Facilitating FPGA Prototyping with Hardware OS Primitives

Katherine Lim

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington
2025

Reading Committee:
Thomas Anderson, Chair
Baris Kasikci, Chair
Irene Zhang

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science & Engineering

©Copyright 2025

Katherine Lim

University of Washington

Abstract

Facilitating FPGA Prototyping with Hardware OS Primitives

Katherine Lim

Co-Chairs of the Supervisory Committee:
Thomas Anderson
Paul G. Allen School of Computer Science & Engineering
Baris Kasikci
Paul G. Allen School of Computer Science & Engineering

Both data center operators and the research community have embraced hardware accelerators, because of their potential for significant improvements in performance and energy efficiency. There have now been several large-scale deployments of accelerators in datacenters from companies such as Google, Facebook, and Microsoft. FPGAs have become a compelling acceleration platform, because their reconfigurability allows them to be repurposed as the application mix changes. Both Microsoft and Amazon have deployed FPGAs throughout their datacenter to both rent to consumers as well as accelerate their own services. Microsoft in particular attaches the FPGAs it uses to accelerate its own workloads directly to the network. Directly attaching the FPGA to the network further reduces latency, improves cost-performance, and reduces energy use relative to mediating network communications with CPUs. However, building accelerated applications or services for direct-attached FPGAs is challenging, especially with the complex I/O and multi-accelerator capacity of modern FPGAs.

This thesis argues that direct-attached accelerator systems can be built in a modular manner that preserves the benefits of a direct-attached accelerator while also reducing the engineering burden. We first describe a design and prototype for Apiary, a microkernel operating system for direct-attached FPGA accelerators based on messaging passing over a network on chip (NoC) architecture. The key idea in Apiary is to raise the level of abstraction for accelerated application code, with isolation, threaded execution, and interprocess communication provided by a portable

hardware OS layer in order to ease development difficulties. We propose specific hardware OS primitives to provide these services and abstractions. We then conduct an end-to-end case study of Apiary by prototyping a selection of these primitives to evaluate how well they serve Apiary's design goals. We then describe Beehive, a hardware network stack we designed and prototyped for Apiary based around message passing over a NoC. We show that our architecture is better able to support the complexity of a software datacenter network stack by providing replication of elements and applications and standard TCP and UDP interoperation. At the same time, direct-attached accelerators using Beehive can achieve 4x improvement in end-to-end RPC tail latency for Linux UDP clients versus a CPU-attached accelerator.

TABLE OF CONTENTS

Page

List of Figures . . . iii

List of Tables . . . iv

Glossary . . . v

Chapter 1: Introduction . . . 1
- 1.1 Published Works . . . 4
- 1.2 Outline of thesis . . . 4

Chapter 2: Background . . . 6
- 2.1 Direct-Attached Accelerators . . . 6
- 2.2 Modern FPGA Trends and Difficulties . . . 9
- 2.3 Network-on-Chip . . . 10

Chapter 3: Apiary: An OS for the Modern FPGA . . . 12
- 3.1 Use-case Example . . . 12
- 3.2 Design Goals . . . 14
- 3.3 Apiary Proposal . . . 15
- 3.4 Apiary Prototype . . . 21
- 3.5 Apiary Evaluation . . . 25
- 3.6 Conclusion . . . 29

Chapter 4: Beehive: A Flexible Network Stack for Direct-Attached Accelerators . . . 30
- 4.1 Design Goals . . . 33
- 4.2 Design . . . 36
- 4.3 Implementation . . . 43
- 4.4 Integrating with Beehive . . . 49

4.5 Evaluation 51
4.6 Conclusion 62

Chapter 5: Related Work 64
5.1 Software Frameworks 64
5.2 FPGA Operating Systems Infrastructure 64
5.3 FPGA Networking Frameworks 66

Chapter 6: Conclusion 69
6.1 Future work 69

Bibliography 71

LIST OF FIGURES

Figure Number | Page

1.1 Representations of different acceleration communication architectures 2

2.1 Relationship between the terms used for data transfer units in NoCs 10

3.1 An overview of Apiary's architecture 15
3.2 A diagram of the base layout of the Apiary prototype used for evaluation 26
3.3 Graph of throughput vs time for two connections, one of which faults partway through due to a bad memory access. 28

4.1 Representations of **(a)** a standard CPU server node; **(b)** a direct-attached accelerator using the Beehive network stack; **(c)** an accelerator using a CPU network stack. 31
4.2 A high-level diagram of the type of network stack Beehive supports 33
4.3 Architecture of a Beehive tile. 36
4.4 The flow through which a packet is processed or constructed in Beehive. 37
4.5 An example of how Beehive tile assignments affect deadlock. 40
4.6 Beehive tile layout for Viewstamped Replication. 49
4.7 Packet size vs. goodput for a UDP echo application. Beehive and CALM perform almost identically across all packet sizes and outperform Demikernel. 53
4.8 Diagram of the UDP stack architectures used for the echo microbenchmark 54
4.9 Packet size versus goodput for Beehive and Linux TCP send 56
4.10 Experimental setups for the evaluation of our Viewstamped Replication system . 58
4.11 Latency versus throughput for the Viewstamped Replication key-value store workload varying the number of shards and client threads. 59
4.12 Packet size versus goodput for a UDP echo application, running on Beehive with multiple network stacks instantiated. 63

LIST OF TABLES

Table Number Page

2.1 Comparison of median and p99 round-trip times of a UDP echo across different configurations . . . 8
2.2 Logic cell counts for several FPGA parts in the previous and most recent Virtex generations. . . . 10
3.1 Proposed state for an accelerator context . . . 17
3.2 Application message interface for the Apiary prototype's memory system . . . 21
3.3 Total throughput and average request latency versus the number of RS encoder tiles . . . 27
4.1 Beehive and prior work versus the goals in Section 4.1.1. . . . 34
4.2 Energy consumption and goodput for Reed-Solomon encoding using Beehive versus CPU as a function of the number of application instances. . . . 57
4.3 Energy per operation (measured at the witness) and performance metrics (measured at the clients) at the circled points in Figure 4.11. . . . 60
4.4 FPGA resource utilization of selected modules in Beehive and Limago. . . . 61
4.5 Lines of code per new tile instantiation in Beehive for end-to-end applications. . . . 62

GLOSSARY

- **Accelerator**: A hardware device specialized for a specific computation.
- **ASIC**: Application-specific integrated circuit, a type of accelerator.
- **Direct-attached accelerator**: An accelerator that is connected to a network such that it can communicate over the network without any CPU intervention.
- **DPDK**: Data Plane Development Kit. A framework of libraries and NIC drivers for building high performance packet processing systems.
- **FPGA**: Field-programmable gate array, a reconfigurable hardware device that can be programmed with various hardware designs.
- **IPinIP**: An IP tunnelling protocol that encapsulates IP packets in IP packets and can be used for network virtualization.
- **LoC**: Lines of code. Often used as a measure of code complexity.
- **OS**: Operating system. A low-level abstraction and management layer for hardware.
- **NIC**: Network interface controller. A hardware device that provides physical layer networking capabilities.
- **NoC**: Network-on-chip. A type of hardware interconnect to allow different chip components to communicate.

- **p99**: A performance metric in computing that represents the 99th percentile of latency meansurements.

- **PCIe (PCI Express)**: Peripheral Component Interconnect Express, the host-device peripheral interconnect utilized by today's NICs and SmartNICs.

- **RPC**: Remote Procedure Call. A communication protocol that allows a program to execute a function or procedure on another computer or process as if it were a local call.

- **RX**: Receive. The path for incoming packets.

- **SoC**: System on Chip. An integrated circuit that places all components onto the same chip, such as CPU, memory, and graphics processing.

- **TCP**: Transmission Control Protocol, a common reliable transport network communication protocol.

- **TX**: Transmit. The path for outgoing packets.

- **UDP**: User Datagram Protocol, a common unreliable transport layer network communication protocol

- **VXLAN**: Virtual eXtensible LAN. An encapsulation protocol that encapsulates Ethernet packets in UDP packets that is used for network virtualization

- **XML**: eXtensible Markup Language. A language to store and transport data.

ACKNOWLEDGMENTS

This thesis would not exist without a number of people who have supported me in grad school and beyond. I feel incredibly privileged and lucky to have been surrounded by people who have provided me with so much support both in academia and in my personal life.

Tom: thank you for taking me in as your student 7 years ago. Thank you for letting me pursue the wild hardware research I wanted to even though it was outside your normal expertise. I have learned so much watching you tease apart and reframe my research ideas in a cohesive narrative even when I did not have a cohesive idea of what I was trying to say. Thank you for your flexibility in letting me do what I have thought best for my research.

Irene: thank you for your unwavering, unconditional support in every avenue, grad school and otherwise. Thank you for, at times, bending over backwards to make sure I have what I need to do my research, whether that's advice, a safe workspace, or simply food. I aspire to be anywhere close to the researcher and advocate you are.

Baris: thank you for adopting me as your student when you came to UW and being so willing to bring your thoughts and expertise to my project. You always find the perfect question to ask about my papers to highlight new aspects of my research.

All my collaborators and anyone who has provided me with advice on research: thank you for your time and energy. In particular, Jacob, thank you for all your help with research hardware infrastructure. Akshitha, thank you for your broader career advice and support.

Henry, Samantha, Jialin. Thank you for all the lab conversations and making it such a fun space to come to work. It was a privilege to share a research space with you.

Dan: Thank you for always being down to get a meal or hang out. I treasure our hardware shitposting discussions. Looking forward to many more zoo trips.

Kevin: Thank you for always being such a calm, positive energy to my decidedly...not. You always bring a levity to the PhD process, which can at times feel so laborious. Our adventures to concerts and beyond were always such a great time to recharge from work.

The lab with good feminine energy, Maddie, Angie, Theano, Elba, Natalie, Irene, and Priyal: what a rollercoaster of the past few years. I am so, so thankful for the community I found in you all. Thank you for all being such caring and giving individuals. It would have been impossible for me to finish my PhD without the sense of belonging I have felt with you all.

Priyal: Thank you for being such a loyal and steadfast friend. Thank you for either keeping me sane or going the exact same type of insane with me over the last couple of years. Our friendship is one of the best things to come out of this PhD process.

Shannon: My thesis would not physically exist without your insight. Thank you for all your advice and guidance in the world outside of grad school. Your success and perseverance is inspiring to watch and encourages me to keep setting ambitious goals even when my body wants to convince me otherwise.

My physical therapists, Jon and Michelle: Thank you for your endless patience and tireless dedication to keeping me upright and able to work even when my body is trying its best to find new and creative ways to prevent me from doing so. This thesis would also have been physically impossible without you.

My parents, Angela and Beng-Hong and my sister, Teddy. Thank you for your love and unconditional support throughout my entire academic journey as I explored different fields before inevitably landing in architecture. Teddy, thank you for the endless supply of Instagram memes poking fun at me doing a PhD.

My cats Vickie and Void *. Thank you for keeping me company during my late night work sessions and bringing me endless entertainment.

My partner, Jon, I owe my research journey to you in so many ways. You were the one who first introduced me to computer architecture research and encouraged me to pursue it all those

years ago. Thank you for always being my biggest, most enthusiastic supporter and having faith in me or my work even when I don't, both in research and beyond. It is one of my greatest privileges to walk beside you on our journeys in research and in life.

Chapter 1

INTRODUCTION

Datacenter operators and the research community have increasingly turned to specialized hardware for higher performance, lower cost, and better energy efficiency than general-purpose processors. Hardware accelerators have been deployed by large datacenter providers for multiple application workloads. For example, Google, Microsoft, and Facebook have all deployed custom accelerators for machine learning [112, 77, 72]. Other examples of deployed accelerators include video processing [172] or database query acceleration [16] as well as accelerators for infrastructure network virtualization [18, 74]. Research has explored accelerating an even wider range of applications including key-value stores [107, 46, 131], consensus [106], and genomics [198, 204].

Two models have emerged for communication with accelerators: host-mediated versus direct-attached. These are represented in Figure 1.1. Figure 1.1**(a)** represents the more common host-mediated attachment method. In this model, a CPU running a software stack mediates all communication between the accelerator and the network. This involves two copies over the PCIe bus: one from the NIC to the CPU and one from the CPU to the accelerator. Figure 1.1**(b)** represents the direct-attached architecture where communication is handled by a hardware network stack such that network traffic can be delivered directly to an accelerator. This decreases data movement, which reduces CPU overhead, lowers latencies, and further reduces energy. Although this model is less common in deployed hardware, there are a couple of notable recent examples. NVIDIA has introduced GPUDirect, which allows an RDMA smartNIC to communicate directly with the GPU [159]. Microsoft has deployed direct-attached FPGAs to accelerate ML inference with significant energy and latency benefits [41].

FPGAs are an especially intriguing acceleration platform for the datacenter due to their reconfigurability. ASICs, although more efficient for a fixed workload, have a higher initial cost

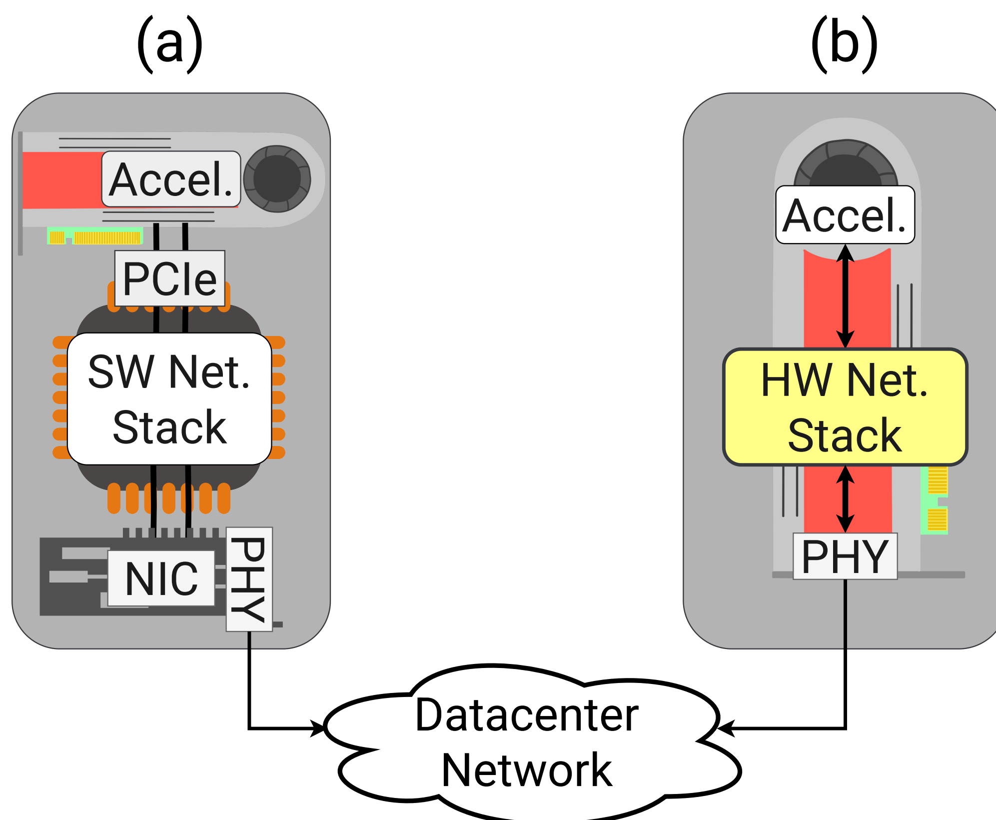


Figure 1.1: Representations of different accelerator communication architectures. (a) represents host-mediated attachment while (b) represents direct attachment.

and cannot evolve with customer needs. As a result, several large cloud providers have deployed FPGAs to rent out to customers directly [14, 10, 97, 152] or to accelerate their own services. Microsoft in addition to Alibaba have used their datacenter FPGAs to deploy machine learning accelerators [77, 20]. Amazon uses its FPGAs to accelerate video encoding [21, 17] and database queries [16].

However, accelerating applications on FPGAs comes with challenges, especially with the complexities introduced by modern FPGAs. Compared to the software development environment on a CPU, the current state of FPGA development infrastructure is like being handed a CPU with the BIOS and bootloader, and little else - roughly where application development stood before the de-

velopment of operating systems. Developers are exposed directly to low-level and non-portable interfaces of various I/O devices such as memory controllers or the Ethernet MACs, and they must wrestle with device-specific details, such as reset procedures and clock domains, to build higher-level services, such as memory allocation and network protocols. All this is needed before one can even start to consider application logic. Modern FPGAs exacerbate these pain points with a wide range of potential I/O devices (e.g. Ethernet vs PCIe vs CXL for accelerator communication) as well as growing logic capacity. The logic capacity of modern FPGAs is sufficient to instantiate multiple accelerators on the same FPGA, but that introduces resource multiplexing, interprocess communication, and fault isolation issues. Software application code can leverage a convenient, portable OS interface that addresses these issues of device independence and portability.

When accelerator operations are filtered through the host CPU, software running on the CPU can provide these types of services. However, this does not exist in the direct-attached hardware setting. In other words, we need a hardware operating system to support portable, modular applications for direct-attached FPGAs, akin to what the OS research community achieved for CPU software in the early 70s.Designing a new operating system requires answering several crucial questions. How should we build a portable OS layer for this setting where there is no host OS and how do we make sure this layer can evolve? We need a virtual machine-like abstraction that is able to isolate and raise the interface level for accelerators. For agile development, we want to be able to take advantage of this abstraction layer for the OS as much as possible.

This thesis argues that direct-attached accelerator systems can be built in a modular way based on message-passing over a network-on-chip (NoC) that preserves the advantages of direct-attached accelerators while also reducing the engineering burden.

We make two contributions: We design and prototype for Apiary, an FPGA OS modeled after a software microkernel to enable accelerators to safely share a single FPGA. Apiary is structured as a collection of hardware OS services and application logic connected over a message-passing layer. As with microkernels, both internal Apiary services and application logic use the same interconnection model — message passing over a switched interconnection fabric, with hardware-enforced capabilities for access control to shared resources such as memory regions. Each module

is wrapped in an Apiary shell that interfaces to the fabric and manages capabilities on the module's behalf similar in style to Barrelfish [34]. For the Apiary prototype, we build out the memory isolation subsystem and integrate it with Beehive to do end-to-end evaluation where we showcase the ability to replicate elements while maintaining memory isolation. We leave for future work prototyping to address aspects such as portability or full context management. Further future work is discussed in Section 6.1.

The second contribution is Beehive, which is Apiary's network stack. Beehive applies the Apiary message-passing architecture specifically to hardware network stacks for direct-attached accelerators in order to facilitate building stacks that meet the agility and complexity needs of a modern datacenter software network stack while still providing the bandwidth, energy efficiency, and latency benefits of network attached accelerators. We show that Beehive can achieve a 4× improvement in end-to-end RPC tail latency for Linux UDP clients while also supporting TCP, stateful network functions, and the ability to replicate elements to scale up processing bandwidth.

1.1 Published Works

- Katie Lim, Matthew Giordano, Irene Zhang, Baris Kasikci, Thomas Anderson. **Apiary: An OS for the Modern FPGA**. 2025. In *The ACM SIGOPS 20th Workshop on Hot Topics in Operating Systems (HotOS '25).*

- Katie Lim, Matthew Giordano, Theano Stavrinos, Irene Zhang, Jacob Nelson, Baris Kasikci, Thomas Anderson. **Beehive: A Flexible Network Stack for Direct-attached Accelerators**. 2024. In *57th IEEE/ACM International Symposium on Microarchitecture (MICRO '24).*

1.2 Outline of thesis

Chapter 2 expands on the motivation by describing the trends of modern FPGA technology, the direct-attached accelerator setting and its latency benefits, and a summary of existing approaches to providing OS functionality to FPGAs. Chapter 3 describes the Apiary architecture and shows how software-like OS primitives might be realized in hardware. Chapter 4 describes our work on

Beehive and details some of the more specific benefits and challenges with building a hardware network stack within the design goals of Apiary. Chapter 5 provides a broader overview of related FPGA OS and network infrastructure work. Finally Chapter 6 describes conclusions and future work.

Chapter 2

BACKGROUND

In this chapter, we describe the challenges and opportunities posed by direct-attached FPGAs in more detail. We first define direct-attached accelerators and illustrate their potential latency benefits over traditional CPU-hosted accelerators. We then overview challenges with realizing the direct-attached accelerators on modern FPGAs. Finally, we briefly introduce the network-on-chip (NoC), a preexisting hardware design technique that both Apiary and Beehive take advantage of.

2.1 Direct-Attached Accelerators

Direct-attached accelerators deliver network traffic straight to the accelerator, bypassing the CPU as shown in Figure 1.1(b). Processing network traffic directly in hardware can have several benefits:

- Latency and tail latency: it cuts down on interconnect crossings between the CPU and various devices on the on the PCIe bus, eliminating associated control and data overheads
- Energy efficiency: specialized hardware, rather than software, is used to process network traffic
- Bandwidth: A streamlined and specialized datapath means higher processing rates.

Broadly, direct-attached accelerators can be further subdivided based on whether they are attached to specialized networks or the general-purpose datacenter network.

For a specialized network architecture, host CPUs on an accelerator server node are connected to the general Ethernet datacenter network while the accelerators are connected to a custom

network that only includes the accelerators. Examples of accelerators attached to a specialized network include machine learning accelerators or data analytics accelerators [114]. In this style of deployment, accelerators are used as a sort of "supercomputer" to scale up computation for a single application. Because the accelerators are homogeneous, the interconnect can be specialized to the workload. However, there are also downsides to maintaining a custom network when accelerators are not used as a tightly-coupled system. Microsoft, where FPGAs functioned independently to respond to requests, moved from a dedicated interconnect to connecting its FPGAs to the general datacenter network, citing maintenance challenges, limited scalability, and complex fault handling [41].

This thesis focuses on direct-attached accelerators where accelerators share the datacenter network with host CPUs. The most notable work on direct-attached accelerators on the general datacenter network is Microsoft's second-generation Catapult FPGAs. Microsoft has used Catapult to accelerate machine learning inference accelerators for Bing queries [41] where researchers found improved tail latency and energy savings versus their software version of the service. Microsoft has also used the Catapult platform to accelerate network virtualization [74]. Other work has investigated the direct-attached setting further to see how it applies to other applications, such as key-value stores [105], consensus algorithms [106], or regex matching [184]. Researchers have also explored the benefits specifically of a hardware versus software network stack in more detail [201, 202], trying to determine the benefit from the network stack specifically. These researchers found significant improvements over the Linux network stack. However, modern state-of-the-art systems aiming for the lowest possible latency typically use an RDMA or DPDK network stack, which can achieve single-digit microsecond latencies [211, 116, 194] at the cost of certain facilities provided by the kernel, such as multiplexing between applications or applying network traffic policies.

