control and data plane. ...... 24
3.4 The State Table data structure with locking mechanism to guarantee consistency. ................................................................. 26
3.5 The Multi-Queue data structure maps multiple queues, each with a variable capacity, to fixed-sized on-chip memory. ............................... 28
3.6 Timer, based on true dual port BRAMs allowing concurrent iteration over timer entries and updating of timers. ................................. 30
3.7 Scaling of resource usage when changing the number of connections respectively queue pairs (QP). The circuit is clocked at 156.25 MHz with a data path width of 8 B. ................................. 32
3.8  Scaling of resource usage when changing the data path width. The circuit is
clocked at 156.25 MHz, supports 512 queue pairs, and deployed on a Xilinx
UltraScale+ VU9P device. .................................................. 33

4.1  Block diagram of the implemented TCP/IP stack .......................... 39
4.2  Block diagram of the TCP module ........................................ 41
4.3  TCP buffer and window management .................................... 45
4.4  Block diagram of RX Engine with separation of data path, control plane,
and state-keeping data structures. ....................................... 46
4.5  Block diagram of TX Engine with separation of data path, control plane,
and state-keeping data structures. ....................................... 48
4.6  TCP Application interface. .................................................. 49
4.7  Interaction between Nagle’s algorithm and Delayed Acknowledgment in-
creases the latency of the response significantly. ....................... 58
4.8  Interaction between client and FPGA without delaying acknowledgments
(a), with piggybacking acknowledgments on data packets (b), and with
merging delayed acknowledgments (c). ................................ 59
4.9  TCP module with RX\_DDR\_BYPASS and TCP\_NODELAY optimizations enabled.
The RX Buffer is replaced by an on-chip FIFO. ....................... 61
4.10 Unidirectional throughput of TCP/IP stack measured using iperf. The
dotted line indicates the theoretical peak throughput. ................ 63
4.11 Unidirectional throughput of TCP/IP stack handling a high number of con-
current sessions. The dotted line indicates the theoretical peak throughput. 64
4.12 Data path latencies in the TCP/IP stack. ................................ 65
4.13 Impact of the available memory on the processing rate of different TCP
configurations. The dotted line indicates the theoretical peak throughput. . 66
4.14 Network Service Layer on cloudFPGA. Source: Network-Attached FPGAs
for Data Center Applications [WPAH16] ............................... 69
4.15 Caribou offers replicated intelligent storage for shared-data databases. Replication uses Zookeeper’s Atomic Broadcast and is transparent to the pro-
cessing layer. ................................................................. 70
4.16 Internal architecture of a storage node in Caribou. .......................... 71
4.17 System architecture of memcached accelerator on F1. Source: *Accelerating Memcached on AWS Cloud FPGAs* [CLL+18] .......................... 72

5.1 InfiniBand and RoCE protocol layers. RoCE v1 is indicated in the Ethernet frame by the EtherType 0x8915, RoCE v2 is indicated by the UDP destination port 4791. .............................................. 79

5.2 Hardware modules deployed on the FPGA, connected over PCIe to the host machine and over 10G Ethernet to the network, the figure indicates commands issued (dashed lines) and the data-flow. .................. 81

5.3 Accesses in virtual memory space leading to memory page crossing in the physical address space, illustrated by the grey data block, which is contiguous in virtual space but non-contiguous in physical space. .................. 84

5.4 Architecture of the RoCE v2 network stack, with clear separation between data paths and state-keeping data structures. Protocol processing is fully pipelined. .............................................. 85

5.5 Throughput of the DMA engine for read and write operations issued by the FPGA through the *Descriptor Bypass* interface. .......................... 88

5.6 Median latency of RDMA read and write operations. Error bars indicate the 1st and 99th percentile. .............................................. 89

5.7 Latency breakdown of *RDMA WRITE* and *RDMA READ* operation on 64 B of data. In case of *RDMA WRITE*, the software issues a request, the FPGA requests data from the DMA engine, the FPGA transmits the data, the remote FPGA receives the data from the network and writes it to the memory through the DMA. In case of *RDMA READ*, the software issues a request, the FPGA generates a read request packet, the remote FPGA receives the read request, issues a DMA read command and transmits the data over the network. The requesting FPGA receives the data and writes it to the memory through the DMA engine. .............................................. 90

5.8 Throughput of RDMA read and write operations. The dotted lines indicate the theoretical peak throughput of 10 G RoCE v2 (MTU 1500) and 10 G RoCE v1 (MTU 1024), respectively. .............................................. 91
5.9 Message rate of RDMA read and write operations. The dotted lines indicate the theoretical peak throughput of 10 G RoCE v2 (MTU 1500) and 10 G RoCE v1 (MTU 1024), respectively. ........................................... 92

6.1 STROM kernel and its hardware interfaces. ........................................... 103
6.2 Internals of the traversal kernel ....................................................... 109
6.3 Median latency of hash table lookup using RDMA READ vs a traversal kernel RPC, while varying the value size. Error bars indicate the 1st and 99th percentile. ....................................................... 111
6.4 Element of the linked list with 4 B-aligned fields, each of size 8 B. ....... 112
6.5 Linked list in remote memory ......................................................... 112
6.6 Traversing remote linked list using conventional RDMA READ vs a traversal kernel RPC. Whiskers indicate the 1st and 99th percentile. Value size 64 B. 113
6.7 Traversing remote linked list using conventional RDMA READ vs a traversal kernel RPC. Whiskers indicate the 1st and 99th percentile. Value size 4 KB. 113
6.8 Inner node of a B-tree. ................................................................. 114
6.9 Data objects in FaRM [DNCH14] having a version number in each cache line and in Pilaf [MGL13] having a CRC64 checksum over the whole object. 114
6.10 Internals of the consistency kernel ................................................. 115
6.11 Median latency of reading a remote value without a consistency check, with a local CRC64 check in software, and with the CRC64 check offloaded to the consistency kernel on the remote NIC. Error bars indicate the 1st and 99th percentile. ....................................................... 117
6.12 Average latency of reading a remote data object with varying failure rates and sizes. In case of a failure, the following re-read will always succeed. ... 118
6.13 Data shuffling over the network using the shuffling kernel. ................. 119
6.14 Internals of the shuffling kernel .................................................... 120
6.15 Average execution time for partitioning and transmitting data consisting of 8 B tuples. ................................................................. 121
## List of Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Function interface of STROM kernel</td>
<td>103</td>
</tr>
<tr>
<td>6.2</td>
<td>Main function of the GET kernel</td>
<td>105</td>
</tr>
<tr>
<td>6.3</td>
<td>Function in GET kernel to fetch the hash table entry from the host memory</td>
<td>105</td>
</tr>
<tr>
<td>6.4</td>
<td>Function in GET kernel to match the key against the buckets and fetch the value from host memory.</td>
<td>106</td>
</tr>
<tr>
<td>6.5</td>
<td>Application interface to issue an RDMA RPC</td>
<td>107</td>
</tr>
</tbody>
</table>


Bibliography


[HDGT+17] T. Hoefler, S. Di Girolamo, K. Taranov, R. E. Grant, and R. Brightwell. “sPIN: High-performance Streaming Processing In the Network.” In Pro-


Bibliography


International Conference on Reconfigurable Computing and FPGAs, ReConFig ’11, pp. 286–290. 2011.


the 35th IEEE International Conference on Distributed Computing Systems, ICDCS’15. 2015.