2.1.1 Direct-Attached Accelerator Latency Experiment

To answer the question of how much of a benefit direct-attached accelerators have relative to the fastest possible CPU mediation. We use a simple Beehive configuration to compare the perfor-

Client	DPDK Client		Linux Client	
Server	Beehive	DPDK to Accelerator	Beehive	Linux to Accelerator
Median Latency (μs)	4.08	6.22	11.6	17.6
p99 Latency (μs)	4.43	6.79	15.3	61.2

Table 2.1: Comparison of median and p99 round-trip times of a UDP echo across different configurations. Client machines use software networking. Beehive corresponds to the configuration in Figure 1.1(b); Linux and DPDK to Accelator correspond to Figure 1.1(a).

mance of a direct-attached accelerator to one hosted by either a Linux or DPDK network stack running on CPUs. We choose to benchmark both CPU stacks, because Linux is the stack that is generally used throughout datacenter applications today while DPDK network stacks provide what is considered state-of-the-art low-latency networking, but can be difficult to integrate with preexisting applications. Our experiment compares the software-hosted configuration in Figure 1.1(b) to the direct-attached configuration in Figure 1.1(b). We evaluate the performance of UDP echo, where the client sends a UDP packet to a server and waits for the response packet before sending another. We use Linux and F-Stack [194], a DPDK network stack, as the software network stacks. We run 1,000,000 requests and measure the round-trip time (RTT) for each request.

For the direct-attached configuration, we use Beehive implementing a UDP echo server. We try both Linux and F-Stack as the clients. For the software-hosted configuration, we use either the Linux network stack or F-Stack as the software network stack and Ensō [179] as the FPGA accelerator. Ensō is an FPGA-based NIC designed for efficient NIC-CPU communication over PCIe. Internally, we tie Ensō's network output to its input, so it operates as a loopback. For software-hosted configurations, the client and server machines run the same software stack as the software CPU host (e.g. an accelerator hosted by the Linux network stack on the server has a Linux client for benchmarking).

We report median and 99th percentile (p99) round trip times. in Table 2.1. As expected, tram-

polining RPCs through the CPU on the way to the FPGA is both slower and more variable than when the FPGA is directly attached to the network using Beehive. With DPDK, the network stack is at user level on both the client and server. Both latency and latency variance is reduced as the server CPU busy-waits for incoming packets, at the cost of higher CPU overhead. However, Beehive with a CPU client retains an advantage, with 1.5× better median and p99 tail latency relative to best-case redirection through the CPU. When the network stack is provided by Linux, message latency can be affected by CPU scheduling contention, so that Beehive has 4× better p99 tail latency than redirection through the CPU on this benchmark, and 1.5× better median latency.

In summary, this shows that even with a DPDK stack, direct-attached accelerators can still provide a latency improvement, and the relative improvement is larger for tail latency compared to the Linux network stack. With this in mind, direct-attached accelerators are an appealing option.

2.2 Modern FPGA Trends and Difficulties

FPGAs are on a long-term upward trend in size and board complexity. This means they are applicable to more use cases, but this has also resulted in a more complex developer experience. Consider the variety of different I/O devices available on FPGA boards from Intel and AMD. Modern boards include high speed versions of I/O devices present in previous generations, such as PCIe Gen 5, 100 Gbit networking, and HBM memory [19, 102]. For any single type of I/O device, developers are expected to interact through IP cores provided by the vendor, which differ between boards and even between speeds. Modern boards have also added new types of I/O, such as storage [22] or CXL [102, 23].

To compare size, we looked at Xilinx FPGAs parts from the previous 7 series in 2010 and the initial release of the most recent UltraScale+ generation in 2016, as shown in Table 2.2. For each generation, we chose the smallest and largest available parts. Comparing the smallest parts, the number of logic cells has increased by about 150%, while the largest parts have scaled up by about 330% between generations. Using these larger FPGAs is still a developing field, with work exploring multi-accelerator systems [215, 33, 121, 39, 126].

Family	Year Released	Part Number	Logic Cells
Virtex 7	2010	XC7V330T	326,400
	2010	XC7VH870T	876,160
Virtex Ultrascale+	2016	VU3P	862,000
	2016	VU19P	3,780,000

Table 2.2: Logic cell counts for several FPGA parts in the previous and most recent Virtex generations.

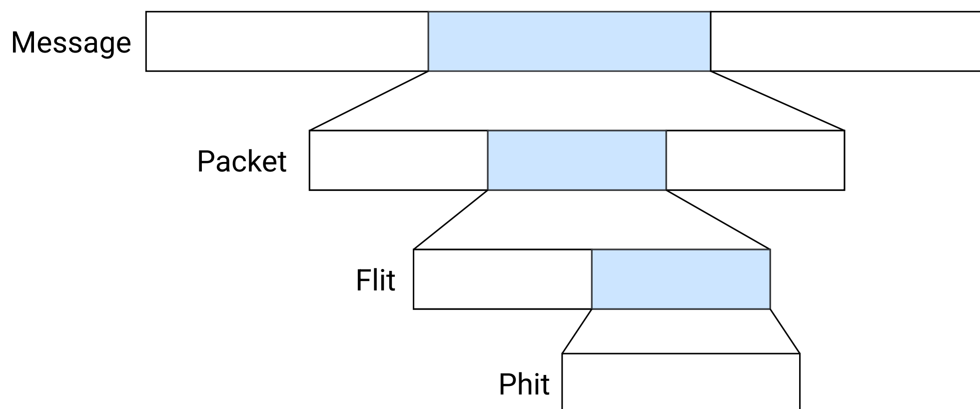


Figure 2.1: Relationship between the terms used for data transfer units in NoCs

2.3 Network-on-Chip

Another important issue to consider as multi-accelerator systems grow in size is the physical architecture and interconnection. Most modern CPUs and complex systems-on-chip (SoC) use a network on chip (NoC), which is a type of on-chip interconnect, to connect different components, replacing buses or crossbars due to the scalability properties of NoCs [109]. NoCs are similar to the interconnection networks between servers where routers and wires in various topologies programatically route packets from one point to another and are used to connect cores to each other, to memory, or to I/O [53]. NoCs differ from more conventional datacenter networks in certain ways though, and research into NoCs is its own subfield. This includes research into issues such as deadlock detection and recovery, different topologies, routing algorithms, flow

control methods, quality of service mechanisms, and security.

Some of the most notable differences are in topology and routing. Common topologies in NoCs include the 2D mesh, 2D torus, or ring [109] as opposed to the fat tree for datacenter networks. Common routing algorithms in NoCs are also typically static and simpler than in datacenter scale networks. The most popular routing algorithm is dimension-ordered routing for meshes and tori [53], which also produces deadlock-free routing.

The conventional NoC terminology for describing units of data transmitted differs slightly from Internet networking, and an illustration of the relationship between the different terms is shown in Figure 2.1. A full unit of data sent from one node to another is called a *message*. This message may be broken up into *packets*, if needed. Each packet is made up of *flits*, which is a shortened version of "flow control digit", which describes the unit of data on which the NoC performs flow control. Finally, each flit may be broken up into *phits* ("physical digits), which correspond to the physical width of the network). For this work, messages do not exceed NoC packet size, so we will always use message to describe on-chip communication to avoid confusion with Internet-level network packets.

Both Apiary and Beehive rely on a NoC to meet their design goals. In both works, the NoC serves as the message passing substrate. This allows them to meet their design goals of modularity and scalability by relying on the well-established modularity and scalability of the NoC interconnect in existing hardware research.

Chapter 3
APIARY: AN OS FOR THE MODERN FPGA

We now move into describing Apiary in this chapter and Beehive in the next chapter. These two works are closely linked as Beehive is Apiary's network stack and as such, has very similar goals and design philosophy. The Apiary chapter focuses on motivating the overall need for an FPGA operating system and surveying and proposing primitives with a proof-of-concept prototype and evaluation. Beehive's chapter focuses on realizing an OS primitive in depth following and explores hardware implementation design choices in more depth with a more detailed evaluation.

To begin this chapter, we provide a use-case example of building up a complex, multi-accelerator SoC to illustrate issues that arise in hardware development, which are typically handled by the OS in a software development environment. We then describe the design of hardware primitives for Apiary that aim to provide the high-level abstractions and multiplexing that are normally provided by an OS. Finally, we describe the subset of primitives we implement in order to evaluate an end-to-end application built on top of Apiary.

3.1 Use-case Example

To illustrate some of the multiplexing and isolation issues that can come up when building complex, multi-accelerator systems, we begin by discussing an example use case. Consider customizing a video encoding service to accelerate part of a video processing pipeline as a direct-attached accelerator on an FPGA board. Requests to the service include a chunk of video, which the service processes and then sends to the next stage of the pipeline. We would like to store frames in the on-board DRAM and need to use the network to receive requests and send response. We would potentially also like to incorporate third-party accelerators.

The first difficulty is integrating the accelerator with the I/O devices, such as the memory and networking. This poses challenges related to programmability and portability and must be overcome to even run the accelerator on the FPGA. Software OSes provide portable abstractions for common I/O devices, but typically there is no equivalent when developing for FPGAs. As a result, developers are exposed to the full complexity of interfaces and operational details and need to build higher-level services which would often be taken for granted in software (e.g. memory allocation, reliable network protocols). Because implementations are often designed for a specific project, reusing previously developed capabilities often requires substantial engineering.

A second set of challenges are in the complexity of the accelerated systems that can be built with larger FPGAs. For example, a single video encoding accelerator could be augmented with additional functionality to handle multiple, independent streams of video. Alternately, we might replicate the encoding accelerator to provide additional encoding throughput. These cases raise questions of how to schedule requests to multiplex the accelerators as well as how to properly allocate memory for multiple video streams. In software, an OS would be to provide multiplexing via a scheduler and would manage memory. However,in hardware there is no subsystem that allows isolation for sharing the DRAM address space. This means that the original encoding accelerator now must be modified to enable this sharing. At this level, certain ad-hoc solutions used in previous systems such as assigning one DRAM channel per accelerator [105] or statically partitioning memory [107] might be sufficient. However, modifying an accelerator may not be feasible, such as if the accelerator was originally developed by a third party. Consider if we were to use the extra FPGA area to instantiate accelerators specialized for other functionality. These accelerators could be composed with existing accelerators on the board. For example, the encoding accelerator could be composed with a compression accelerator to produce a compressed, encoded video stream. Since compression is a common function, we might want to reuse a third-party accelerator. This accelerator would not be designed to participate in a bespoke memory partitioning setup. At this point, the complexity requires a properly designed subsystem for memory isolation. Integrating third-party accelerators also raises questions around the trust model and isolation in case of faulty or even malicious accelerators. In hardware, the current model is that all compo-

nents are mutually trusting which means faults propagate widely throughout the design when they occur, making it difficult to find a root cause and debug.

3.2 Design Goals

To facilitate supporting the kind of complex, multi-accelerator FPGA systems we have just described in Section 3.1, Apiary has the following design goals:

- **High-level interfaces**: Apiary should provide a common set of high-level services for portability and to raise the abstraction level, reducing the engineering cost of developing new functionality.

- **Modularity**: Apiary should support composing applications and services with each other. In particular, it should also be easy to add or remove components when designing the SoC at compile-time without altering existing components and components should not have to be aware of other applicaitons running on the FPGA if they are not directly communicating.

- **Isolation:** Apiary should follow the principle of least privilege and provide isolation to prevent unintended interactions between components.

3.2.1 Trust model

The focus of our design is to implement fault isolation features rather than to enable security guarantees between mutually distrusting applications. This is for a couple reasons. First, there is a class of multi-tenancy attacks on FPGAs that allow co-located applications to attack each other through specialized gadgets in an FPGA design [60]. At the moment, the known mitigations are all circuit-level, design-time modifications. Implementing these mitigations are out of scope, so it would be impossible for Apiary to ensure hard isolation for 3rd party accelerators.

Second, without a high-level abstraction, there is not a practical model or solid context of multitenancy for FPGAs, so we focus on that in this dissertation. As a result, we assume accel-

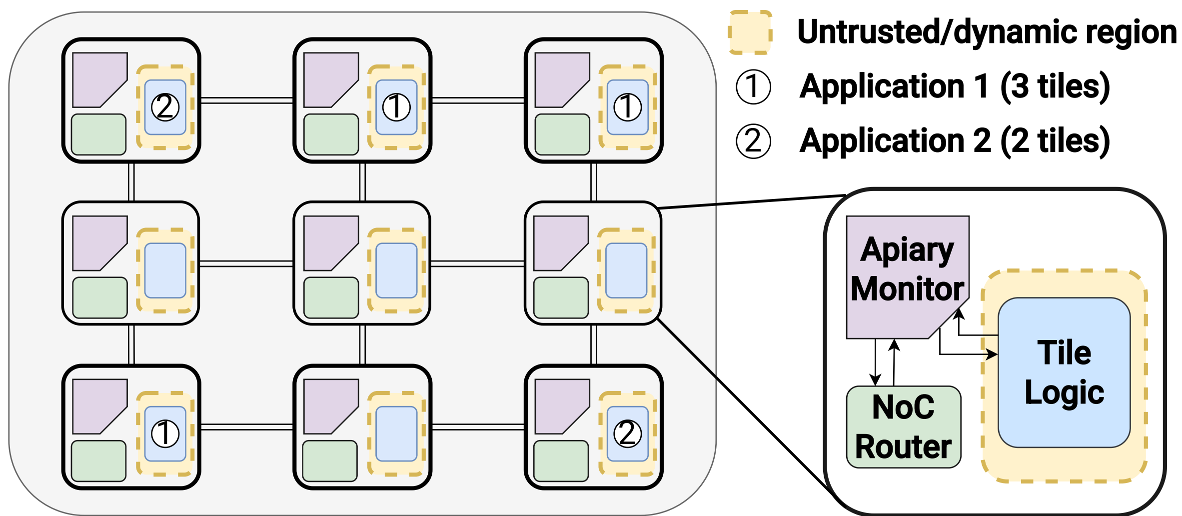


Figure 3.1: An overview of Apiary's architecture. This configuration has two applications composed of multiple accelerators, which may include tiles for Apiary services such as networking. Each tile contains a NoC router for communication, Apiary's monitor to provide isolation and manage capabilities, and an accelerator or Apiary service. The monitor and router are trusted by t he kernel, while the accelerator logic is untrusted.

erator developers are clumsy, but not malicious and that faults occur from mistakes rather than malice.

3.3 Apiary Proposal

3.3.1 Hardware Architecture

Apiary is a Network-on-Chip (NoC) based hardware microkernel. Each tile on the NoC contains an untrusted accelerator, an Apiary monitor, and a NoC router, as shown in Figure 3.1. The accelerator slot can be used either by an OS service such as networking or a user accelerator. In microkernel fashion, most OS services are considered untrusted with respect to the kernel and implemented as user-level processes with the exception of the core memory protection and interprocess communication. Although Apiary does not currently support partial reconfiguration, the intent is that the untrusted tile slots are dynamically instantiated regions, while Apiary's NoC

and monitor reside in the static area, so accelerators can be reprogrammed independent of each other and the rest of Apiary's framework. We omit discussion of scheduling what is configured into the untrusted slot; AmorphOS and Coyote both explore scheduling of partial reconfiguration. We instead focus on inter-accelerator interaction.

Accelerators communicate with each other or services via message passing over the NoC conducted through an IPC service. The Apiary monitor serves as the accelerator's interface to the OS, so all messages go through it. This allows the monitor to implement the trust isolation properties we desire, which we elaborate on in following sections. In Apiary, an application is one or more accelerators that communicate with each other to complete a computation as in the examples in Section 3.1.

3.3.2 Applications and Contexts

One of the primary purposes of an OS is the multiplexing of resources. However, there is not an established way to multiplex an accelerator. One of the key requirements to be able to multiplex a resource is to be able to define its architectural state. For a CPU running software, this state is well-defined by the instruction set architecture (ISA). However, for accelerators, by their very nature of being heterogeneous architectures, there is no standard set of architectural state, which makes it more difficult for a hardware OS to provide completely transparent multiplexing. The architectural state could potentially include every register within the accelerator. While there is work on using compile-time static analysis to identify architectural state and modify the hardware to extract it during runtime to allow for transparent multiplexing [129], it does incur area overhead. It also does not address interactions that could occur between the accelerator and other external modules (e.g. a network stack).

For Apiary, we define the concept of an accelerator context: a unit of computation on one accelerator with associated architectural and OS state. An example of a context might be the state for one network connection on an accelerator. For something like a Viewstamp Replication accelerator, which we discuss later in 4.4.2, it could include the current view and operation numbers as well as some information about the current state of the replicated logs. Given our assumption

for Apiary that accelerators in the system are cooperative, we expect contexts to yield voluntarily rather than relying on preemption, and we expect the accelerator to assist in formatting its architectural state to be stored in the context. For Apiary, we propose a context state to be common to all accelerators in Table 3.1.

Data	Purpose
Context ID	Unique ID for this context
Accelerator ID	What accelerator the context belongs to
Context memory handle	An address for a segment of memory that is maintained across scheduling slices. Used by Apiary to store the process state block when swapped out
Data segment table	Addresses of allocated memory segments available to the process and their capabilities

Table 3.1: Proposed state for an accelerator context

The larger benefit of a user accelerator and the Apiary having a shared understanding of the accelerator context block is that Apiary can help manage the multiplexing and scheduling of an accelerator. For example, we can consider a network-attached accelerator serving requests for several independent network connection and accelerator state. When it yields and provides its state to Apiary, it can then request Apiary provide it the next connection with a waiting request and the corresponding context block with Apiary automatically handling the fetching and storing of state. This is a common pattern across network-attached accelerators, so this also eases the development experience by moving this multiplexing into a common primitive provided by Apiary. This also allows us to offer better isolation since context state is maintained by the Apiary monitor rather than in the accelerator itself where any of the contexts can access each other's state.

3.3.3 Fault Isolation

Next, we describe fault isolation with respect to contexts in Apiary. In hardware systems currently, there are no trust boundaries or fault isolation by default. This means that all accelerators and OS services in current systems implicitly trust one another, and any faults that occur can affect all other processing entities in the system. This is a difficult model to develop under. Different users will not trust each other's applications, and even assuming mutually trusting users, there are no guarantees on how errors or faults will be handled. In comparison, in software, faults are isolated to the process that faulted and any other processes that explicitly established communication with it through IPC. A process that has faulted can also be replaced by restoring the state of a running, healthy process.

For Apiary, our failure model is that a faulting context causes the whole accelerator to be unavailable. This is because we rely on the accelerator to store to or load from its specific internal registers to restore or save its context in order to multiplex them. If the accelerator is faulted, it cannot load a new context even if it is provided by Apiary. To prevent any further interaction with the system, the per-tile monitor will prevent sending or receiving any further messages by sinking them.

3.3.4 Memory Management & Isolation

A memory subsystem is not provided is not provided by default on an FPGA. In an OS, the memory subsystem provides an important high-level interface to help manage allocation as well as isolation. In contrast, most general-purpose CPUs use page-based address space virtualization. Previous virtual memory systems for FPGAs have focused on managing shared virtual memory pages between CPUs and FPGAs [126, 128]. Because CPU memory translation units are hardware, these page sizes have a single or a small, fixed choice of page sizes. Shared memory can be implemented by mapping the same physical pages into multiple address spaces.

However, it is unclear that a fully paged translation system is necessary in Apiary for memory isolation and address translation between accelerators where there is no established isolation and

translation system. For simplicity and flexibility, we propose using segments for virtual addressing as was proposed for CPUs in MULTICS[59] with capabilities for memory isolation.

Segments allow more flexibility in the size of an memory allocation and translation. In a paged system, page sizes are typically fixed to one (e.g. 4 KB) or maybe a few sizes, such as systems that support huge pages on top of standard paging [57]. These granularities are relatively arbitrary and may not match how an application is using the memory, which can lead to costly page faults [196]. This overhead is more pertinent for accelerators where they often have specific memory access patterns and can gain their advantages from specializing to it [93]. With segments, Apiary can avoid the arbitrary sizing imposed by paging and allow the accelerator context to control its optimal memory allocation and translation sizing.

Access to memory is controlled via capabilities. An application can only access a region of memory if it holds the a capability for that region of memory with the right level of permissions. The capability table is maintained by the per-tile monitor. To prevent tampering with capabilities, accelerator contexts manipulate capabilities indirectly via an interface provided by the per-tile monitor. The per-tile monitor is responsible for checking that the requested operation can be performed (e.g. checking the capability has the right permissions) and then manipulating the fields of the capability or sending messages containing the capability. The API we implement for is discussed in more depth in Table 3.2.

On each memory access, the capability table is checked by the per-tile monitor. If the context has the proper capability, the per-tile monitor will send a message requesting the appropriate physical address according to the segment table. Memory sharing is done by sending capabilities via message passing. For example, if application context *A* has a capability for a memory region that it wants to share with context *B*, it should use the capability it has for that memory region to mint a new one and then send it to *B*. We do not attempt to provide failure isolation between applications that voluntarily share memory. That is, if context *A* and context *B* voluntarily establish a shared region of memory, and context *A* misbehaves in some way and corrupts the shared region of memory, context *B* will not be isolated from the corrupted data. The capability system could potentially be used to provide better failure isolation with respect to shared memory. For

example, . We leave the details of this to future work.

While Apiary provides virtual addressing, it does not provide full virtual memory by providing the illusion of an infinitely large address space to each accelerator context. There must be sufficient memory physical available for all accelerator contexts. To prevent an accelerator context from hogging memory, Apiary would rely on policies to set memory usage limits. We leave implementing and setting these policies to future work.

3.3.5 Communication

One of the crucial privileged components in a microkernel is interprocess communication (IPC). The purpose of IPC is to allow OS-managed communication between two processes, and the kernel is responsible for providing channels which should be performant as well as accessed controlled. IPC can enable the composition of different services or programs on a system, which is beneficial for mdoularity. For modularity and isolation, Apiary also needs an access controlled communication primitive.

A form of IPC already exists between accelerators on FPGAs in the form of queues that are used to pipeline accelerators [90, 215, 126]. Because accelerator computation is usually trusted, these queues are not accessed controlled in any way. With untrusted accelerators, rate limiting or access control can help mitigate unintentional behavior that degrades performance.

In Apiary, we use the NoC infrastructure to do message passing and routing already exists since we use a NoC. The Apiary monitor sits between the accelerator and the NoC, so it can inspect all communications between the accelerator and the rest of the system to enforce access control or other policyies. By using a NoC we can take advantage of prior NoC research for implementing our IPC layer, because prior work has addressed topics such as security [200, 199], application message level deadlock [156, 130], quality of service guarantees [163, 87], and other common concerns in software IPC infrastructure.

To enforce access control for IPC, we also use capabilities. An accelerator context must hold a capability for the message destination in order to send a message to that destination.

3.4 Apiary Prototype

We implement a subset of Apiary's proposal. Specifically, we implement a capability-based memory protection subsystem and integrate it with Beehive's TCP engine which is described further in Section 4.3.4 as well as a Reed-Solomon application to create an end-to-end case study. We choose these components because they are foundational to the process context model, memory isolation, and microkernel model for building Apiary. We begin by describing our implementation of the memory protection subsystem. We then describe how the TCP stack and application interact with this new subsystem. Finally, we evaluate qualitatively and quantitatively how well this system meets the design goals we laid out for Apiary in Section 3.2. We implement our prototype in SystemVerilog and evaluate it in cycle-accurate simulation.

`allocate`	Request application data memory region of size `s` from Apiary. On success, the application is sent a message containing the capability handle `i` and allocated size `s`.
`free`	Free the capability with handle `i`.
`send`	Derive a copy of capability `i` with size `size`, starting at `offset` bytes into the capability region and permissions `perm`. Send it to the tile at (`x`, `y`) and pass a user-level message to the remote app context with the remote handle `i'`.

Table 3.2: Application message interface for the Apiary prototype's memory system.

3.4.1 Memory Protection and Allocation

Apiary's memory subsystem consists of a central memory management tile that performs allocations and creates capabilities for the allocations, and per-tile components that enforces the segment bounds and manage further operations on the capabilities. The application-facing API is given in Table 3.2. The `send` operation operates on a remote capability table and also requires an IPC capability due to the user-level message to notify the remote tile of the new capability handle.

On start-up, the central management tile is given the whole of memory as one large segment. We integrate Falafel [80], an open-source hardware memory allocator, to manage the allocation of memory. On request, the management tile will allocate a buffer and then create a capability for it that is sent back to the requesting application. The capability record stores the base physical address of the segment, its size, the permissions for the segment, and the tile coordinates of the addressed memory. The memory management tile is agnostic as to the type of backing memory: the target memory tile just needs to respond to basic load and store operations.

Capabilities are stored in a per-tile table. An application context addresses a location in memory using an address made up of the index, which is the capability capability handle, and an offset in the region. The top n bits of the address are the index while the bottom k bits are the offset. Capabilities are checked by the MMU on outgoing messages by extracting the index and comparing the address to the base and length of the segment associated with that capability. The MMU is also responsible for address translation by extracting the offset and adding it to the base physical address for the segment.

Applications can grant other applications access to a segment by deriving a new capability for that segment and sending it to them. The new capability must be the same or more restrictive than the initial capability. On a memory fault, such as an out of bounds memory access, the MMU prevents the access, sends a message to a management tile described in more detail in Section 3.4.5.

In the current implementation, Apiary makes no fault isolation guarantees between accelerator contexts that have voluntarily shared memory. That is, if two contexts have shared a memory region and one of them crashes and writes bad data to the shared region, Apiary makes no effort to prevent the other from accessing the data. Future work could extend the capability system to implement a stronger memory ownership model using capability permissions to provide better failure isolation.

3.4.2 Communication

We use a NoC as Apiary's messaging layer. We describe the implementation details in more depth in Section 4.3.1. To enforce IPC access control, we use the capability table. We insert dummy capabilities of size 1 that are associated with the destination tile that are checked on each message send. For our prototype, we setup these entries statically ahead of time, but in a fully realized version of Apiary, a runtime service would manage setting up these capabilities.

3.4.3 TCP Integration

We modify Beehive's TCP engine described more in Section 4.3.4 to integrate with Apiary's memory subsystem. Specifically, we place the receive and transmit engines in Apiary untrusted tiles behind the Apiary monitor. We modify the new flow logic such that the receive and transmit buffers the TCP engine shares with the application are allocated as segments associated with capabilities on connection open. The TCP engine then uses the `send` capability operation to grant the application listening for that flow access to these memory buffers for late reads and writes. These capability messages also serve as the notification of an opened connection. The user-level message includes a flow ID that identifies a particular TCP connection, the capability handle on the remote tile, and a buffer type specifier as to whether it is a send or receive buffer.

The TCP user API consists of several messages to put data into and retrieve data from the TCP stack

TCP Send API

1. `send_msg_req`: Sent by the application with the size of the payload it wishes to enqueue and a flow ID to identify the connection it wants to send on

2. `send_msg_rsp`: Sent by the TCP stack to the application in response when there is enough room in the shared buffer to enqueue the requested payload. The response contains the flow ID and the tail pointer which specifies the offset in the buffer at which the application should write the data

3. `bump_send_ptr`: Sent by the application after it is finished writing the data to notify the TCP stack of enqueued data. It contains a flow ID and the updated tail pointer value

TCP Receive API

1. `recv_msg_req`: Sent by the application with the size of the payload it wishes to receive and a flow ID to specify the connection it wants to receive from

2. `recv_msg_rsp`: Sent by the TCP stack to the application when it has received enough data to satisfy its request. The response contains the flowID and the head pointer which specifies the offset in the shared buffer at which the application should read data from

3. `bump_recv_ptr`: Sent by the application after it is finished writing the data to notify the TCP stack of freed space. It contains a flow ID and the updated head pointer value

3.4.4 Application Integration

We modify the Reed-Solomon (RS) encoder described further in Section 4.4.1 to integrate with Apiary and the Apiary version of TCP. We put the encoder in an untrusted tile with an Apiary monitor. We add a skeleton context module to handle capabilities sent from the TCP engine. This module is responsible for receiving the user-level messages from the `send` operations from the TCP engine. The context module stores the information that comes with the `send` message associated with a particular flow ID. This module also tracks active flows and manages the TCP-specific context for the application; when the module provides a flowID of an active flow to the RS encoder, it also provides the TCP buffer handles as well. The RS encoder infrastructure interacts with the TCP stack in a synchronous manner. Specifically, after it has issued a `send_msg_req` or `recv_msg_req`, it waits to receive the response from the TCP stack. As an optimization, it could operate in an asynchronous manner where it requests to be notified about data on several different flows, without waiting for the response. In this case, the context module would be responsible for receiving and buffering these notifications.

We expose the Apiary service user interface via a specific set of SystemVerilog package files. Since Apiary is message-passing based, these package files contain SystemVerilog structs that represent the message types available for interacting with Apiary services. The accelerators import these packages to have the definitions, akin to the Linux ‘uAPI‘ header files.

3.4.5 Fault handling

For our prototype, we implement a simple version of the management tile to catch and manage faults. For our prototype, we raise an error in our testbench if a memory or IPC access fault occurs. When these occur, we handle them in the MMU. The MMU sinks the offending message and then sends an error message to the management tile.

3.5 Apiary Evaluation

In this section, we evaluate our end-to-end case study and how it meets our goals of modularity, high-level interfaces, and isolating memory faults. Some of the goals we assess purely qualitatively while others are evaluated through end-to-end experiments. The base design we use for this experiment is shown in Figure 3.2.

Our experiments are done in cycle-accurate simulation using Questa. We use cocotb [76], a co-routine based Python cosimulation framework, to simulating the client node and the I/O components shown as cosimulation models in Figure 3.2. We use a cycle time of 4ns. We take performance measurements from cocotb and inject requests into Apiary in a closed-loop manner using 12K blocks. The physical Ethernet, memory, and error management tiles are simulation models. Unless specified, the Ethernet and memory cosim models are set to run with no delay, so any bottlenecks are from the design itself.

3.5.1 Modularity

For modularity, we are interested in whether components can easily interact with each other as well as if we can easily add or remove components from the design.

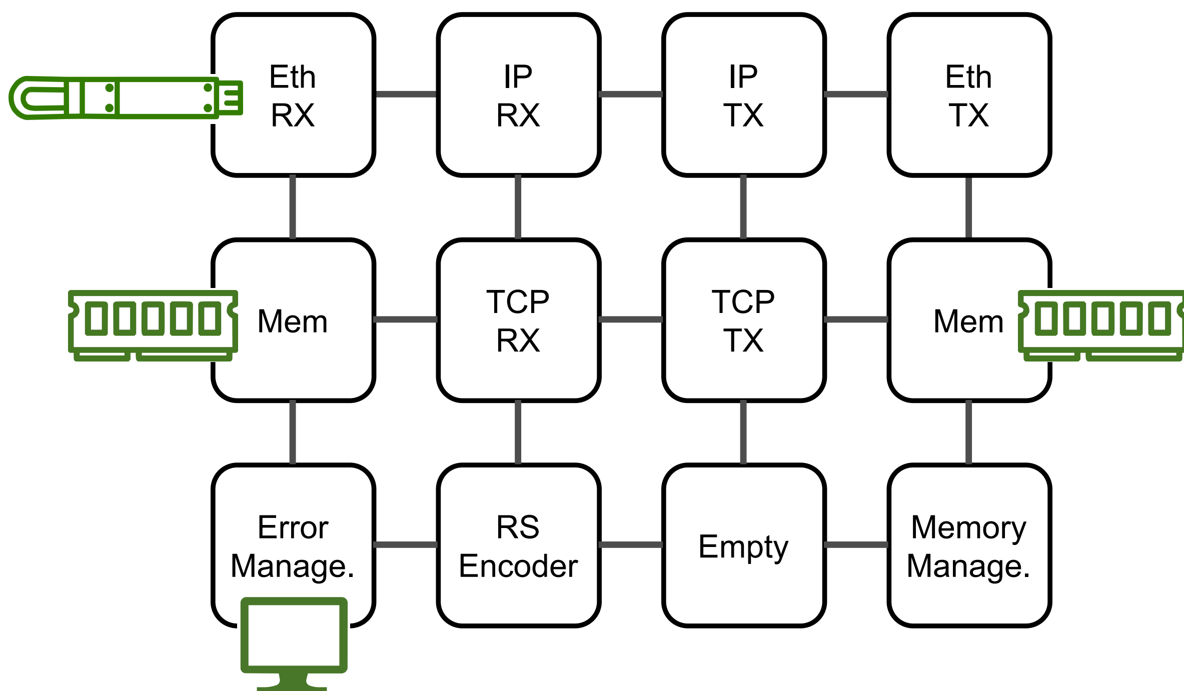


Figure 3.2: The layout of the Apiary prototype used for evaluation with one Reed-Solomon application. The elements in green (Ethernet, memory, and error management) represent components which are implemented as co-simulation models.

From this case-study, we are able to compose an application with a service, the encoder with the TCP stack, as well as services with each other, the TCP stack with the memory management. To evaluate if we can add or remove components to the design, we start with the topology shown in Figure 3.2 and add additional encoder tiles and evaluate the resulting performance. Each tile is given one TCP connection, so there are 4 TCP connections with 4 tiles. We exclude the first request in our latency average, because it includes connection setup. For 1 tile, the first request takes 11μs with TCP connection setup, including buffer allocation taking 1.8 μs.

The results are shown in Table 3.3. Without making any changes to the services themselves, we are able to support additional application tiles. We see the aggregate bandwidth increases by 1.8× when going from 1 tile to 4 tiles while the average latency increases by 1.3×. The bandwidth scaling is not linear, because all the encoders share the same memory tiles, leading to contention.

Tiles	Total Thruput (kreqs/sec)	Avg. Latency (μs)
1	105	9.4
2	147	13.1
3	175	16.5
4	190	18.3

Table 3.3: Total throughput and average request latency versus the number of RS encoder tiles. The total throughput increases as more tiles are added as does the average latency.

3.5.2 High-level Interfaces

To assess our high-level interfaces, we are interested in whether the provided services are helpful to the development experience. The TCP engine uses our memory management service for dynamic buffer allocation on connection start up whereas previously, the TCP buffers were allocated statically.

With the memory system and addressing in place, we are also able to change where buffers are allocated without changing the application logic at all. Specifically, during development and debugging, we use a single memory tile, so the application's TCP connection send and receive buffers are allocated from the same memory tile. However, to improve performance during our experiments, we add a second memory tile, because the memory tile can only handle one operation (read or write) at a time, and both the TCP engine and the RS encode accelerator read from and writes to the memory at the same time. Anecdotally, the additional checking was helpful during development experience to catch bugs earlier rather than after they corrupted or made otherwise unintentional memory writes.

To quantitatively assess the usefulness of the high-level interface, we evaluate the portability of the Apiary interfaces. For this, we change the simulated speed of the memory to mimic the difference in using different types of memory (e.g. SRAM vs DRAM) on the FPGA. We are able to run this experiment making no changes to either the services or the applications; only the simulated memory tile changes.

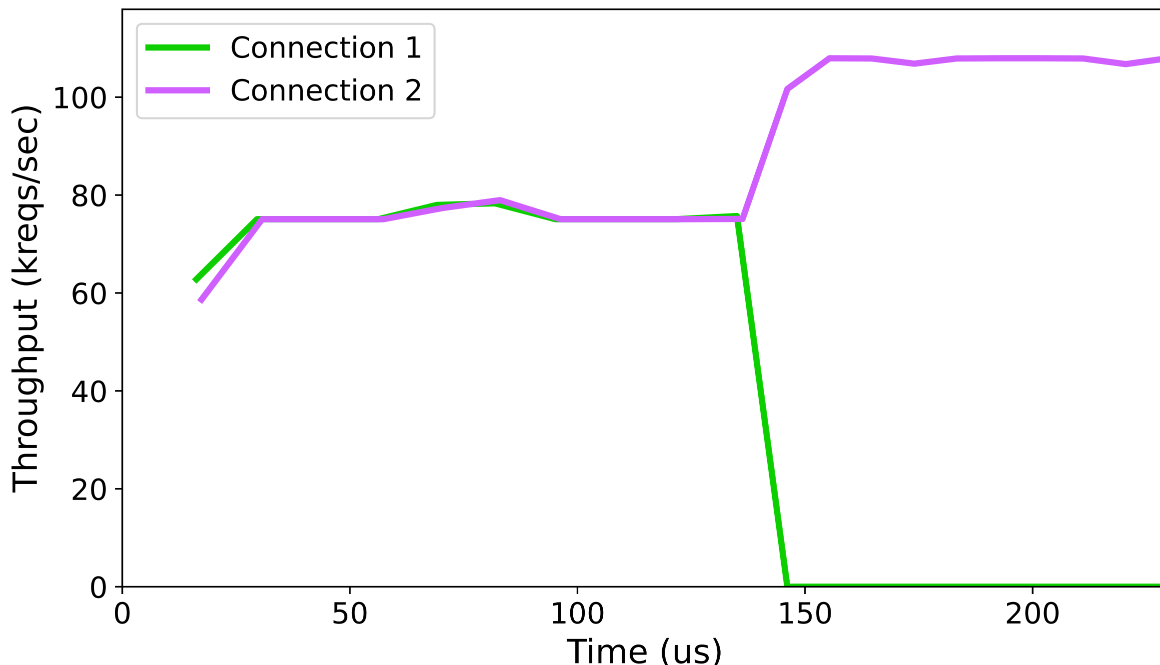


Figure 3.3: A graph of throughput vs time for two connections, one of which faults partway through due to a bad memory access. The other connection is able to continue operating normally.

We add a random delay of between 10 to 20 cycles to every memory operation. This drops the average throughput by 80% to 20.5 Kreq/s and raises the average request latency by 420% to 49 μs from the base Reed-Solomon encoder performance shown in Table 3.3.

3.5.3 Isolating memory faults

A goal of Apiary is to provide isolation between accelerators if one misbehaves. Specifically, we look at what happens when one of the RS encoders makes a bad memory access. As an experiment, we instantiate a design with two tiles and make a connection to each. We then measure the throughput periodically from the clients perspective of the two connections. After a while, we purposefully make a bad memory access on the encoder of connection 1. Figure 3.3 shows the throughput over time for the two connections. We can see that the effect of the fault

is isolated to the failed accelerator, because the throughput on that connection drops to 0 while the other connection continues operating as normal with slightly increased throughput. This is because the failed accelerator no longer contends for memory after the fault.

3.6 Conclusion

Modern FPGAs are increasing in size and thus implement hardware systems of increasing complexity. In this chapter, we described how software operating systems principles can help address some of this rising complexity. We propose Apiary, an FPGA operating system, and hardware primitives to provide similar functionality to what is expected in a software operating system. To evaluate Apiary, we implement a memory protection and allocation service and construct an end-to-end case study consisting of a Reed-Solomon encoder as a TCP service to show how the Apiary architecture can help in building out complex accelerator systems.

Chapter 4

BEEHIVE: A FLEXIBLE NETWORK STACK FOR DIRECT-ATTACHED ACCELERATORS

In this chapter, we discuss specifically our work on Beehive, which is Apiary's hardware network stack. We focus on the network subsystem in particular, because it is a key component of building a direct-attached accelerator system. A barrier to any hardware network stack implementation is the difficulty of meeting the full set of datacenter network operational requirements [145, 29]. Network manageability, diagnostic visibility, and interoperability are often non-negotiable requirements, made more complex by the rapid evolution in host network stacks to meet application and operational needs. Beyond core protocols, such as TCP/IP, modern applications require higher-level functionality like remote procedure call (RPC) processing, quality-of-service (QoS) management [212, 45], encryption [139, 48], application-specific load balancing [104, 47], and information flow control [69]. Deployment flexibility necessitates management features like virtual networking [125, 73, 54], access control lists [151], congestion control [127, 135], traffic prioritization [155, 88], and load balancing [164, 64, 183, 203]. Deployment maintainability requires dynamic support for network monitoring [213, 31], reconfiguration [38, 124], and debugging [192].

An example of a highly-flexible software network stack is Google's Snap networking system [145]. It is designed around composable message-passing engines, with modules for load balancing, network virtualization, network management, and custom transport protocols. New modules can be easily inserted anywhere in the stack, without re-engineering the rest of the stack. Our question is whether we can do something similar in hardware. Existing hardware network stacks are typically designed to support only a single application with minimal protocol complexity. Although some recent work has focused on flexible packet-level processing in

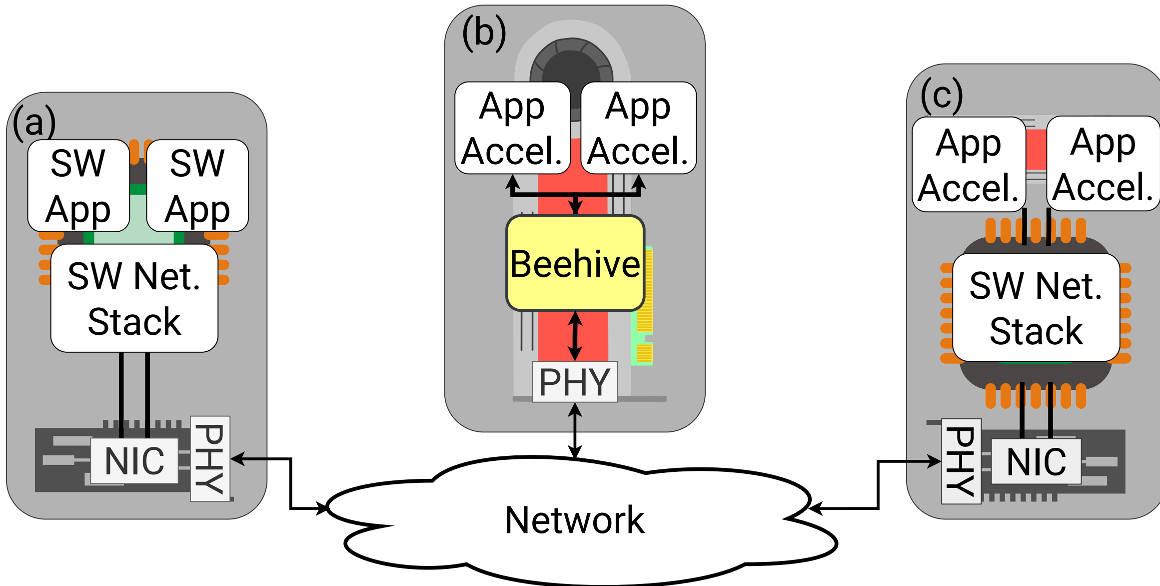


Figure 4.1: Representations of (a) a standard CPU server node; (b) a direct-attached accelerator using the Beehive network stack; (c) an accelerator using a CPU network stack.

hardware [134, 138], our aim is to support flexibility across the entire network stack, including transport and application protocols. Other work has looked at hardware offload of transport protocols, but these systems lack a range of essential network functions [178, 44, 25], or in the case of RDMA, require extensive engineering to make work in practice [29, 177].

This paper explores the design of an FPGA network stack that can realize the benefits of direct-attached accelerators while supporting the extensibility, incremental scalability, and manageability needed for production use. Flexibility is needed at multiple points in the network stack: in packet processing (layer 3), transport and congestion control (layer 4), the application layer (layer 7), and in control/diagnostics operating alongside, and using, the data plane. Adding new functionality, differentially scaling protocol elements to meet application throughput needs, or inserting a new load balancing policy should be simple, as it is in software, without the need to disrupt or re-engineer other layers.

We propose and implement Beehive, an open-source hardware network stack architected as a

collection of protocol functions that communicate via message-passing over a scalable network-on-chip (NoC). We provide automated tooling for managing differential scaling and load balancing of protocol elements, a control plane for diagnostics monitoring, and compile-time deadlock analysis. To make our design concrete, we implement Ethernet, IP, UDP, TCP, network address translation (NAT), IP-in-IP encapsulation, and additional support for control and debugging of network functions. Our implementation interoperates with Linux TCP and UDP clients, allowing unmodified remote procedure call (RPC) clients to use our accelerator.

For our evaluation, we implement Beehiveand evaluate it on FPGA. We show that it offers a 4×/1.5× improvement in end-to-end client RPC tail latency over Linux/user-level TCP relative to mediating accelerator traffic through the server CPU, and up to 31× higher per-core throughput than a state-of-the-art CPU kernel-bypass stack on small messages.

We implement two example applications using Beehive: erasure coding as a bandwidth-oriented application and distributed consensus as a latency-sensitive application. First, modern datacenter storage systems often use erasure coding for better storage efficiency than replication with comparable fault tolerance. We implement an erasure coding accelerator in Beehiveand show that, compared to a CPU-only version, the accelerator scales out to 62 Gbps using 20× less energy. Second, we show that accelerating a key piece of distributed consensus in hardware can reduce end-to-end median operation latency by 1.13×, with 1.14× better per-core throughput and 2× less energy than the CPU-only version.

In summary, we contribute:

- Beehive, a design framework to build efficient and complex hardware network stacks for direct-attached accelerator deployments in modern datacenters.

- An open-source FPGA implementation of Beehivethat includes tools and reusable components to build network stacks for accelerators that use different transport protocols, network virtualization, and layer 7 functionality.

- A demonstration of Beehive's ability to support scalability, flexibility, low latency, high

throughput, and energy efficiency by integrating and evaluating an erasure coding accelerator and a consensus accelerator.

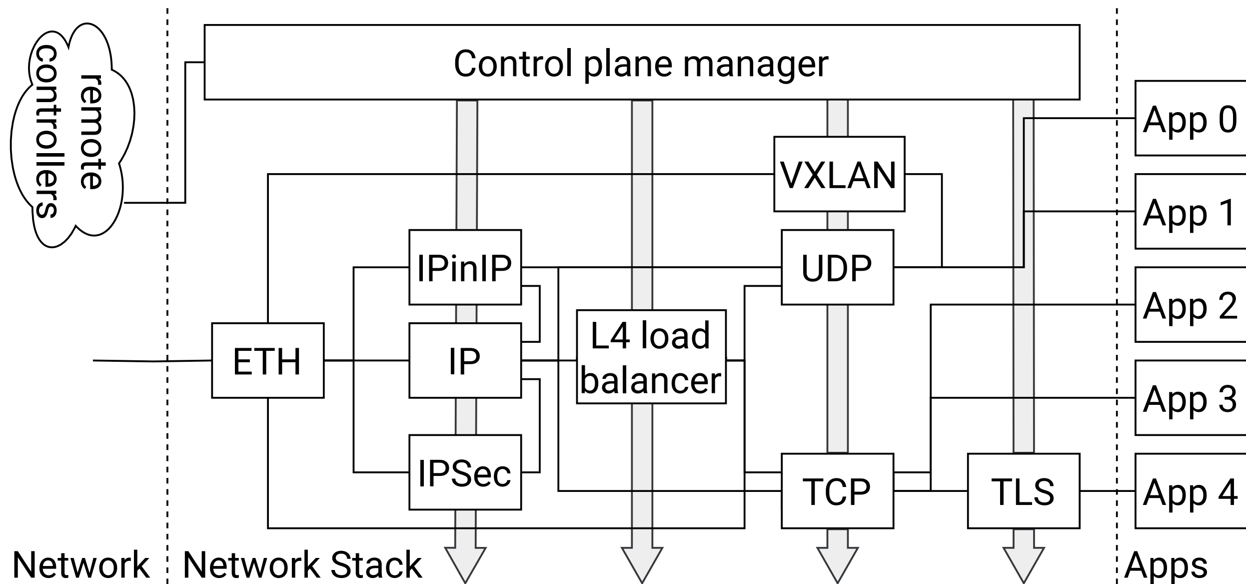


Figure 4.2: A high-level diagram of the type of network stack Beehive targets. Along with multiple transport protocols, this stack has IP-in-IP and VXLAN for network virtualization and a component for an L4 load balancer. The downward arrows represent control-plane communication, which potentially needs access any module internal to the network stack.

4.1 Design Goals

As described in Section 2.1.1, direct-attached accelerators can have appealing improvements in latency, especially tail latency. However, realizing this benefit in a realistic datacenter setting requires a hardware network stack that can be flexibly reconfigured to meet the needs of datacenter network management. Our overarching goal for Beehive is to build an open-source FPGA hardware design to support emerging applications for direct network-attached accelerators in a production environment. Figure 4.2 shows a high-level diagram of the type of network stack architecture we want to be able to support. Applications may only use some subset of these

	Std. Protocols	Modular	Scalable	Performant	Mgmt. Features	Open Source
Limago [178]	✓	✳	✗	✓	✗	✓
PANIC [138]	✳	✓	✗	✓	✗	✳
ClickNP [134]	✳	✓	✳	✓	✓	✗
LTL [41]	✗	✗	✳	✓	✳	✗
Beehive	✓	✓	✓	✓	✓	✓

Table 4.1: Beehive and prior work versus the goals in Section 4.1.1. The stars indicate partially meeting the goal.

protocols and network functions. We now discuss our specific design goals for Beehive

4.1.1 Beehive Goals

Standard client protocols. The vast majority of distributed applications that might benefit from the availability of hardware acceleration are designed to communicate using standard protocols such as IP, TCP, and remote procedure call (RPC). Our framework needs to be able to support unmodified client application and client host software communicating with the accelerator using these standard protocols.

Modularity. However, network stacks are not fixed. Requirements are constantly changing with new custom protocols (e.g. Google's Pony Express [145] or 1RMA [8]) and network functions. In order to facilitate rapid development and customization of the network stack, our framework must be modular, so we can compose or integrate new components with minimal to no modifications to existing components.

Scalability. Building a complex network stack potentially means supporting a variety of different components in the same design. Different components may be a bottleneck depending on the application workload. Thus, the architecture should be able to duplicate and scale out individual components, whether application or protocol logic, as needed.

Performance overhead and predictability. Since performance and performance predictability

are key motivations to offload the network stack, the stack should be able to deliver end-to-end application bandwidth at 100 Gbps with minimal jitter if the accelerators have the capacity to support it.

Management flexibility. Components in a network stack need to be able to interact beyond just passing packet data. For example, components need to be able to expose interfaces to the control plane for telemetry and debugging [66]. The control plane may also need to update state used by a protocol or network function, such as configuring the load balancer used to parcel work across application accelerator instances. Such configurability should be possible even in large designs without extensive manual optimization.

4.1.2 Comparing versus related work

As shown in Table 4.1, other related work does not meet all these goals. In terms of complexity, the Limago, a TCP engine written in Vitis HLS, is the closest to Beehive. However, it is not designed to allow for addition or replication of components within the stack, so it is limited in scalability and modularity. We discuss FPGA utilization comparisons further in Section 4.5.7. Unfortunately, we were unable to run Limago on FPGA using their code [78] to evaluate its performance, because the QSFPs did not come up on the FPGA board.

PANIC and ClickNP are the most similar architecturally to Beehive as they are both based on message-passing over an interconnect, leading to similar performance and modularity benefits as Beehive. Their implementations do not provide standard protocol support directly, but they could be extended to support the logic needed for these protocols. Additionally, their interconnects can limit their scalability. While working on the experiment in Section 4.5.3, we found PANIC's crossbar was unable to support more than 8 endpoints, 4 of which are always used by its infrastructure. In ClickNP, components are directly connected using FIFOs, potentially causing fan-out issues when duplicating components. Because ClickNP is not open-source, we were unable to compare to it directly.

4.2 Design

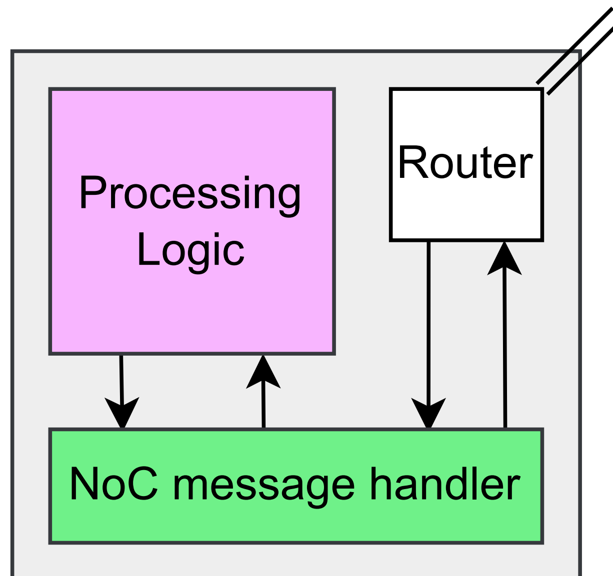


Figure 4.3: Architecture of a Beehive tile.

4.2.1 Beehive's architecture

The basic component in Beehive is the tile, shown in Figure 4.3. Each tile has a network-on-chip (NoC) router, some logic that handles NoC message construction and deconstruction, and some processing logic, such as a protocol layer, network function, or application. Tile routers are connected together to form the NoC topology. We do not require a particular topology, although our prototype uses a 2D mesh. We require that the NoC is reliable, point-to-point ordered, and uses deterministic, deadlock-free routing.

A network packet is processed or constructed by passing NoC messages through a chain of tiles. A NoC message consists of one header flit followed by some number of body flits. The header flit typically contains data only relevant to NoC-level routing, such as source and destination tile coordinates or number of body flits. The body flits typically consist of both metadata flits containing packet header fields and a number of data flits carrying unprocessed packet payload.

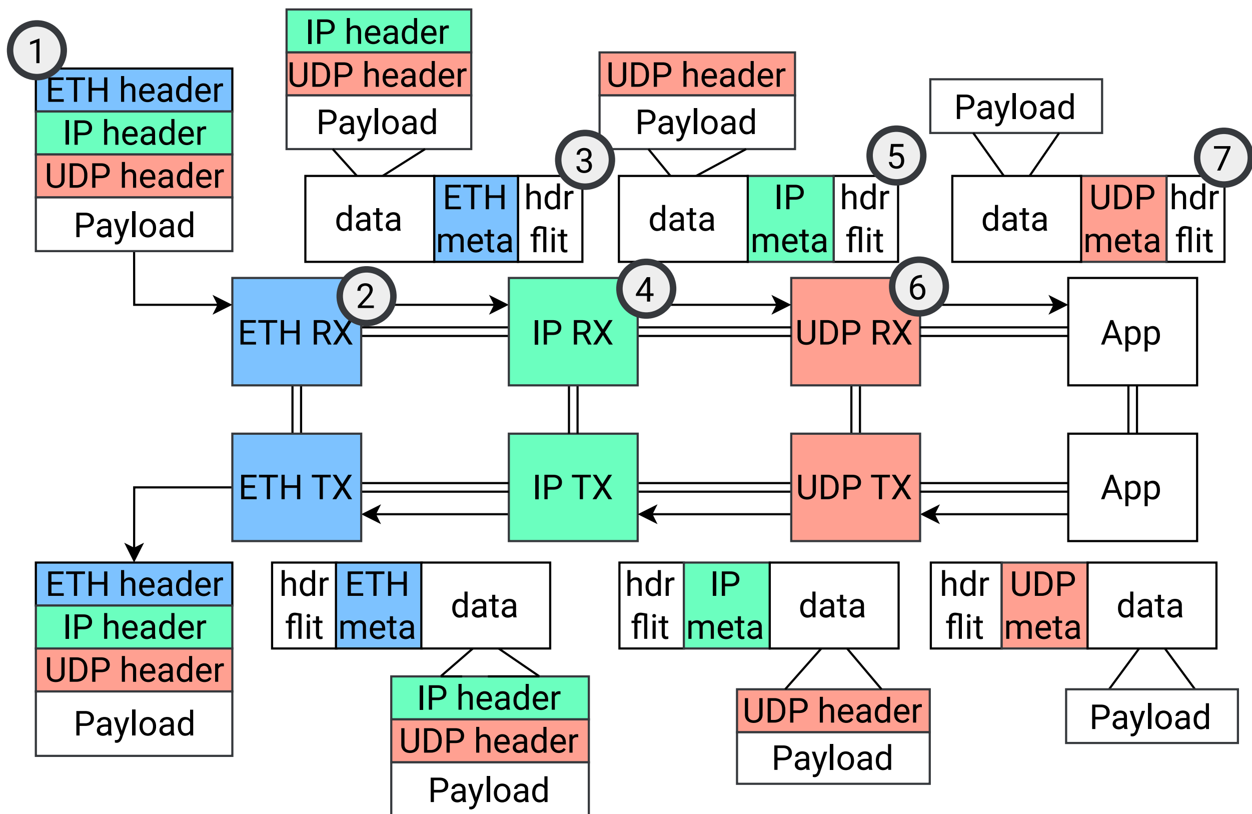


Figure 4.4: The flow through which a packet is processed or constructed in Beehive.

Each tile hop is responsible for determining the next tile that a message should be sent to. This design is in contrast to earlier work which assumes that routes can be fully determined on packet arrival [138]. We discuss this decision in more detail in Section 4.2.4. This component may vary in complexity from a static CAM to more complex logic, such as content-based routing. The set of possible message chains is known ahead of time for deadlock analysis, described in Section 4.2.5.

4.2.2 Processing a packet

Figure 4.4 shows an example of a basic UDP stack in Beehive, with a UDP packet moving through the receive and send paths. On the receive side, an Ethernet frame enters the Ethernet tile, which

has ports for the I/O from the transceivers in addition to the ports connecting to other tiles. The processing logic within the tile parses and removes the Ethernet header, realigning the data. This is then turned into a NoC message consisting of a header flit, a metadata flit with the parsed Ethernet header, and some number of data flits containing the remaining packet data. The routing component in the Ethernet tile uses the type field in the Ethernet header to determine that the message should be passed to the IP tile. The IP tile similarly parses the IP header, validates the header's checksum, and then creates a NoC message to be sent to the UDP layer. Finally, the UDP tile parses the UDP header, validates the packet's checksum, and generates a NoC message to be sent to the application based on the port in the UDP header. The transmit path runs similarly, except instead of parsing headers from the data flits, headers are added by each protocol tile. After the Ethernet tile adds on the Ethernet header, it is sent out the ports for I/O with the transceivers. This incremental composability is good for our goal of modularity as it makes it easier to insert new functionality between stages.

While there is only one possible destination for the tiles in this design, there can potentially be multiple endpoints, such as other protocols (e.g. TCP connected to IP), network services (e.g. network virtualization), or replicated tiles for higher bandwidth. With replicated tiles, there are multiple ways to decide on which tile should receive an incoming packet. The simplest method is to distribute packets between them in a round-robin fashion. However, more complex scheduling may be necessary if a tile holds state for particular flow. In this case, it is important that packets from the same flow always go to the same tile. This distribution can either be integrated within a tile or placed in a dedicated tile. We discuss how we distribute packets to duplicated tiles in Section 4.4.

4.2.3 Message-passing interconnect

Being able to compose elements is essential for facilitating customization. We opt for a message passing model. This is beneficial for modularity, because defining a message-passing format allows us to standardize the physical interconnection between components, a recognized benefit in SoC design [52], and makes it easier to chain offloads together. ClickNP [134] and PANIC [138],

two modular packet processing frameworks, have also used a message-based approach. The message passing can be done over dedicated connections, which is the approach used by ClickNP, or a NoC which is used by PANIC.

We prefer a NoC interconnect for two main reasons related to our goal of scalability. First, we can take advantage of the multiplexing provided by the NoC routers. Certain tiles may interact with many other tiles, e.g. if we instantiate multiple copies of the same component or common services such as memory buffer storage. Direct connections can lead to large multiplexers and wires with significant fan-out. Although we could create specialized pipelined multiplexers and arbiters, these essentially look like NoC routers.

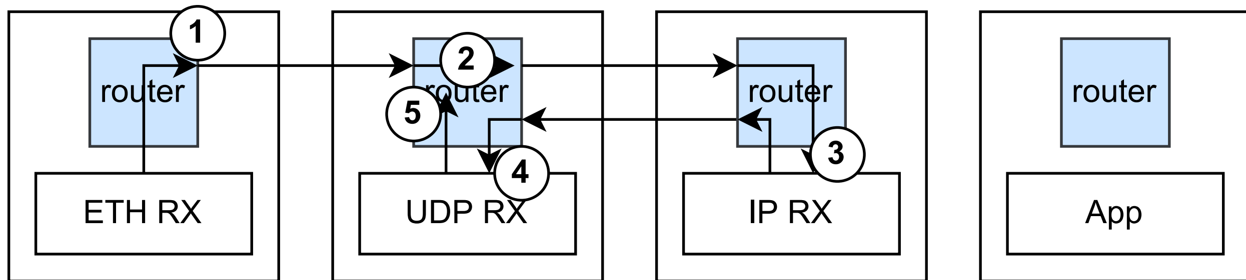
Second, we would like the interconnect wiring to remain stable whenever possible. In the ClickNP model, top-level wires are determined by the computational graph. If we wish to form a chain that links together two components that did not communicate before, we must add new interconnect wires, which are typically the longest wires. A NoC allows us to reuse physical wiring to chain any elements that exist in the design, as long as we are careful with deadlock.

These scalability benefits apply both to the data plane and control plane. We discuss the benefits further for the control plane specifically in Section 4.2.6.

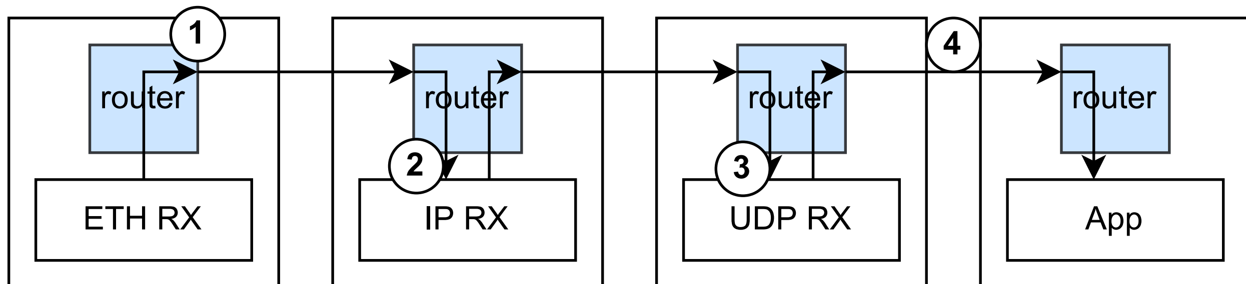
4.2.4 Tile chain routing

In addition to NoC-level routing, Beehive routes at the network packet level to determine the sequence of tiles that need to be chained together. We considered two routing methods: node-table routing, where each tile determines the correct next tile, and source routing, where the chain of tiles is completely determined when the first NoC message in the chain is created, such as when a packet is first received from the network. We use node-table routing, because certain classes of traffic we want to support for interoperability require per-flow state or non-trivial protocol processing to fully determine the chain of tiles.

Specifically, we consider routing for traffic that is either encrypted or is for layer 7. Encryption may obfuscate parts of packet payloads that are needed to fully route a packet, which would require the ingress tile to handle the decryption. An application request can span multiple pack-



(a) This tile assignment deadlocks due to the order that the packet needs to be processed in versus the order of the NoC links it traverses.



(b) This tile assignment is able to avoid deadlock since the packet acquires NoC resources in order.

Figure 4.5: An example of how Beehive tile assignments affect deadlock. Beehive takes advantage of protocol layer ordering, so a packet always acquires NoC resources in the same order.

ets. Which application tile should receive an RPC may depend on the RPC header or even the contents of the request. Further, the packets of one request, which may not fit in the first packet, could be reordered or interleaved with other requests. To properly route such requests, an ingress tile would need to assemble or reorder the stream, further complicating the implementation. In both cases, the ingress tile would need to implement significant, high-level protocol logic which is detrimental for modularity.

4.2.5 Deadlock

As with any NoC-based design, avoiding message-based deadlock must be a consideration. We note that NoC deadlock detection, avoidance, and recovery is a complex problem with a whole

body of research behind it [51, 130, 156, 24, 180].

NoCs can deadlock in two ways: at the routing level and at the message passing level. To prevent routing-level deadlocks, we employ dimension-ordered routing [51]. Message passing deadlocks are a bigger concern in Beehive, because any tile can route to any other tile at runtime. This means that our routing resources can get exhausted. The deadlock in Figure 4.5a is an example of this, in which the UDP RX tile must route east twice in one chain, and it cannot route east a second time.

We apply resource acquisition ordering to solve this problem. Resource ordering can be imposed by taking advantage of the fact that protocol layers and services are composed in certain orders. Although packet routing is dynamic, we assume that all possible paths through the network stack for supported packet types are known when the network stack is compiled. As a simple example, Figure 4.5 shows different topologies for the receive path of a UDP stack. Beehive's NoC uses wormhole, dimension-ordered routing. The packet should be processed by Ethernet, IP, UDP, and then the application. With the tile layout in Figure 4.5a, the route from the Ethernet to IP tile passes through the UDP tile's router (2). As the UDP tile attempts to pass the packet along to the application (4), it must reacquire a NoC link that is still in use (5) and is thus deadlocked. If tiles are laid out as in Figure 4.5b, no resources need to be reacquired, and the packet can be processed successfully.

We statically analyze all message paths in our prototypes at compile-time to avoid deadlock by creating a resource dependency graph that takes into account every possible path through the network stack. If a message path is found that could cause deadlock, the designer should modify the tile layout to one that does not.

Repeated protocol headers (e.g. two IP headers in the IP-in-IP protocol) break resource ordering. In Beehive, we choose to duplicate tiles (e.g. two IP RX tiles). If tiles are too expensive to duplicate, a potential solution is adding buffers to break dependencies [130, 188]. These buffers give space for the NoC to drain into, freeing routing resources.

4.2.6 Control plane interfaces

For manageability, network operators need to be able to reconfigure protocol components from an external controller over a transport-layer connection. In Beehive, we choose to use an additional separate message-based, routed NoC for the control plane rather than a dedicated control bus. This is because control plane management also benefits from a structured interconnect for scalability reasons.

First, for complex designs with a large number of components, it becomes costly to run dedicated, ad-hoc wires to every tile. Second, we want configuration to be over a reliable transport. This requires the control plane to use the transport layer, and a NoC enables this without physically coupling the component to the transport layer. This also enables us to add specific control plane management tiles to orchestrate state modifications. We describe a specific example in Section 4.3.5.

Because the control plane has lower performance requirements, in Beehive we use a separate, lower-width NoC. This also prevents control plane traffic from contending for the same resources as long dataplane chains in the deadlock dependency graph, so there is more flexibility in placement.

4.2.7 Application interfaces

Many application accelerators process requests at a coarser granularity than a packet, so they need the ability to communicate with the transport protocol layer and request data from a particular flow rather than being pushed packets in the order they arrive. While we could use dedicated wires for this communication, it can also benefit from the use of the NoC.

The NoC provides a convenient structure to multiplex between duplicated application tiles connected to the same transport layer in a scalable manner. The modularity provided by message passing on the NoC also allows an application to easily interface with any protocol in the network stack while reusing existing wires if, for example, we want to switch from TCP to a custom reliable transport protocol. Finally, the standardized NoC interface enables easy insertion of filters on the

application's NoC messages, so network operators can enforce policies, such as dropping network traffic to or from non-whitelisted nodes. We describe the application NoC interface to our TCP layer in Section 4.3.4.

4.3 Implementation

To demonstrate the Beehive approach, we built a set of core protocol tiles, network functions, and applications. For protocols, we implement tiles for Ethernet, IPv4, UDP, and TCP. For network functions, we implement an IPinIP encapsulation layer and a NAT layer for network virtualization. For applications, we implement a Reed-Solomon encoder and an accelerator for a viewstamped replication node. These applications are described in more detail in Section 4.4.

We also describe our tooling that we developed to lower the effort required to maintain multiple designs and integrate new components. All of Beehive is implemented in standard SystemVerilog and was tested on an Alveo U200 communicating with standard CPU clients using a Linux or kernel-bypass network stack. We embed our Beehive prototype within Corundum [75], an open-source 100 Gbps NIC, in the application slot to provide FPGA-specific infrastructure, such as the Ethernet MAC. Corundum does not provide any higher-level packet processing logic for Beehive.

4.3.1 Network-on-chip (NoC)

We use the 2D mesh NoC from OpenPiton [30] with some modifications. The NoC is wormhole-routed, uses dimension-ordered routing, and is full-duplex. We widen the NoC from 64 bits to 512 bits to match the width of the Xilinx MAC IP core, so it has a maximum throughput of 128 Gbps when running at 250 MHz and increase the flit width to 512 bits. Because the NoC only relies on the top 64 bits of the first flit to do NoC routing, we are able to reuse the NoC without further modification by making the top 64 bits of our first header flit the same as the original NoC header. The maximum payload size for a NoC message is 256 MiB.

4.3.2 Protocol tiles

Protocols are implemented as streaming components, so they begin to transmit the next NoC message as soon as possible rather than storing the entire NoC message before forwarding. This is done to reduce latency as header parsing can be overlapped with payload copying. This is especially important when chaining, because each layer of header adds an extra layer of parsing.

The Ethernet, IP, and UDP tiles construct or remove the appropriate headers and calculate checksums, as shown in Figure 4.4. The Ethernet receive processor can handle VLAN tagged packets. Our IP layer does not support IP fragmentation as our intended use case is for internal datacenter services.

One of the more difficult aspects of removing the headers from network packets is that certain protocols (e.g. IP or TCP) allow headers to have options, so the headers are not a fixed width. This means removing a packet header often requires removing a variable number of bytes from the stream. We implement this by appending two lines of data and then using a shifter to remove the required amounts of bytes.

For a protocol, we place the receive and transmit engines in separate tiles. This is because they are streaming and each router has one input and one output interface, so one engine will utilize an entire router's bandwidth if running at 100 Gbps. Since the packet-level protocol layers do not share state between their transmit and receive sides, this is a straightforward split. The exception to this is the TCP engine which we discuss further in Section 4.3.4.

Protocol tiles also have optional hash tables that use the 4 tuple as the key for load balancing to downstream replicated tiles. We set up initial packet-level routing within the tiles at compile time when we build the FPGA image. The hash table can be rewritten during runtime via the control plane described in Section 4.2.6. Any packet that does not have an entry for a next hop (e.g. traffic with an unsupported protocol) is dropped to filter out unwanted traffic.

4.3.3 *Buffer tiles*

In Beehive, we also have buffer tiles that hold large blocks of memory. In our current prototype, these buffers are large BRAMs, but the backing buffer can also be DRAM. These buffer tiles are accessible to any other tile in the system via NoC messages. This allows us to have shared buffers between tiles, so that multiple tiles can share state when needed.

4.3.4 *TCP engine*

To evaluate how Beehivecan support reliable transport, we prototype a TCP engine that implements server-side TCP. It can receive connection setup requests, generate sequence and ACK numbers, and support fast retransmit and window-based flow control [37]. Currently, it does not support selective acknowledgments, initiating connections, or congestion control. Full TCP offload functionality has been demonstrated by previous work [178] and could be integrated into Beehive.

We split the TCP logic into receive and transmit engines. The receive engine is responsible for determining if received data is in order, calculating the next ACK, and processing ACKs for the transmitted data. The transmit engine is responsible for separating out buffers for sending and updating the sequence number for the transmitted stream.

We use two optimizations to handle state shared between the receive and transmit engines when they are both processing the same flow. We handle this in two ways. First, we divide flow state into two BRAMs by which engine writes the data to prevent write conflicts. Second, we take advantage of the asynchronous nature of the transmit and receive streams in TCP to tolerate slightly stale state and avoid bypassing state when the two engines are processing the same flow. For example, the transmit engine reads the current flow state with the ACK number for the received stream as ACK_RECV_1 in cycle n. Meanwhile, the receive engine has processed a packet and updated the ACK number to ACK_RECV_2 in cycle $n+1$. The transmit engine can still use ACK_RECV_1 as long as it still uses all the other state it read in cycle n. Functionally, this is the same as if the received packet had been received slightly later and processed after the transmit

engine had sent its packet, which is allowed due to the assumptions of TCP.

While the TCP engine has an RX router and a TX router like the other protocol tiles, the send and receive paths in TCP must share state. For example, the transmit path needs to know for which packets it has received acknowledgments. We choose to support sharing by running dedicated wires between the tiles. Every receive path only has one corresponding transmit path, so wires do not fan out. We could implement state sharing over NoC messages, but the state is read and updated frequently, so the frequent NoC messages needed for state updates would encourage these tiles to be placed close to each other on the NoC anyway.

On the completion of the 3-way handshake, the TCP engine sends a NoC message to notify an application tile based on the destination port for the connection. On the receive side, the TCP engine lets an application specify the size of the request it should be notified for with a NoC message. When enough data has arrived to satisfy that request, the TCP engine sends a notification message back to the application with the buffer address where the data requested has been stored. The application then retrieves the data from the buffer for processing before sending another message to the TCP stack when it has finished using the data.

The TCP engine implements a similar interface for the transmit engine where the application can request space in its transmit buffer of a certain size. The TCP engine sends a notification when there is room in that buffer with the address where the data should be stored. The application then copies the data into the buffer and notifies the TCP engine.

The receive and transmit buffers shared by the TCP engine and applications are statically allocated at compile-time with constant addresses. This means that even if very little data is stored into these shared buffers, the memory is still allocated, which is potentially a waste. There is also no isolation to prevent applications from accessing different buffers that are not for their specific connections. We plan to address these limitations using Apiary, which is described in more detail in **??**.

4.3.5 *Network function tiles*

We implement both IPinIP encapsulation and an IP NAT. For both tiles, the control plane can dynamically update the table mapping virtual IPs to physical IPs, which occurs when the a client machine migrates. To change this mapping, we implement an internal controller as a separate tile that receives an RPC over TCP from an external controller. The internal controller utilizes the control NoC to send NoC messages to the IP encapsulation or NAT tiles with the information needed to update their tables. Finally, the internal controller sends a confirmation response to the external controller.

4.3.6 *Debugging and logging*

In Beehive, tiles may keep logs, and we provide UDP and TCP-based protocols to externally fetch logs. Each log is associated with a particular port and exposes an interface on the NoC to the network stack for readback. The layer 4 receive tiles are responsible for directing packets to the appropriate log interfaces. The log read interface keeps a small buffer for requests and drops requests when it is full. The client program reads out the log an entry at a time and resend requests for any entries for which it does not receive a response.

This logging ability was invaluable for debugging TCP when running on an FPGA. TCP is underspecified and the main verification is running against a common implementation, such as the Linux kernel [36], so we needed to run it on an FPGA to verify that it behaves as expected. The reduced visibility in this setting increases the difficulty of the already hard task of debugging a TCP implementation, due to the asynchronous and non-deterministic setting where certain bugs are dependent on the available bandwidth and loss events. As a result of the asynchrony, we need a cycle accurate trace for proper replay, because the TCP engine may behave differently depending on the timing of events (e.g. it may drop different packets). As a result of the bandwidth-dependence, we cannot rely on `tcpdump` to collect traces, because of the possibility different packets might be dropped by the engine versus `tcpdump`.

We inserted tiles that log information about TCP packet headers into the processing between

the TCP and IP layers. These tiles have two NoC interfaces: one is used to forward packets to and from the TCP engine and logs the header information with a cycle timestamp, the other interface allows the logs to be read out over the network in response to a request sent over UDP. Because the logging tiles are embedded within the fabric, they can record the exact timing and sequence of packets that entered and exited the TCP engine. Once this log is collected, we are able to replay the log in a cycle-accurate manner using the recorded timestamps by replacing the logging tiles with an interface to our trace replay framework.

4.3.7 Tooling

We developed a set of tools to lower the engineering effort to create new designs, such as generating portions of the Verilog (e.g. top-level wiring for NoCs) or performing compile-time deadlock analysis. The design configuration is passed to these tools via an XML file, which contains the design dimensions as well as an element for each NoC tile endpoint. At minimum, this element contains tags specifying a name to use for the endpoint as well as its X and Y coordinates. It may also contain fields with information for generating the tables used for determining the correct next hops.

Given the dimensions in the XML file, we generate declarations of all the top-level wires between tiles. We also generate the subset of the port connections for each tile that correspond to wires between NoC routers and connect the appropriate wires for the tile configuration. We choose not to generate the whole tile instantiation, because certain tiles need to maintain additional ports for I/O, such as the Ethernet MAC.

The XML file also enables us to check whether the high-level topology of the NoC is sound. For example, we check if two tiles have the same X and Y coordinates, and all NoC coordinates are within the expected dimensions of the design. Because a 2D mesh must be a rectangle, this also gives us the opportunity to automatically generate empty tiles that just contain a router, as in the bottom rightmost tile in the UDP stack shown in Figure 4.8a. We also use information about the NoC topology and next hops in the XML file to generate a resource dependency graph that we analyze for cycles to ensure a deadlock-free design. Figure 4.6 is a visualization of the layout

generated by the XML file for the consensus witness design in Section 4.4.2.

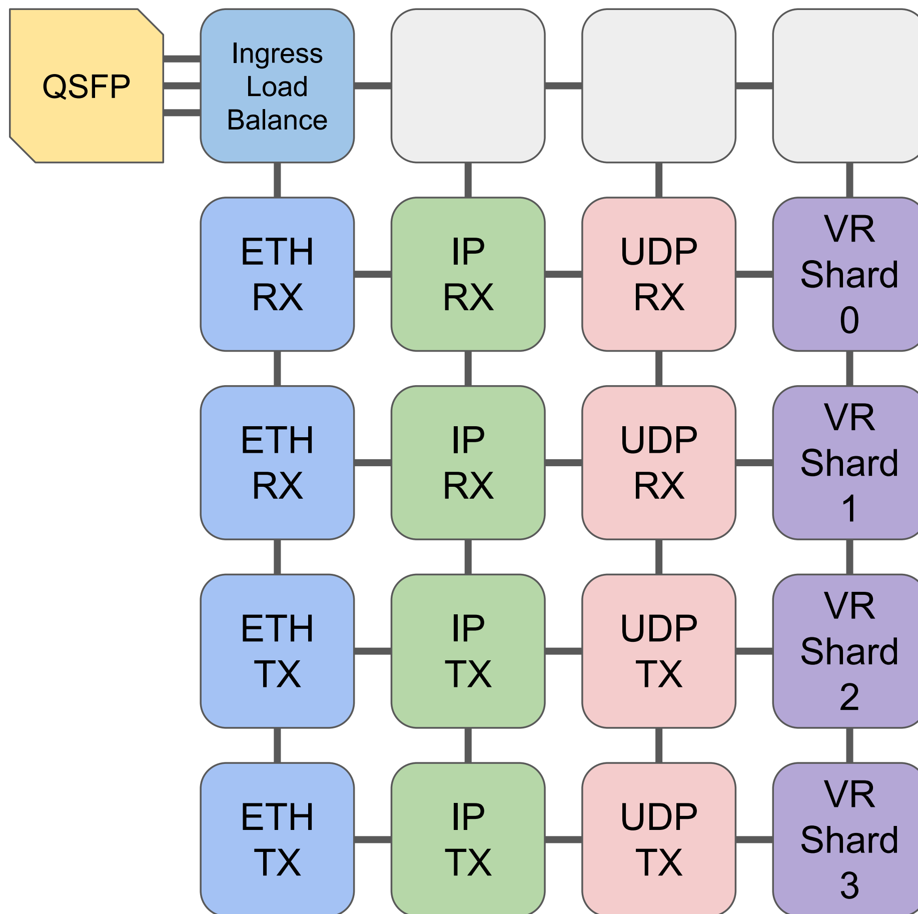


Figure 4.6: Beehive tile layout for Viewstamped Replication.

4.4 *Integrating with Beehive*

4.4.1 *Erasure coding*

To demonstrate the benefits of Beehivefor a throughput-oriented application, we integrate an accelerator for Reed-Solomon encoding. Erasure codes such as Reed-Solomon (RS) are commonly used in distributed storage systems to achieve high resilience to disk failures with modest storage overhead [95, 174, 120]. An RS encoder adds redundancy bits to input data at a pre-set ratio,

striped across storage servers. If some storage elements fail, the remaining blocks from the stripe can be combined with these extra blocks to regenerate the missing blocks.

We configure our system to use an (8,2) code (8 data blocks and 2 redundancy blocks) to emulate a storage system that could tolerate up to two disk failures. We integrate an RS encoding accelerator operating on 4KB requests into Beehiveas a UDP application, instantiating four copies of the application to scale out. The accelerator is stateless, so any request can go to any copy. We introduce a front-end round-robin scheduler tile to distribute work among the RS tiles. Each RS tile also logs metadata to calculate bandwidth.

4.4.2 Consensus witness

To demonstrate how Beehive performs in a latency-sensitive, communication-intensive application, we construct a consensus system that uses FPGA-accelerated witness nodes. Consensus algorithms are an essential part of many deployed distributed systems as they enable a strictly consistent order for stateful client operations even in the face of failures and message delays/retransmissions. Most consensus algorithms [140, 42, 161] follow a common pattern: an elected leader proposes an order for arriving client requests, verifies with a set of replicas that it is still leader, and commits the request. It then performs any necessary application logic (e.g., to update state), replies back to the client, and informs the other replicas, so that they can also perform the application logic in the same order. Because there are multiple round-trips between nodes to complete one round of consensus, message-handling latency and tail latency are especially important [214].

A common type of application built on top of consensus is a key-value (KV) store. To achieve higher throughput, the key space is often sharded with a leader and replica set for each slice. However, even with sharding, consistent reads can be expensive, because the leader must validate, each time, that it is still the leader before replying with the value stored with the key. As a result, it is common in practice to configure the system to return stale reads, allowing the leader to reply immediately [98, 49, 105]. This places a burden on the client developer to handle the (rare) case where a failover can lead to inconsistent client data.

In our evaluation, we show that a consensus accelerator can help reduce the cost of consistency [106], especially in a multi-shard setting. Our accelerator operates as a witness, that is, it only validates the leader and tracks the operation order; it does not execute client operations. Single node fault tolerance can be achieved with one leader, one witness, and one replica. To add further fault tolerance, we add additional witnesses and replicas. For example, two-node fault tolerance can be achieved with one leader, two witnesses, and two replicas. To validate a read or write operation, the leader only needs to receive a verification from the witnesses before replying to the client. The witness can be designed in hardware to reply with low and reliable latency.

Prior work [106, 105] has demonstrated full offload of consensus and application logic to an FPGA. We target a use case where application logic remains on CPUs and only a portion of the consensus protocol is run on Beehive. Importantly, this requires no change to the CPU-based application running on top of the consensus engine. This is advantageous as consensus algorithms are commonly used as a building block in larger distributed systems, so this allows accelerated consensus to be used without requiring the whole application to be ported to hardware. We also demonstrate how Beehivecan be used to scale a consensus system to support multiple shards, which previous work did not explore.

Our witness protocol is based on a modified version of the Viewstamped Replication (VR) used in previous studies of high-performance consensus [168]. VR witnesses are integrated into Beehiveas UDP applications. To handle multiple shards, we use one VR witness tile per shard. Unlike the RS encoder, the VR witness is not stateless and requests for a shard must always go to the same tile. We distribute work to the VR tiles by matching on the destination port number.

4.5 Evaluation

Our evaluation tests Beehive's ability to support scalability, low latency, and flexibility in a range of network stack configurations. We begin by evaluating Beehive with UDP and TCP microbenchmarks designed to test RPC performance and then evaluate two case studies: Reed-Solomon encoding acceleration and Viewstamped Replication acceleration.

4.5.1 Setup

We use Vivado 2021.2 for building our FPGA images. Beehiveis configured on an Alveo U200 at 250 MHz. The FPGA and the clients are connected to an Arista DCS-7060CX-32S-R 100G switch with jumbo frames enabled. We use five machines during evaluation with Turboboost disabled. All of them have Mellanox ConnectX-5 100G NICs and are running Ubuntu 20.04. Two machines have Intel Xeon Gold 6226R CPUs; the other three machines have Intel Xeon Gold 5218 CPUs.

In experiments where energy is measured, we use the RAPL counters on the CPUs and the Alveo CMS registers on the FPGA. For CPU energy experiments we use a two-socket machine, so we run all the application and network processing code on one socket and poll the counters from the other socket. We only use RAPL's CPU counters, which is an underestimate as we do not include DRAM energy or network interface energy. On the FPGA, we use the Corundum framework to read the CMS registers that report instantaneous power and current usage [205]. We poll these counters every second to calculate energy over the benchmarking period.

4.5.2 Baselines

Hardware Network Stacks (PANIC and Limago): We compare against PANIC, an FPGA-based smartNIC framework, for our UDP echo microbenchmark. We are unable to compare against PANIC for our other applications using UDP, because they require scaling to more tiles than PANIC supports, and PANIC's memory allocation makes it unwieldy to generate responses of a different size than the request. We also cannot compare against PANIC for our TCP microbenchmark, because it cannot support reliable transport applications. We also evaluate in PANIC's original simulation evaluation infrastructure, because their released code does not include an FPGA flow. We integrated it into Corundum as suggested in the documentation, but we were unable to get it to meet timing for the Alveo U200. While they used an ADM-PCIE-9V3 [55], both our board and theirs have 16nm FPGA parts. The Alveo's FPGA part is also comparable in resources available to the ADM-PCIE-9V3. For these reasons, we think the comparison is fair.

We compare against Limago, an HLS TCP stack, for our hardware utilization. We were unable

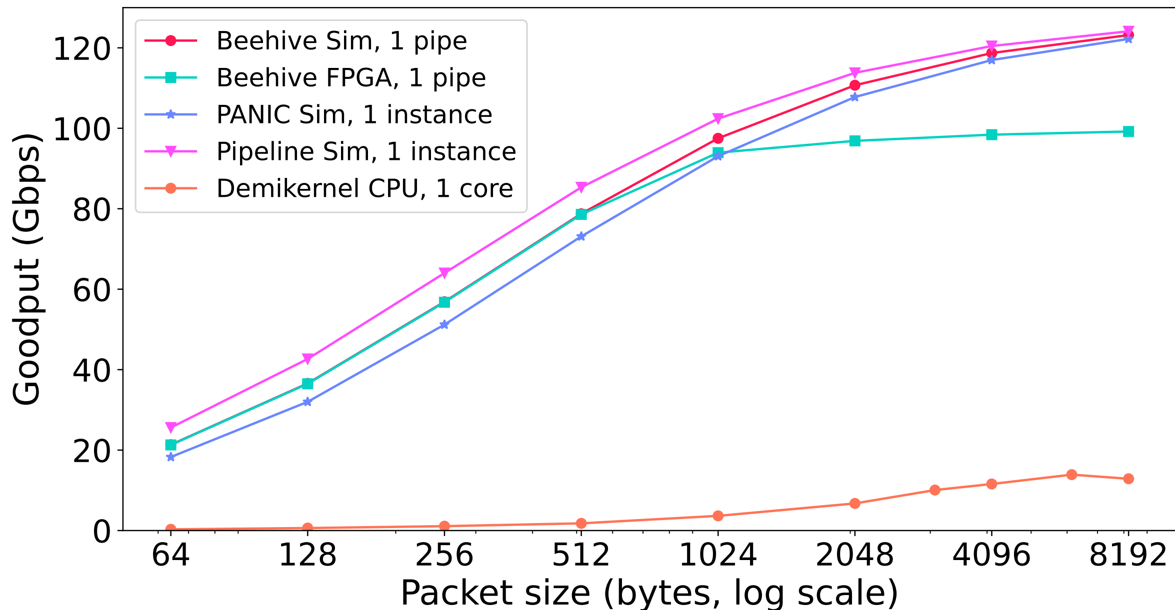


Figure 4.7: Packet size vs. goodput for a UDP echo application. Beehive and CALM perform almost identically across all packet sizes and outperform Demikernel.

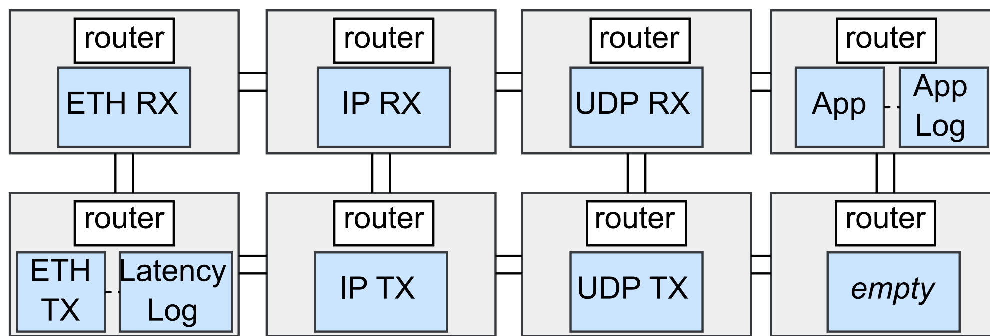
to run benchmarks on it, because the QSFP links did not come up when the image was put on the board.

Software Network Stacks (Demikernel and Linux): We also compare against Demikernel [211], an optimized DPDK network stack, in cases where it is faster than Linux. This is only the case in the UDP echo benchmark. Otherwise, we compare against Linux's network stack.

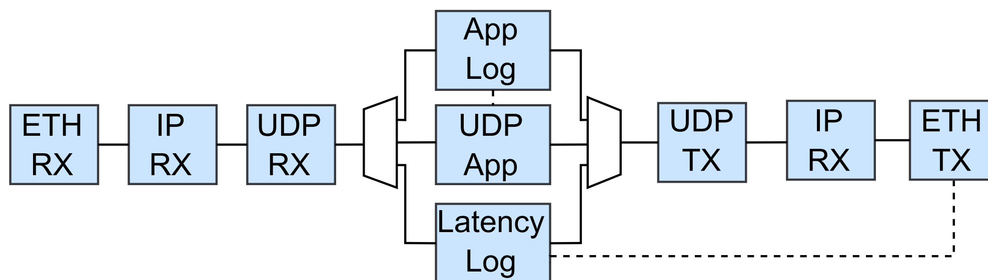
4.5.3 UDP echo

Throughput: We compare UDP echo goodput for Beehive(shown in Figure 4.8a) and Demikernel on different packet sizes. We also evaluate these against an FPGA with a pipelined UDP network stack design where the protocol engines are connected directly (shown in Figure 4.8b), and a UDP network stack implemented within the PANIC framework which we will refer to as CALM.

In our experiments, the Demikernel server runs on an Intel Gold 6226R machine, and we use three Intel Gold 5128 machines as clients using the standard Linux network stack. We spawn



(a) Setup for Beehive's UDP stack



(b) Setup for the fixed pipeline UDP stack

Figure 4.8: Diagram of the UDP stack architectures used for the echo microbenchmark

the number of client threads that yields the highest server bandwidth for that packet size, and they send in an open-loop manner. We give the server a single core to compare against Beehive's single application tile.

For Beehive, we run a packet generator on another U200 FPGA. This is because the client machines used for the CPU experiments cannot generate enough traffic to saturate the FPGA. We use 7 tiles in total: we separate the Ethernet, IP, and UDP layers, and then we separate their receive and transmit paths for 6 tiles and then one tile for the application.

For CALM, we implement a UDP echo server within its framework starting from their publicly available code [157]. We use 3 tiles to implement the echo server: one providing a fixed UDP receive path, one providing the application, and one providing a fixed UDP send path. We were unable to modify PANIC to support more than 8 tiles, only 4 of which are available for user

functionality, so we could not make every layer into a tile as we do in Beehive. We note that this means it is less flexible than Beehive's network stack, because we lose the opportunity to easily insert network functions or alternate protocols alongside the UDP paths.

Figure 4.7 shows our throughput benchmark results. Beehive and CALM provide similar performance despite Beehive having more tiles. Both achieve line rate at 1024 byte packets. Beehive on FPGA levels out at this point, because the actual Ethernet link has a maximum bandwidth of 100 Gbps. However, in simulation, both Beehive and CALM continue to scale to the theoretical maximum of 128 Gbps. The pipelined implementation is slightly better than Beehive, due to the overhead of constructing and deconstructing NoC messages. However, this difference decreases as payload sizes increase since the extra header flits are amortized over a larger payload. The optimized CPU stack remains far below maximum bandwidth even with jumbo frames. The performance difference is especially pronounced at small packet sizes where Beehive is able to sustain echoing 9 Gbps of 64-byte packets (18392 KReq/s) whereas single core Demikernel provides 0.3 Gbps (584 KReq/s), a 31× speedup.

Latency: For our latency experiment, we use Beehive and a single client thread to ping-pong a single 1-byte UDP packet. We record the latency by tagging the packet with a timestamp when it enters the network stack at the Ethernet parsing layer, taking another timestamp when it finally exits the Ethernet layer on transmit, and recording both timestamps into a log which we read back over the network. The latency through Beehive is 368 ns (92 cycles). Similarly, CALM UDP latency is 362 ns, although their system is less flexible than Beehive.

4.5.4 TCP throughput

To characterize the throughput performance of our TCP engine, we run a single-connection experiment and measure unidirectional send and receive performance across a range of packet payload sizes. Because Demikernel's TCP implementation is optimized for latency, it performs worse than Linux on this experiment, so we configure Demikernel to use Linux TCP as its backend. The sending application sits in a tight loop, submitting data into the network stack as fast as possible; the receiver pulls data out of the network stack without doing further processing on it.

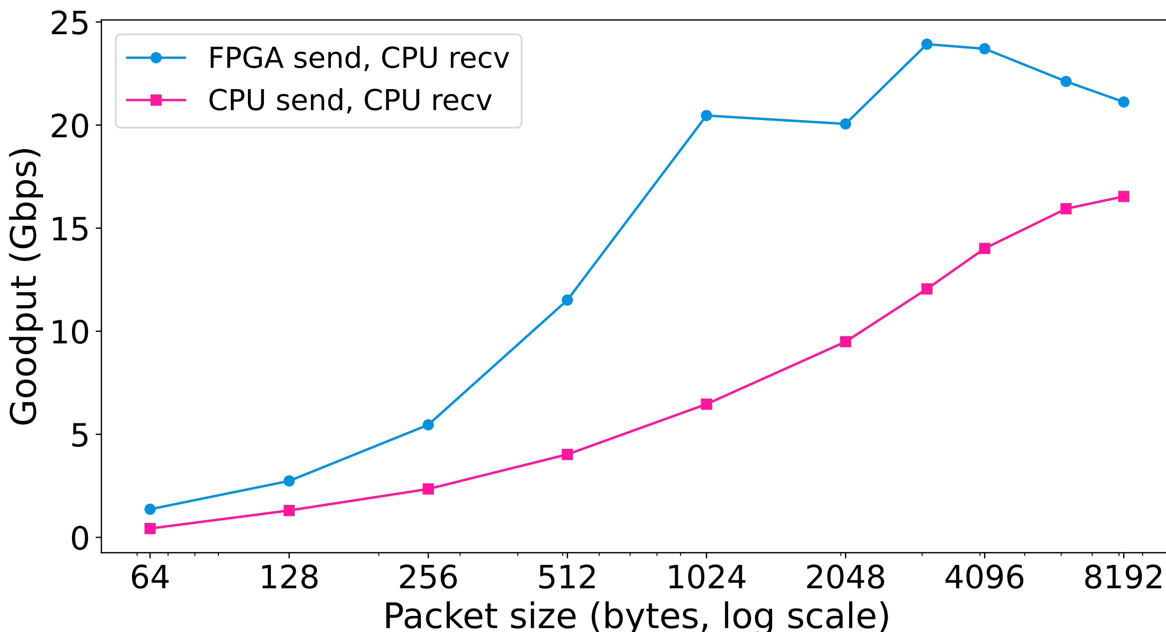


Figure 4.9: Packet size versus goodput for Beehive and Linux TCP send. The (CPU send/FPGA receive) is omitted, as it is approximately the same as (CPU send/CPU receive) due to the CPU send path being the bottleneck.

We vary whether the sender or the receiver is the FPGA or the CPU. The results are shown in Figure 4.9. We omit the (CPU send/FPGA receive) results, because they are almost the same as the all-CPU configuration; in both situations, the CPU sender is the bottleneck. The CPU is more efficient at streaming TCP data than UDP data because it allows batching data into jumbo frames. By contrast, Beehive's TCP stack is slower than its UDP stack, because of the complexity of stateful packet handling in hardware. In particular, our TCP engine is designed to only achieve full bandwidth across multiple simultaneous connections. Even so, Beehive outperforms Linux TCP across all request sizes. The speedup is most pronounced at small packet sizes, where Beehiveachieves 2666 KReq/s versus the CPU's 843 KReq/s, a 3.2× speedup.

Table 4.2: Energy consumption and goodput for Reed-Solomon encoding using Beehive versus CPU as a function of the number of application instances.

Apps	1	2	3	4
CPU Energy (mJ/op)	1.1	0.59	0.41	0.32
Beehive Energy (mJ/op)	0.05	0.03	0.02	0.02
Energy efficiency	22×	20×	20×	16×
CPU Goodput (Gbps)	2.0	4.0	6.0	8.0
Beehive Goodput (Gbps)	15	31	45	62
Speedup	7.5×	7.8×	7.5×	7.8×

4.5.5 Reed-Solomon encoding acceleration

To evaluate Beehive's scaling architecture, we evaluate a duplicated Reed-Solomon (RS) encoding accelerator on Beehiveversus a CPU implementation of the same algorithm in Table 4.2. The client sends blocks of 4 KB to the encoder using UDP; the accelerator replies with 1K of erasure data. This could be organized into an (8,2) stripe for double fault tolerance. We measured that one instance of the Reed-Solomon encoder can consume data at 15 Gbps; our FPGA has room for four encoder instances, which consume data at 62 Gbps as shown in Table 4.2. For comparison, we use the open-source Reed-Solomon encoding implementation from BackBlaze [28] running on CPUs which we then duplicate across cores.

We also compare the energy efficiency of the two approaches in Table 4.2. The FPGA is about 20× more efficient per operation than the CPU implementation.

4.5.6 Viewstamped replication witness acceleration

We next turn to a latency-sensitive application, evaluating Beehive hosting a viewstamped replication (VR) witness appliance. We first evaluate the witness on a single shard. We then take

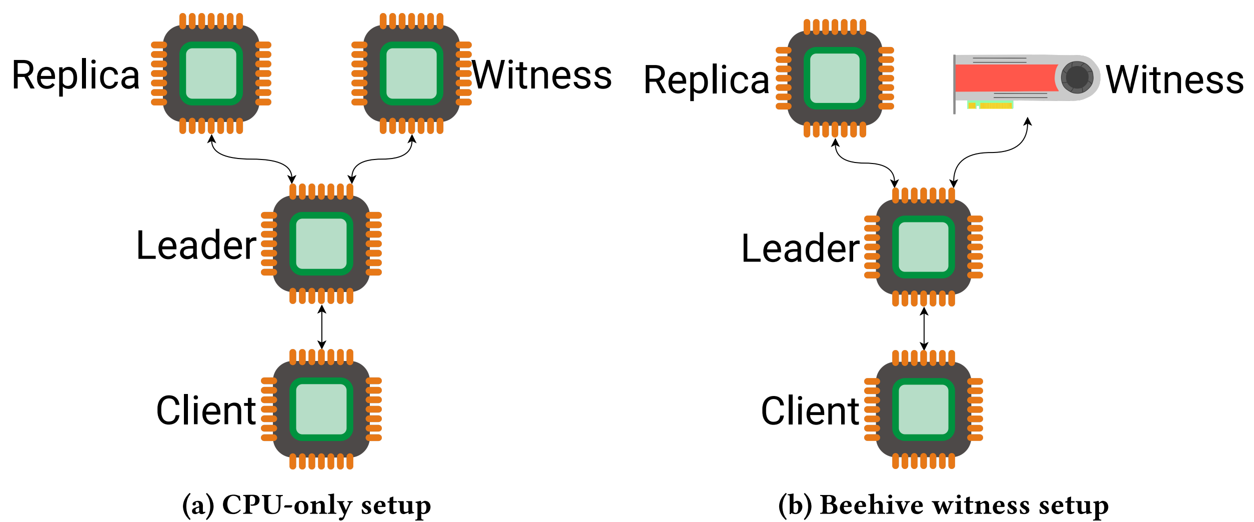


Figure 4.10: Experimental setups for the evaluation of our Viewstamped Replication system

advantage of Beehive's ability to duplicate both internal components and applications to host a 4-shard witness appliance. We also duplicate protocol tiles to prevent them from becoming a bottleneck.

Setup: For all experiments, we evaluate a three-node VR configuration as shown in Figure 4.10, with either the FPGA or CPU serving as a witness. Other nodes are run on CPUs. The CPU VR replicas run on Intel Xeon Gold 5218 CPUs. Client threads run on Intel Xeon Gold 6226R CPUs and are closed loop, i.e., only have one outstanding request at a time. The shard leaders are distributed evenly between two CPU machines. Each shard may handle more than one request at a time. The CPU witness(es) run on a separate server to allow us to measure the energy used by a CPU witness appliance. We use UDP as our transport protocol, because VR does not assume reliable message delivery.

Workload and Metrics: We evaluate our VR accelerator with a replicated key-value store application with 64-byte keys and 64-byte values. The workload uses a read-write mix of 90% reads and 10% writes and a uniform key distribution. Input load is increased by increasing the number of clients. Latency is measured at the clients as the time between the initial request and the even-

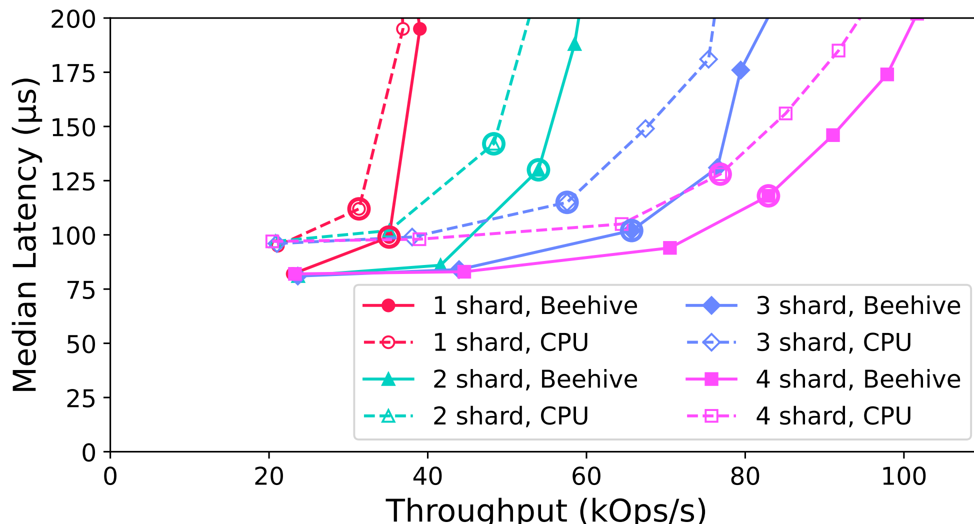


Figure 4.11: Latency versus throughput for the Viewstamped Replication key-value store workload varying the number of shards and client threads. The FPGA witness consistently outperforms the equivalent CPU cores in both latency and throughput.

tual response. Peak throughput numbers are chosen at the points before the latency begins to spike, an indication that the system is overloading and queues in the system are growing. These points correspond to operational setups where increased latency might be considered acceptable in exchange for better throughput [141, 43, 153].

Results: We plot latency versus throughput for differing numbers of shards in Figure 4.11. We increase offered load by increasing the number of client threads sending requests to the leader. The results are shown in Figure 4.11. The system using the FPGA witness can provide up to 1.14× more per-core throughput and up to 1.13× lower median latency.

For each shard, we take the median energy measurement, throughput, median latency, and 99th-percentile (p99) latency at each circled point in Figure 4.11. These results are shown in Table 4.3. The FPGA is between 2.07× and 2.32× more energy efficient per operation compared to the CPU while providing better overall throughput and latency to key-value store clients.

Table 4.3: Energy per operation (measured at the witness) and performance metrics (measured at the clients) at the circled points in Figure 4.11.

Shards	1	2	3	4
CPU Energy (mJ/op)	1.51	1.03	0.90	0.70
BeehiveEnergy (mJ/op)	0.73	0.48	0.39	0.31
Energy efficiency	2.07×	2.16×	2.32×	2.27×
CPU Throughput (kOps/s)	31	48	58	77
BeehiveThroughput (kOps/s)	35	54	66	83
Speedup	1.12×	1.12×	1.14×	1.08×
CPU Median Latency (μs)	112	142	115	128
BeehiveMedian Latency (μs)	99	130	102	118
Improvement	1.13×	1.09×	1.13×	1.08×
CPU p99 Latency (μs)	273	372	339	412
Beehivep99 Latency (μs)	281	334	304	394
Improvement	0.97×	1.11×	1.12×	1.05×

4.5.7 Hardware resource utilization

The hardware utilization of the Beehive infrastructure is shown in Table 4.4. For the UDP stack used in Section 4.5.3, Beehive components use 4% of the LUTs available on the Alveo U200 and 2% of the BRAMs. In a tile, a router uses around 6000 LUTs, twice the size of the UDP processing. For comparison with a more complex module, we include the utilization of the TCP receive path. The router's resource usage does increase slightly, compared to the router in the TCP tile, but the utilization of the actual processing logic in the TCP engine outweighs the utilization of the router. This suggests that Beehive will scale better in designs with more complex processing components. This aligns well with Beehive's goal to support more complex, stateful processing pipelines.

We also compare our resource utilization to that of Limago. We find that our design is larger in terms of LUT usage, but smaller in terms of BRAM usage. Most of our usage comes from the routers rather than the protocol logic, indicating that there is a cost to our increased flexibility.

Table 4.4: FPGA resource utilization of selected modules in Beehive and Limago.

	LUTs (# / % total)	BRAM (# / % total)
Beehive UDP full	58540 / 4.95	41 / 1.90
UDP RX Tile	10054 / 0.85	9.5 / 0.44
Router	5946 / 0.50	0 / 0
NoC Message Parsing	897 / 0.07	0 / 0
UDP RX Processing	2912 / 0.25	9.5 / 0.44
UDP TX Tile	10128 / 0.86	9.5 / 0.44
Router	5955 / 0.50	0 / 0
NoC Message Parsing	658 / 0.06	0 / 0
UDP TX Processing	3105 / 0.26	9.5 / 0.44
Beehive TCP/UDP stack	144491 / 12	84.5 / 4
Beehive TCP Layer	41677 / 3.5	25 / 1.1
TCP RX Processing	10304 / 0.87	9 / 0.4
TCP RX Router	8847 / 0.74	0 / 0
Limago TCP/UDP stack	116948 / 9.9	155 / 7.2
Limago TCP Layer	52134 / 4.4	99 / 4.6

However, in the context of total resources available on the FPGA, the extra logic cost is relatively small.

4.5.8 Flexibility

As a quantitative proxy for flexibility, we count the lines of code (LoC) required to insert an additional instance of an implemented service (network function or application) into the design for our three designs. Results are shown in Table 4.5.

4.5.9 Scalability

We did two experiments to evaluate the scalability of Beehive: one bandwidth-oriented and one hardware resource oriented. For the bandwidth-oriented experiment we repeated the UDP echo

Table 4.5: Lines of code per new tile instantiation in Beehive for end-to-end applications. XML configuration numbers are given as LoC for declaring the tile plus the LoC to add it as a destination.

	Lines of Code	
	XML Config.	Verilog Top Level
Reed-Solomon	25 + 6	13
Viewstamped Replication	18 + (6×# of UDP tiles)	17

experiment in cycle-accurate simulation, duplicating the UDP stack and adding a simple load-balancing tile at the front that splits flows evenly between the stacks. The maximum goodput the load balancer can achieve is 32Gbps for 64-byte UDP packet since each takes 4 cycles to process at the load balancer: 3 for the NoC message and 1 recovery cycle. We hit the maximum possible goodput of the load-balancer of 32Gbps for 64-byte packets. With two stacks, at small packet sizes, we also roughly double the bandwidth as with one stack. This performance difference decreases at larger payload sizes and both stacks converge to the maximum possible goodput of the network link.

To evaluate hardware resource usage scability, we duplicate echo application tiles connected to a UDP stack. On the Alveo U200, we can place 22 application tiles and 28 tiles total. We are limited by timing rather than resource utilization; the critical path is between NoC routers. Each router is fairly expensive, because the 512-bit width of the bus results in a number of high-fanout wires. This is exacerbated by the fact that the FPGA part in the Alveo U200 is made up of several chiplets, and chiplet crossings add significant delay. Several FPGAs [7, 210] now support hardened NoC resources and could improve the quality of results.

4.6 Conclusion

Modern datacenter networking relies on a variety of network functions and protocols, but current hardware network stacks fall short on these features. As datacenters continue to offload computation to accelerators, it is becoming increasingly important to enable direct-attached ac-

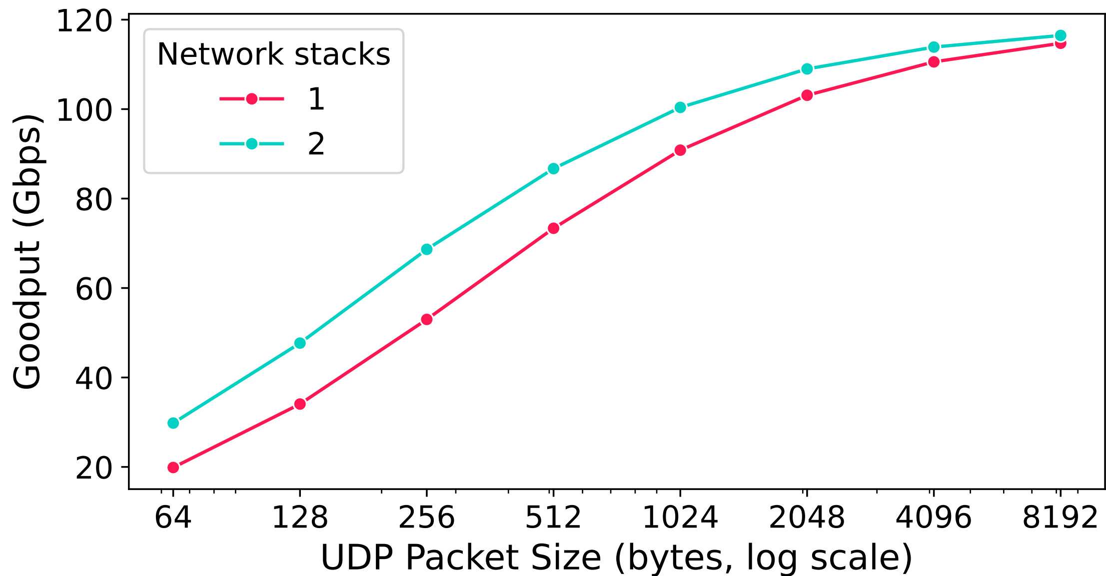


Figure 4.12: Packet size versus goodput for a UDP echo application, running on Beehive with multiple network stacks instantiated. 2 network stacks maxes out the load balancer's throughput.

celerators to reduce network overhead. We presented the design and implementation of Beehive, a NoC-based network stack for direct-attached accelerators that is customizable and supports the variety of protocols and management functions needed for datacenter networking. We demonstrated that Beehive can combine replicated protocol elements and replicated applications for higher bandwidth, provide consistent low latency, with minimal overhead.

Chapter 5

RELATED WORK

Apiary is related to prior work on developing infrastructure for facilitating building and deploying accelerators on FPGAs. There are several different approaches to these goals with.

5.1 Software Frameworks

Xilinx Vitis and OpenCL attempt to create a software-like flow for FPGA development by defining a build framework that extends from hardware to drivers with a focus on easing construction of accelerators. Users are expected to write their accelerators in an HLS language, which is where developers write C++ code with additional annotations. The HLS tool then turns the high-level code into a hardware design. The flow is also responsible for then automatically generating driver or other "library" code for the user to use to interact with their accelerator. The advantage of controlling the whole stack is that it opens up more opportunities for tools to automatically provide features, such as the ability to port between FPGAs.

The downside is that certain types of computation may not be well-suited to the underlying assumptions of the flow. For example, HLS is often less performant than a handwritten HDL implementation [13], making these frameworks less suitable for workloads that do not automatically map well to HLS. These types of frameworks also do not address higher-level service or multiplexing concerns that can arise in multi-accelerator settings.

5.2 FPGA Operating Systems Infrastructure

Another branch of related work focuses on improving the FPGA development experience by providing additional hardware infrastructure that accelerator developers can integrate with. Most related work focuses either on multiplexing the logic fabric, providing portability, or higher-level

services, but not all three simultaneously.

There are several works that look at infrastructure for virtualizing FPGAs to address multiplexing the logic fabric at a low-level. SYNERGY is a framework for FPGA virtualization that uses static analysis in order to identify state that needs to be stored if an accelerator is paused at a given cycle, so it can be resumed seamlessly at a later time. SYNERGY only runs one application on the FPGA at a time, so does not address isolation for inter-accelerator interactions or other OS-level concerns, such as modularity or portability.

AmorphOS [121] is a CPU-hosted FPGA OS that dynamically recompiles accelerators into different configurations, called morphlets. The CPU component may split the FPGA fabric into multiple zones and use partial reconfiguration to multiplex an FPGA between different applications. While AmorphOS does provide a standard shell (they call it a hull) interface to accelerators runnin in its framework to enable portability, it does not provide higher-level services to simplify programming of accelerators. It also does not address the composition or scaling of elements.

Other works have focused on multiplexing specific capabilities of the FPGA. István et al [107] modifies a specific FPGA-accelerated key-value store called Caribou [105], as a case study on how to allow accelerators to support multiple users. The resulting accelerator is able to support multiple connections, but only supports Caribou.

FSRF [128] looks at providing multiplexing and higher-level services for on-FPGA memory for FPGAs attached to a CPU. It provides a fully virtualized, unified address space with the CPU for several on-FPGA accelerators. However, its focus is on memory, and it does not address other services.

Coyote [126] is the most closely related FPGA OS to Apiary and the only one that provides features for multiplexing, portability, and higher-level services, although it is designed for hosted system where an accelerator is attached to a specific CPU process on whose behalf it is acting. Coyote partitions an FPGA into multiple slots and then provides a standard interface to each one that includes higher-level memory, network, and basic interprocess communication services. The memory system is a fully virtualized, unified address space with the CPU. The CPU is responsible for handling page faults on behalf of the FPGA applications. The network stack is a

fully-hardware network stack that implements TCP, although the network stack has limited flexibility. Interprocess communication is done in the form of two queues that connect to "adjacent" accelerators, but Coyote does not discuss any isolation for these channels or in general discuss multi-application interactions in other subsystems.

5.3 FPGA Networking Frameworks

Due to the importance of network processing for direct-attached accelerators, we also survey related works specifically on flexible hardware network processing. Broadly, we group them by the workload they target: packet processing or transport protocol offload.

5.3.1 Packet processing

PANIC [138] is a smartNIC framework that supports integration of arbitrary packet processing elements, including general purpose cores. Unlike Beehive, PANIC targets packet processing rather than full stack support for application accelerators. PANIC uses a similar model to Beehive of chaining message-passing elements over a NoC, but it relies on a crossbar and centralized scheduler, limiting scalability. While PANIC does not directly address deadlock, its centralized scheduler is responsible for dropping packets when it runs out of buffer space. Dropping packets to free up buffer space means that packets can always drain out of the NoC into a buffer, preventing deadlock. However, this dropping behavior makes integrating RPC/TCP applications into PANIC challenging, because the central scheduler assumes that operations occur on a packet level. RPC protocols may have multi-packet messages, and TCP has to perform stream reconstruction, which means accelerators for these protocols must look across packets. However, the centralized scheduler may drop any packet at any time. If it drops an acknowledged packet it would violate TCP semantics.

ClickNP [134] is an FPGA-accelerated packet processing framework that also supports the integration of arbitrary processing elements. It does not use a NoC. Instead, components are directly connected via FIFOs. The lack of structured interconnect makes it harder to replicate elements. Since ClickNP aims to accelerate software network functions, it also lacks support

for higher-level network protocols and direct-attached accelerators. It further assumes a PCIe connection to a CPU, which it relies on for control-plane configuration.

Rosebud [122] is an FPGA framework for middleboxes. It uses an interconnect to connect custom processing elements they call reconfigurable processing units (RPUs) which can include accelerators. Because it targets middleboxes, they do not evaluate a network stack with full reliable transport protocol support. While it does provide support to chain RPUs, they acknowledge it was not designed to do so, and as a result, inter-RPU traffic has a fairly significant latency penalty.

There are also more restrictive approaches to packet processing. A group of works leverage processing architectures based on reconfigurable match-action tables where an action (e.g. strip a header, rewrite a field, drop a packet) is taken based on some header fields in the header of the packet. These tables can be rewritten to change what is matched and what is the resultant action. Typically, there is a pipeline of these processing elements [74, 118, 38]. However, match-action style processing is not well-suited for highly stateful processing [167] typical of application-level offloads. Other models have been proposed for stateful packet processing. Flowblaze uses an FSM-based model [167]. However, they specifically say that workloads above the transport layer are out of scope. hXDP proposed a processor for eBPF bytecode [40] designed for offloading kernel-level eBPF programs. Because of its sequential execution model, hXDP performs best on small programs and is a poor fit for more complex processing such as Reed-Solomon encoding.

5.3.2 Transport protocol offloads

Another related vein of work are transport protocol offloads. Most of these are TCP offload engines available as custom chips [44] or encrypted IP cores for FPGAs [207, 154, 62]. They generally do not support the full range of functions found in datacenter network stacks.

Some TCP offload engines could potentially support modification. Limago [178] is an open-source TCP and RoCEv2 offload engine written in Vivado HLS. However, it does not provide any specific APIs or hooks for adding other protocols, so introducing a new network function or new protocol would require fairly extensive modifications to the stack itself. Tonic [25] is an

open-source implementation of the TCP send path and supports customization of the transport protocol, but does not address any lower-level packet processing layers; it also lacks a complete receive path implementation.

FlexTOE [182] is a software implementation of TCP offload engine using the Netronome DPU, a processor designed specifically for network processing that is programmable using C or eBPF. While they do support network functions, their work targets TCP offload for CPUs while our work shows that a direct-attached hardware accelerator does not need a CPU core to support software stack functionality.

Microsoft Catapult's FPGAs use a custom transport protocol called LTL [41], which is a reliable transport protocol over UDP. Similar to most TCP engines, it is presented as a fixed IP core with no interface for extension. Catapult also supports a single-layer RMT, used for network virtualization [74]. However, it is unknown if these are ever combined and if so, how it would support new protocols or network functions.

Chapter 6

CONCLUSION

As datacenters continue to integrate application accelerators for improved compute efficiency, the design of system infrastructure for these accelerators becomes increasingly important. Current infrastructure is low-level and does not always provide helpful abstractions to raise the productivity of the designers of accelerated systems. At the same time, it is important to carefully architect common infrastructure such that it can be reused across multiple accelerators without sacrificing efficiency gains.

This thesis argues that it is possible to architect modular direct-attached accelerator systems that provide operating system services and abstractions without sacrificing efficiency. We propose Apiary, a microkernel-like FPGA OS based on message-passing over a NoC. We describe the design of primitives for message passing, isolation, context swapping, cooperative scheduling, and networking. We then implement the message passing, memory management, isolation, and integrate it into the network stack. We evaluate how well the combination of services satisfies Apiary's design goals. We then describe Beehive, Apiary's network stack. Beehive is also a NoC-based architecture. Beehive demonstrates how Apiary's message passing design philosophy enables flexible composition

6.1 Future work

This work serves as a specific example of how to and lays the foundation for further exploration on infrastructure for direct-attached accelerators. The most immediate set of future work is implementing and evaluating the other proposed primitives for Apiary as well as adding additional features to Beehive such as expanding the context management subsystem or integrating some of its current network functions with preexisting systems. For example, Beehive's current IP en-

capsulation uses a custom protocol to update virtual IP addresses. In deployment, Beehivewould need to provide a service for interfacing with the datacenter's control plane service that manages the virtual networking. However, are other design decisions or abstractions that could and should be explored given the potential of direct-attached accelerators and the many open questions in the area, such as enabling partial reconfiguration, full fault isolation, or handling malicious accelerators. Apiary is designed to be modular in order to enable this sort of exploration and different service implementations should be easily swapped in or out to evaluate them and their tradeoffs.

Future work in order to bring the Apiary development experience closer to that of a software OS would require additional compile-time or runtime tooling. At compile-time, for example, Beehive's deadlock-checking tool should be expanded to support message analysis in Apiary. At runtime, for example, FPGAs in general need tooling to support user provisioning within the datacenter with additional augmentation to properly configure Apiary, such as the virtual networking integration with Beehive mentioned above.

Enabling direct-attached accelerators also opens up areas of research at the datacenter level, such as on disaggregated systems. Beehive's ability to support complex, software-like network stacks makes it easier to integrate it into preexisting networked systems rather than developing an entire bespoke stack. Furthermore, direct-attached accelerators can be allocated without reliance on a hosting CPU which could open interesting resource management questions for disaggregated systems.

BIBLIOGRAPHY

[1] Apache Thrift. https://thrift.apache.org. Accessed: 2022-6-28.

[2] Controlling Access to the Kubernetes API. https://kubernetes.io/docs/concepts/security/controlling-access/. Accessed: 2022-6-28.

[3] gRPC. https://grpc.io. Accessed: 2022-6-28.

[4] Identity-Aware and HTTP-Aware Policy Enforcement. https://docs.cilium.io/en/v1.11/gettingstarted/http/. Accessed: 2022-6-28.

[5] Intel FPGA Device Plugin for Kubernetes. https://intel.github.io/intel-device-plugins-for-kubernetes/cmd/fpga_plugin/README.html. Accessed: 2022-6-28.

[6] NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect. Accessed: 2022-6-28.

[7] Achronix. Revolutionary New 2D Network-on-Chip. https://www.achronix.com/revolutionary-new-2d-network-chip. Accessed: 2022-7-4.

[8] Aditya Akella, Amin Vahdat, Arjun Singhvi, Behnam Montazeri, Dan Gibson, Hassan Wassel, Joel Scherpelz, Milo M. K. Martin, Monica C Wong-Chan, Moray Mclaren, Prashant Chandra, Rob Cauble, Sean Clark, Simon Sabato, and Thomas F. Wenisch. 1RMA: Re-Envisioning Remote Memory Access for Multi-Tenant Datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, page 708–721, New York, NY, USA, 2020.

[9] Algo-Logic. Ultra-low-latency 40g tcp endpoint. http://algo-logic.com/sites/default/files/40G_TCP_Endpoint_0.pdf. Accessed: 2019-10-24.

[10] Alibaba. Deep Dive into Alibaba Cloud F3 FPGA as a Service Instances. https://www.alibabacloud.com/blog/deep-dive-into-alibaba-cloud-f3-fpga-as-a-service-instances_594057. Accessed: 2020-01-29.

[11] Alibaba. How does alibaba cloud build high-performance cloud-native pod networks in production environments? https://www.alibabacloud.com/blog/how-does-alibaba-cloud-build-high-performance-cloud-native-pod-networks-in-production-environments_596590, 2020. Accessed: 2021-9-27.

[12] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 conference*, pages 63–74, 2010.

[13] Wesson Altoyan and Juan J. Alonso. Investigating performance losses in high-level synthesis for stencil computations. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 195–203, 2020.

[14] Amazon. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/. Accessed: 2019-10-11.

[15] Amazon. EC2 P3 Instances. https://aws.amazon.com/ec2/instance-types/p3/. Accessed: 2020-02-06.

[16] Amazon. AQUA (Advanced Query Accelerator). https://aws.amazon.com/blogs/aws/new-aqua-advanced-query-accelerator-for-amazon-redshift/, 2021. Accessed: 2025-1-4.

[17] Amazon. Deep dive on amazon ec2 vt1 instances. https://aws.amazon.com/blogs/compute/deep-dive-on-amazon-ec2-vt1-instances/, 2021. Accessed: 2025-1-5.

[18] Amazon. The Components of the Nitro System. https://docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/the-components-of-the-nitro-system.html, 2025. Accessed: 2025-1-4.

[19] AMD. Alveo V80 Product Brief. https://www.amd.com/content/dam/amd/en/documents/products/accelerators/alveo/v80/alveo-v80-product-brief.pdf. Accessed: 2025-1-5.

[20] AMD. AMD Powers Alibaba Cloud FaaSwith AI Acceleration Solution for E-Commerce Business. https://www.amd.com/content/dam/amd/en/documents/resources/case-studies/alibaba-case-study.pdf, 2023. Accessed: 2025-1-5.

[21] AMD. AMD Provides Twitch with Plug and Play VP9 Transcoding Solution for Live Video Streaming. https://www.amd.com/content/dam/amd/en/documents/resources/case-studies/twitch-case-study.pdf, 2023. Accessed: 2025-1-5.

[22] AMD. SmartSSD Computational Storage Drive. https://www.xilinx.com/publications/product-briefs/xilinx-smartssd-computational-storage-drive-product-brief.pdf, 2023. Accessed: 2025-1-5.

[23] AMD. Versal Architecture and Product Data Sheet: Overview. https://docs.amd.com/v/u/en-US/ds950-versal-overview, 2024. Accessed: 2025-1-6.

[24] KV Anjan and Timothy Mark Pinkston. An efficient, fully adaptive deadlock recovery scheme: DISHA. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 201–210, 1995.

[25] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling Programmable Transport Protocols in High-Speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, February 2020.

[26] ARM. AMBA 4 AXI4-Stream Protocol Specification. https://developer.arm.com/documentation/ihi0051/a/Introduction/About-the-AXI4-Stream-protocol. Accessed: 2022-6-28.

[27] InfiniBand Trade Association. Infiniband architecture specification. 2000.

[28] Backblaze. JavaReedSolomon. https://github.com/Backblaze/JavaReedSolomon. accessed: 2023-11-28.

[29] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering Azure Storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, Boston, MA, April 2023. USENIX Association.

[30] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. OpenPiton: An Open Source Manycore Research Framework. In

Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, page 217–232, New York, NY, USA, 2016. Association for Computing Machinery.

[31] Jeffrey R Ballard, Ian Rae, and Aditya Akella. Extensible and Scalable Network Monitoring Using OpenSAFE. *INM/WREN*, 10, 2010.

[32] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. The datacenter as a computer: Designing warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 13(3):i–189, 2018.

[33] Suhail Basalama, Atefeh Sohrabizadeh, Jie Wang, Licheng Guo, and Jason Cong. Flexcnn: An end-to-end framework for composing cnn accelerators on fpga. *ACM Trans. Reconfigurable Technol. Syst.*, 16(2), March 2023.

[34] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 29–44. Association for Computing Machinery, 2009.

[35] Brian N Bershad, Thomas E Anderson, Edward D Lazowska, and Henry M Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(2):175–198, 1991.

[36] Steve Bishop, Matthew Fairbairn, Hannes Mehnert, Michael Norrish, Tom Ridge, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with Logic: Rigorous Test-Oracle Specification and Validation for TCP/IP and the Sockets API. *J. ACM*, 66(1), dec 2018.

[37] Ethan Blanton, Dr. Vern Paxson, and Mark Allman. TCP Congestion Control. RFC 5681, September 2009.

[38] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, aug 2013.

[39] Andrew Boutros, Mathew Hall, Nicolas Papernot, and Vaughn Betz. Neighbors from hell: Voltage attacks against deep learning accelerators on multi-tenant fpgas. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 103–111, 2020.

[40] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient Software Packet Processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990. USENIX Association, November 2020.

[41] Adrian Caulfield, Eric Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A Cloud-Scale Acceleration Architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–13. IEEE Computer Society, October 2016.

[42] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live - An Engineering Perspective (2006 Invited Talk). In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, 2007.

[43] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 407–418, 2003.

[44] Terminator 6 ASIC. https://www.chelsio.com/terminator-6-asic/. Accessed: 2019-10-24.

[45] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload Control for u-scale RPCs with Breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 299–314. USENIX Association, November 2020.

[46] Jongsok Choi, Ruolong Lian, Zhi Li, Andrew Canis, and Jason Anderson. Accelerating memcached on AWS cloud FPGAs. In *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, page 2. ACM, 2018.

[47] Cilium. L7 Load Balancing and URL re-writing. https://docs.cilium.io/en/latest/gettingstarted/servicemesh/envoy-traffic-management/. Accessed: 2022-6-28.

[48] Cloudflare. What is mutual TLS (mTLS)? https://www.cloudflare.com/learning/access-management/what-is-mutual-tls/. Accessed: 2022-6-28.

[49] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *Proceedings of the ACM Symposium*

on Principles of Distributed Computing, PODC '17, page 73–82, New York, NY, USA, 2017. Association for Computing Machinery.

[50] Robert C. Daley and Jack B. Dennis. Virtual memory, processes, and sharing in multics. In *Proceedings of the First ACM Symposium on Operating System Principles*, SOSP '67, page 12.1–12.8, New York, NY, USA, 1967. Association for Computing Machinery.

[51] Dally and Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5):547–553, 1987.

[52] William J. Dally and Brian Towles. Route Packets, Not Wires: On-Chip Inteconnection Networks. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, page 684–689, New York, NY, USA, 2001. Association for Computing Machinery.

[53] William James Dally and Brian Patrick Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[54] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 373–387, Renton, WA, April 2018. USENIX Association.

[55] Alpha Data. ADM-PCIE-9V3 - High-Performance Network Accelerator. https://www.alpha-data.com/pdfs/adm-pcie-9v3.pdf. Accessed: 2024-4-16.

[56] David Patterson. Domain Specific Architectures for Deep Neural Networks: Three Generations of Tensor Processing Units (TPUs), 2019. Allen School Distinguished Lecture, https://www.youtube.com/watch?v=VCScWh966u4.

[57] Debian. Hugepages. https://wiki.debian.org/Hugepages, 2023. Accessed: 2025-3-7.

[58] Anant Deepak. eBPF / XDP Firewall and Packet Filtering. http://vger.kernel.org/lpc_net2018_talks/ebpf-firewall-LPC.pdf. Accessed: 2021-10-26.

[59] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, March 1966.

[60] Colin Drewes, Olivia Weng, Keegan Ryan, Bill Hunter, Christopher McCarty, Ryan Kastner, and Dustin Richmond. Turn on, tune in, listen up: Maximizing side-channel recovery in time-to-digital converters. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '23, page 111–122, New York, NY, USA, 2023. Association for Computing Machinery.

[61] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless computing on heterogeneous computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 797–813, New York, NY, USA, 2022. Association for Computing Machinery.

[62] TCP Offload Engine – Offloading TCP into Hardware. https://www.easics.com/tcp-offload-engine/. Accessed: 2022-6-28.

[63] eBPF. What is ebpf? https://cloudblogs.microsoft.com/opensource/2021/05/10/making-ebpf-work-on-windows/. Accessed: 2022-06-30.

[64] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, Santa Clara, CA, 2016.

[65] Envoy. Life of a Request. https://www.envoyproxy.io/docs/envoy/latest/intro/life_of_a_request. Accessed: 2021-10-26.

[66] Envoy Proxy. https://www.envoyproxy.io/, 2022.

[67] Haggai Eran, Maxim Fudim, Gabi Malka, Gal Shalom, Noam Cohen, Amit Hermony, Dotan Levi, Liran Liss, and Mark Silberstein. FlexDriver: A Network Driver for Your Accelerator. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 1115–1129, New York, NY, USA, 2022. Association for Computing Machinery.

[68] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An infrastructure for inline acceleration of network applications. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 345–362, Renton, WA, July 2019. USENIX Association.

[69] European Parliament and Council of the European Union. Regulation (EU) 2016/679 of the European Parliament and of the Council, 2016.

[70] Mazen Ezzeddine, Raghid Morcel, Hassan Artail, Mazen AR Saghir, Haitham Akkary, and Hazem Hajj. Restful hardware microservices using reconfigurable networked accelerators in cloud and edge datacenters. In *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*, pages 1–4. IEEE, 2018.

[71] Arthur Fabre. L4Drop: XDP DDoS Mitigations. https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations/. Accessed: 2021-10-26.

[72] Facebook. Accelerating Facebook's Infrastructure with Application-Specific Hardware. https://engineering.fb.com/data-center-engineering/accelerating-infrastructure/. Accessed: 2019-10-11.

[73] Daniel Firestone. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 315–328, Boston, MA, March 2017. USENIX Association.

[74] Daniel Firestone, Andrew Putnam, Hari Angepat, Derek Chiou, Adrian Caulfield, Eric Chung, Matt Humphrey, Kalin Ovtcharov, Jitu Padhye, Doug Burger, Dave Maltz, Albert Greenberg, Sambhrama Mundkur, Alireza Dabagh, Mike Andrewartha, Vivek Bhanu, Harish Kumar Chandrappa, Somesh Chaturmohta, Jack Lavier, Norman Lam, Fengfen Liu, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Kushagra Vaid, and David A. Maltz. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2018.

[75] Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. Corundum: An Open-Source 100-Gbps NIC. In *28th IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2020.

[76] FOSSi Foundation. cocotb. https://www.cocotb.org. Accessed: 2025-9-27.

[77] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steve Reinhardt, Adrian Caulfield, Eric Chung, and Doug Burger. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *Proceedings of the 45th International Symposium on Computer Architecture, 2018*. ACM, June 2018.

[78] ETH Zurich FPGA @ Systems Group. EasyNet: 100 Gbps TCP/IP Network Stack for HLS. https://github.com/fpgasystems/Vitis_with_100Gbps_TCP-IP. Accessed: 2024-6-20.

[79] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 249–264, 2016.

[80] Joan Farres Garcia. "falafel". https://github.com/jfarresg/falafel-private. Accessed: 2025-9-25.

[81] HiTech Global. 10G TCP/IP Offload Engine (TOE) IP Core. http://www.hitechglobal.com/IPCores/10Gi-TOE.htm. Accessed: 2019-10-24.

[82] Google. Cloud GPUs. https://cloud.google.com/gpu. Accessed: 2020-02-06.

[83] Google. Google's scalable supercomputers for machine learning, Cloud TPU Pods, are now publicly available in beta. Accessed: 2019-10-11.

[84] Google. The Next Wave of Google Cloud Infrastructure Innovation: New C3 VM and Hyperdisk. https://cloud.google.com/blog/products/compute/introducing-c3-machines-with-googles-custom-intel-ipu. Accessed: 2025-1-4.

[85] Google. Using FPGA On YARN. https://hadoop.apache.org/docs/r3.1.0/hadoop-yarn/hadoop-yarn-site/UsingFPGA.html. Accessed: 2019-10-11.

[86] Google. New gke dataplane v2 increases security and visibility for containers. https://cloud.google.com/blog/products/containers-kubernetes/bringing-ebpf-and-cilium-to-google-kubernetes-engine, 2020. Accessed: 2021-9-27.

[87] K. Goossens, J. Dielissen, and A. Radulescu. Aethereal network on chip: concepts, architectures, and implementations. *IEEE Design & Test of Computers*, 22(5):414–421, 2005.

[88] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E Anderson. Backpressure Flow Control. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 779–805, 2022.

[89] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, page 51–62, New York, NY, USA, 2009. Association for Computing Machinery.

[90] Licheng Guo, Yuze Chi, Jason Lau, Linghao Song, Xingyu Tian, Moazin Khatti, Weikang Qiao, Jie Wang, Ecenur Ustun, Zhenman Fang, Zhiru Zhang, and Jason Cong. Tapa: A scalable task-parallel dataflow programming framework for modern fpgas with co-optimization of hls and physical design. *ACM Trans. Reconfigurable Technol. Syst.*, 16(4), December 2023.

[91] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A Hardware-Software Co-Designed Disaggregated Memory System. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 417–433, New York, NY, USA, 2022. Association for Computing Machinery.

[92] Roni Haecki, Radhika Niranjan Mysore, Lalith Suresh, Gerd Zellweger, Bo Gan, Timothy Merrifield, Sujata Banerjee, and Timothy Roscoe. How to diagnose nanosecond network latencies in rich end-host stacks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 861–877, Renton, WA, April 2022. USENIX Association.

[93] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. Desc: Decoupled supply-compute communication management for heterogeneous architectures. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 191–203, 2015.

[94] Yutaro Hayakawa, Michio Honda, Douglas Santry, and Lars Eggert. Prism: Proxies without the Pain. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 535–549. USENIX Association, April 2021.

[95] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure Coding in Windows Azure Storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 15–26, Boston, MA, June 2012. USENIX Association.

[96] Muhuan Huang, Di Wu, Cody Hao Yu, Zhenman Fang, Matteo Interlandi, Tyson Condie, and Jason Cong. Programming and runtime support to blaze fpga accelerator deployment at datacenter scale. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 456–469, New York, NY, USA, 2016. ACM.

[97] Huawei. FPGA Development Suite. https://github.com/huaweicloud/huaweicloud-fpga. Accessed: 2025-1-5.

[98] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, June 2010.

[99] IETF. Generic routing encapsulation (gre). https://tools.ietf.org/html/rfc1701. Accessed: 2020-01-31.

[100] Intel. Data Plane Development Kit (DPDK). https://www.dpdk.org/. Accessed: 2020-02-05.

[101] Intel. Intel Agilex 7 M-Series Hard Memory NoC Subsystem. https://www.intel.com/content/www/us/en/docs/programmable/773264/23-2-1-3-0/m-series-hard-memory-noc-subsystem.html. Accessed: 2025-1-9.

[102] Intel. Key Features and Innovations in Agilex 7 FPGAs and SoCs. https://www.intel.com/content/www/us/en/docs/programmable/683458/current/key-features-and-innovations-in-fpgas.html, 2024. Accessed: 2025-1-5.

[103] Intilop. 40G Bit TCP Offload Engine (TOE). https://intilop.com/resources/product_briefs/40G_1K-Sess_TCP+UDP_Offload+MAC+Host_IFUltra-LowLatency(INT-40011).pdf. Accessed: 2019-10-24.

[104] Istio. Traffic Management. https://istio.io/latest/docs/concepts/traffic-management/. Accessed: 2022-6-28.

[105] Zsolt István, David Sidler, and Gustavo Alonso. Caribou: Intelligent Distributed Storage. *Proc. VLDB Endow.*, 10(11):1202–1213, August 2017.

[106] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 425–438, 2016.

[107] Zsolt István, Gustavo Alonso, and Ankit Singla. Providing multi-tenant services with fpgas: Case study on a key-value store. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 119–1195, 2018.

[108] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, 2014. USENIX Association.

[109] N.E. Jerger, T. Krishna, L.S. Peh, and M. Martonosi. *On-Chip Networks: Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2017.

[110] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: a berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[111] Prajakta Joshi. Building Scalable Private Services with Internal Load Balancing. https://cloud.google.com/blog/products/gcp/building-scalable-private-services-with-internal-load-balancing. Accessed: 2022-6-28.

[112] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, and et al. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Computer Architecture News*, 45(2):1–12, June 2017.

[113] Theo Julienne. GLB: GitHub's open source load balancer. https://github.blog/2018-08-08-glb-director-open-source-load-balancer/. accessed: 2022-6-28.

[114] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. Bluedbm: An appliance for big data analytics. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13, New York, NY, USA, 2015. ACM.

[115] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, Mansoor Mohsin, Ray Kong, Anmol Ahuja, Oana Platon, Alex Wun, Matthew Snider, Chacko Daniel, Dan Mastrian, Yang Li, Aprameya Rao, Vaishnav Kidambi, Randy Wang, Abhishek Ram, Sumukh Shivaprakash, Rajeet Nair, Alan Warwick, Bharat S. Narasimman, Meng Lin, Jeffrey Chen, Abhay Balkrishna Mhatre, Preetha Subbarayalu, Mert Coskun, and Indranil Gupta. Service Fabric: A Distributed Platform for Building Microservices in the Cloud. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[116] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–16, Boston, MA, February 2019. USENIX Association.

[117] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided ({RDMA}) Datagram RPCs. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 185–201, 2016.

[118] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 67–81, New York, NY, USA, 2016. Association for Computing Machinery.

[119] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. Tas: Tcp acceleration as an os service. In *Proceedings of the 14th EuroSys Conference*, pages 24:1–24:16, New York, NY, USA, 2019. ACM.

[120] Osama Khan, Randal C Burns, James S Plank, William Pierce, and Cheng Huang. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, San Jose, CA, February 2012. USENIX Association.

[121] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 107–127, Carlsbad, CA, October 2018. USENIX Association.

[122] Moein Khazraee, Alex Forencich, George C. Papen, Alex C. Snoeren, and Aaron Schulman. Rosebud: Making FPGA-Accelerated Middlebox Development More Pleasant. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 586–605, New York, NY, USA, 2023. Association for Computing Machinery.

[123] Moein Khazraee, Lu Zhang, Luis Vega, and Michael Bedford Taylor. Moonwalk: NRE Optimization in ASIC Clouds. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 511–526, New York, NY, USA, 2017. ACM.

[124] Hyojoon Kim and Nick Feamster. Improving network management with software defined networking. *IEEE Communications Magazine*, 51(2):114–119, 2013.

[125] Teemu Koponen, Keith Amidon, Peter Balland, Martin Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network Virtualization in Multi-tenant Datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 203–216, Seattle, WA, April 2014. USENIX Association.

[126] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. Do OS abstractions make sense on FPGAs? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 991–1010. USENIX Association, November 2020.

[127] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the ACM SIGCOMM 2020 Conference*, pages 514–528, 2020.

[128] Joshua Landgraf, Matthew Giordano, Esther Yoon, and Christopher J. Rossbach. Reconfigurable virtual memory for fpga-driven i/o. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 556–571, New York, NY, USA, 2023. Association for Computing Machinery.

[129] Joshua Landgraf, Tiffany Yang, Will Lin, Christopher J. Rossbach, and Eric Schkufza. Compiler-driven fpga virtualization with synergy. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 818–831, New York, NY, USA, 2021. Association for Computing Machinery.

[130] Andreas Lankes, Thomas Wild, Andreas Herkersdorf, Soeren Sonntag, and Helmut Reinig. Comparison of Deadlock Recovery and Avoidance Mechanisms to Approach Message Dependent Deadlocks in On-chip Networks. In *2010 Fourth ACM/IEEE International Symposium on Networks-on-Chip*, pages 17–24, 2010.

[131] Maysam Lavasani, Hari Angepat, and Derek Chiou. An fpga-based in-line accelerator for memcached. *IEEE Computer Architecture Letters*, 13(2):57–60, 2014.

[132] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: Efficient and Fast RPCs in Cloud Microservices with near-Memory Reconfigurable NICs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 36–51, New York, NY, USA, 2021. Association for Computing Machinery.

[133] LeWiz. Talon 4220 datasheet. http://www.lewiz.com/products/product_datasheets/talon4220_ds.pdf. Accessed: 2019-10-24.

[134] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 1–14, New York, NY, USA, 2016. Association for Computing Machinery.

[135] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control. In *Proceedings of the ACM SIGCOMM 2019 Conference*, page 44–58, 2019.

[136] Katie Lim and Jonathan Balkind. Accelerator interfacing is like an onion. In *2025 Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE)*, 2025.

[137] Jennifer Lin, Megan O'Keefe, Samrat Ray, Sandeep Parikh, and Eimear Hennessy. The Service Mesh Era: Architecting, Securing and Managing Microservices with Istio. Whitepaper, Google, 2019. Accessed: 2022-6-28.

[138] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 243–259. USENIX Association, November 2020.

[139] Kernel TLS operation. https://docs.kernel.org/networking/tls-offload.html. Accessed: 2022-6-28.

[140] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.

[141] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. Performance-Optimal Read-Only Transactions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 333–349, 2020.

[142] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. A Hypervisor for Shared-Memory FPGA Platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 827–844, New York, NY, USA, 2020. Association for Computing Machinery.

[143] Ikuo Magaki, Moein Khazraee, Luis Vega, and Michael Taylor. ASIC Clouds: Specializing the Datacenter. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[144] Mallik Mahalingam, Dinesh Dutt, Kenneth Duda, Puneet Agarwal, Larry Kreeger, T. Sridhar, Mike Bursell, and Chris Wright. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. RFC 7348, August 2014.

[145] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a Microkernel Approach to Host Networking. In *In ACM SIGOPS 27th Symposium on Operating Systems Principles*, New York, NY, USA, 2019.

[146] Joel Mandebi Mbongue, Danielle Tchuinkou Kwadjo, Alex Shuping, and Christophe Bobda. Deploying Multi-Tenant FPGAs within Linux-Based Cloud Infrastructure. *ACM Trans. Reconfigurable Technol. Syst.*, 15(2), dec 2021.

[147] Mellanox. ConnectX Ethernet Adapters. https://www.mellanox.com/products/connectx-smartnic. Accessed: 2019-10-24.

[148] Mellanox. Mellanox. 2020. connectx®-6 dx en card product brief. https://www.mellanox.com/files/doc-2020/pb-connectx-6-dx-en-card.pdf. Accessed: 2021-10-28.

[149] Mellanox. Mellanox Bluefield Overview. https://www.mellanox.com/products/bluefield-overview. Accessed: 2020-01-29.

[150] Mellanox. Mellanox Innova-2 Flex Open Programmable SmartNIC Product Brief. https://www.mellanox.com/sites/default/files/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf. Accessed: 2020-1-30.

[151] Microsoft. Create security policies with extended port access control lists. https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v-virtual-switch/create-security-policies-with-extended-port-access-control-lists. Accessed: 2022-6-28.

[152] Microsoft. FPGA Web Service: Deploy Models on FPGAs. https://docs.microsoft.com/en-us/azure/machine-learning/how-to-deploy-fpga-web-service. Accessed: 2020-01-29.

[153] David L Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on communications*, 39(10):1482–1493, 1991.

[154] TCP/UDP/IP Network Protocol Accelerator Platform (NPAP). https://www.missinglinkelectronics.com/index.php/menu-products/menu-network-protocol-accelerator. Accessed: 2022-6-28.

[155] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, page 221–235, New York, NY, USA, 2018. ACM.

[156] Srinivasan Murali, Paolo Meloni, Federico Angiolini, David Atienza, Salvatore Carta, Luca Benini, Giovanni De Micheli, and Luigi Raffo. Designing Message-Dependent Deadlock Free Networks on Chips for Application-Specific Systems on Chips. In *2006 IFIP International Conference on Very Large Scale Integration*, pages 158–163, 2006.

[157] Network Research @ UW Madison. PANIC. https://bitbucket.org/uw-madison-networking-research/panic_osdi20_artifact/src/master/. accessed: 2022-6-28.

[158] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 327–341, New York, NY, USA, 2018. Association for Computing Machinery.

[159] NVIDIA. NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect. Accessed: 2025-1-4.

[160] NVIDIA. NVIDIA NVSwitch Technical Overview. http://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf. Accessed: 2019-10-24.

[161] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.

[162] OpenAI. AI and Compute. https://openai.com/blog/ai-and-compute/. Accessed: 2020-1-17.

[163] Jin Ouyang and Yuan Xie. Loft: A high performance network-on-chip providing quality-of-service support. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 409–420, 2010.

[164] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud Scale Load Balancing. *SIGCOMM Comput. Commun. Rev.*, 43(4):207–218, aug 2013.

[165] Charles E. Perkins. IP Encapsulation within IP. RFC 2003, October 1996.

[166] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. Autonomous NIC Offloads. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 18–35, New York, NY, USA, 2021. Association for Computing Machinery.

[167] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. FlowBlaze: Stateful Packet Processing in Hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, Boston, MA, February 2019. USENIX Association.

[168] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[169] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. Optimus Prime: Accelerating Data Transformation in Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1203–1216, New York, NY, USA, 2020. Association for Computing Machinery.

[170] Arash Pourhabibi, Mark Sutherland, Alexandros Daglis, and Babak Falsafi. Cerebros: Evading the RPC Tax in Datacenters. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 407–420, New York, NY, USA, 2021. Association for Computing Machinery.

[171] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Gopi Prashanth Gopal, and Simon Pope. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, pages 13–24. IEEE, June 2014.

[172] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, Anna Cheung, In Suk Chong, Niranjani Dasharathi, Jia Feng, Brian Fosco, Samuel Foss, Ben Gelb, Sara J. Gwin, Yoshiaki Hase, Da-ke He, C. Richard Ho, Roy W. Huffman Jr., Elisha Indupalli, Indira Jayaram, Poonacha Kongetira, Cho Mon Kyaw, Aaron Laursen, Yuan Li, Fong Lou, Kyle A. Lucke, JP Maaninen, Ramon Macias, Maire Mahony, David Alexander Munday, Srikanth Muroor, Narayana Penukonda, Eric Perkins-Argueta,

Devin Persaud, Alex Ramirez, Ville-Mikko Rautio, Yolanda Ripley, Amir Salek, Sathish Sekar, Sergey N. Sokolov, Rob Springer, Don Stark, Mercedes Tan, Mark S. Wachsler, Andrew C. Walton, David A. Wickeraad, Alvin Wijaya, and Hon Kwan Wu. Warehouse-scale video acceleration: co-design and deployment in the wild. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 600–615, New York, NY, USA, 2021. Association for Computing Machinery.

[173] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, Anna Cheung, In Suk Chong, Niranjani Dasharathi, Jia Feng, Brian Fosco, Samuel Foss, Ben Gelb, Sara J. Gwin, Yoshiaki Hase, Da-ke He, C. Richard Ho, Roy W. Huffman Jr., Elisha Indupalli, Indira Jayaram, Poonacha Kongetira, Cho Mon Kyaw, Aaron Laursen, Yuan Li, Fong Lou, Kyle A. Lucke, JP Maaninen, Ramon Macias, Maire Mahony, David Alexander Munday, Srikanth Muroor, Narayana Penukonda, Eric Perkins-Argueta, Devin Persaud, Alex Ramirez, Ville-Mikko Rautio, Yolanda Ripley, Amir Salek, Sathish Sekar, Sergey N. Sokolov, Rob Springer, Don Stark, Mercedes Tan, Mark S. Wachsler, Andrew C. Walton, David A. Wickeraad, Alvin Wijaya, and Hon Kwan Wu. Warehouse-Scale Video Acceleration: Co-Design and Deployment in the Wild. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 600–615, New York, NY, USA, 2021. Association for Computing Machinery.

[174] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster. In *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 13)*, San Jose, CA, June 2013. USENIX Association.

[175] B. Ringlein, F. Abel, , D. Diamantopoulos, B. Weiss, C. Hagleitner, M. Reichenbach, and D. Fey. A Case for Function-as-a-Service with Disaggregated FPGAs. In *Proceedings of the 2021 IEEE 14th International Conference on Cloud Computing (CLOUD 2021)*, pages 333–344, Virtual Conference, September 2021. IEEE.

[176] Burkhard Ringlein, Francois Abel, Alexander Ditter, Beat Weiss, Christoph Hagleitner, and Dietmar Fey. System Architecture for Network-Attached FPGAs in the Cloud using Partial Reconfiguration. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 293–300, 2019.

[177] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. ReDMArk: Bypassing RDMA security mechanisms. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

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

[179] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S. Berger, James C. Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. Enso: A Streaming Interface for NIC-Application Communication. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 1005–1025, Boston, MA, July 2023. USENIX Association.

[180] Ciprian Seiculescu, Srinivasan Murali, Luca Benini, and Giovanni De Micheli. A method to remove deadlocks in networks-on-chips with wormhole flow control. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 1625–1628. IEEE, 2010.

[181] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. FPMR: MapReduce Framework on FPGA. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, pages 93–102, New York, NY, USA, 2010. ACM.

[182] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 87–102, Renton, WA, April 2022. USENIX Association.

[183] Nikita Shirokov and Ranjeeth Dasineni. Open-sourcing Katran, a Scalable Network Load Balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/. Accessed: 2021-10-26.

[184] Ran Shu, Peng Cheng, Guo Chen, Zhiyuan Guo, Lei Qu, Yongqiang Xiong, Derek Chiou, and Thomas Moscibroda. Direct Universal Access: Making Data Center Resources Available to FPGA. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 127–140, Boston, MA, February 2019. USENIX Association.

[185] David Sidler, Zsolt István, and Gustavo Alonso. Low-latency tcp/ip stack for data center applications. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE.

[186] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. StRoM: Smart Remote Memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[187] Mark Silberstein, Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, and Emmett Witchel. GPUnet: Networking abstractions for GPU programs. *ACM Transactions on Computer Systems (TOCS)*, 34(3):1–31, 2016.

[188] Yong Ho Song and T.M. Pinkston. A progressive approach to handling message-dependent deadlock in parallel computer systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):259–275, 2003.

[189] Stuart Sutherland and Don Mills. Standard Gotchas Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know. https://lcdm-eng.com/papers/snug06_Verilog%20Gotchas%20Part1.pdf. Accessed: 2025-1-8.

[190] Ian Swarbrick, Dinesh Gaitonde, Sagheer Ahmad, Brian Gaide, and Ygal Arbel. Network-on-Chip Programmable Platform in Versal ACAP Architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 212–221, New York, NY, USA, 2019. Association for Computing Machinery.

[191] UCSD SysNet. Corundum. https://github.com/ucsdsysnet/corundum. Accessed: 2020-4-13.

[192] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with PathDump. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 233–248, 2016.

[193] Naif Tarafdar, Thomas Lin, Eric Fukuda, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. Enabling flexible network FPGA clusters in a heterogeneous cloud data center. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 237–246. ACM, 2017.

[194] Tencent. F-stack. https://github.com/F-Stack/f-stack. Accessed: 2024-4-17.

[195] David Thaler and Poorna Gaddehosur. Making ebpf work on windows. https://cloudblogs.microsoft.com/opensource/2021/05/10/making-ebpf-work-on-windows/. Accessed: 2021-10-26.

[196] Chandrahas Tirumalasetty, Chih Chieh Chou, Narasimha Reddy, Paul Gratz, and Ayman Abouelwafa. Reducing minor page fault overheads through enhanced page walker. *ACM Trans. Archit. Code Optim.*, 19(4), September 2022.

[197] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A SmartNIC-Driven Accelerator-Centric Architecture for Network Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 117–131, New York, NY, USA, 2020. Association for Computing Machinery.

[198] Yatish Turakhia, Gill Bejerano, and William J. Dally. Darwin: A genomics co-processor provides up to 15,000x acceleration on long read assembly. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 199–213, New York, NY, USA, 2018. Association for Computing Machinery.

[199] Yao Wang and G. Edward Suh. Efficient timing channel protection for on-chip networks. In *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, pages 142–151, 2012.

[200] Hassan M.G. Wassel, Ying Gao, Jason K. Oberg, Ted Huffmire, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Networks on chip with provable security properties. *IEEE Micro*, 34(3):57–68, 2014.

[201] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. Disaggregated FPGAs: Network performance comparison against bare-metal servers, virtual machines and Linux containers. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 9–17. IEEE, 2016.

[202] Jagath Weerasinghe, Raphael Polig, Francois Abel, and Christoph Hagleitner. Network-attached FPGAs for data center applications. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 36–43, 2016.

[203] David Wragg. Unimog - Cloudflare's Edge Load Balancer. https://blog.cloudflare.com/unimog-cloudflares-edge-load-balancer/. Accessed: 2021-10-26.

[204] Lisa Wu, David Bruns-Smith, Frank A. Nothaft, Qijing Huang, Sagar Karandikar, Johnny Le, Andrew Lin, Howard Mao, Brendan Sweeney, Krste Asanović, David A. Patterson, and Anthony D. Joseph. Fpga accelerated indel realignment in the cloud. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 277–290, 2019.

[205] Xilinx. Alveo Card Management Solution Subsystem Product Guide. https://docs.xilinx.com/r/en-US/pg348-cms-subsystem/Register-Space. Accessed: 2023-11-29.

[206] Xilinx. Alveo U200 and U250 Data Center Accelerator Cards Data Sheet (DS962). https://docs.amd.com/r/en-US/ds962-u200-u250/Alveo-Product-Details. Accessed: 2024-4-16.

[207] TCP/IP Offload Engine 10/25G. https://www.xilinx.com/products/intellectual-property/1-y7rb2p.html#productspecs. Accessed: 2022-6-28.

[208] Xilinx. UltraScale Architecture and Product Data Sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf. Accessed: 2019-12-12.

[209] Xilinx. UltraScale+ Integrated 100G Ethernet Subsystem. https://www.xilinx.com/products/intellectual-property/cmac_usplus.html. Accessed: 2020-1-30.

[210] Xilinx. Versal Premium Series. https://www.xilinx.com/products/silicon-devices/acap/versal-premium.html. Accessed: 2022-7-4.

[211] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 195–211, New York, NY, USA, 2021. Association for Computing Machinery.

[212] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. Aequitas: Admission Control for Performance-Critical RPCs in Datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 1–18, New York, NY, USA, 2022. Association for Computing Machinery.

[213] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. Achieving 100Gbps Intrusion Prevention on a Single Server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1083–1100. USENIX Association, November 2020.

[214] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating Distributed Protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1391–1407, Boston, MA, April 2023. USENIX Association.

[215] Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Shixin Ji, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Yiyu Shi, Deming Chen, Jason Cong, and Peipei Zhou. CHARM 2.0: Composing Heterogeneous Accelerators for Deep Learning on Versal ACAP Architecture. *ACM Trans. Reconfigurable Technol. Syst.*, 17(3), September 2024.