Beehive: A Flexible Network Stack for Direct-Attached Accelerators

Katie Lim*, Matthew Giordano*, Theano Stavrinos*,
Irene Zhang†, Jacob Nelson†, Baris Kasikci*, Thomas Anderson*
{katielim, mgiordan, thst}@cs.washington.edu,
{irene.zhang, jacob.nelson}@microsoft.com, {baris, tom}@cs.washington.edu
*University of Washington, Seattle, WA, U.S.A., †Microsoft Research, Redmond, WA, U.S.A.

2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 979-8-3503-5057-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/MICRO61859.2024.00037

***Abstract*—Direct-attached accelerators, where application accelerators are directly connected to the datacenter network via a hardware network stack, offer substantial benefits in terms of reduced latency, CPU overhead, and energy use. However, a key challenge is that modern datacenter network stacks are complex, with interleaved protocol layers, network management functions, and virtualization support. To operators, network feature agility, diagnostics, and manageability are often considered just as important as raw performance. By contrast, existing hardware network stacks only support basic protocols and are often difficult to extend since they use fixed processing pipelines.**

We propose Beehive, a new, open-source FPGA network stack for direct-attached accelerators designed to enable flexible and adaptive construction of complex network functionality in hardware. Application and network protocol elements are modularized as tiles over a network-on-chip substrate. Elements can be added or scaled up/down to match workload characteristics with minimal effort or changes to other elements. Flexible diagnostics and control are integral, with tooling to ensure deadlock safety. Our implementation interoperates with standard Linux TCP and UDP clients, with a 4x improvement in end-to-end RPC tail latency for Linux UDP clients versus a CPU-attached accelerator. Beehive is available at https://github.com/beehive-fpga/beehive

***Index Terms*—hardware acceleration, networking, network stack, FPGA**

I. Introduction

Hardware accelerators are becoming increasingly common in datacenters to reduce cost, improve performance, and reduce energy consumption relative to server CPUs. Typically, accelerators are hosted over the PCIe I/O bus, with the server CPU mediating all communication with the accelerator, illustrated in Figure 1(c). An alternative model directly attaches the accelerator to the network, with its own network functionality implemented in hardware, illustrated in Figure 1(b). Bypassing the CPU potentially reduces end-to-end latency, latency variability, and overhead, freeing up the CPU for other purposes.

A barrier to any hardware network implementation is the difficulty of meeting the full set of datacenter network operational requirements [6], [53]. 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 [17],

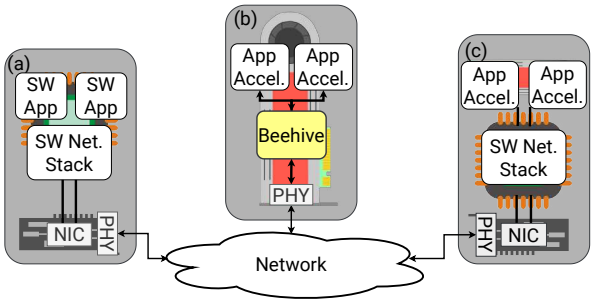


Fig. 1: **(a)** represents a standard CPU server node; **(b)** a direct-attached accelerator using the Beehive network stack; **(c)** an accelerator using a CPU network stack.

[80], encryption [19], [50], application-specific load balancing [18], [36], and information flow control [28]. Deployment flexibility necessitates management features like virtual networking [23], [29], [44], access control lists [54], congestion control [45], [48], traffic prioritization [33], [57], and load balancing [26], [61], [70], [75]. Deployment maintainability requires dynamic support for network monitoring [8], [81], reconfiguration [11], [43], and debugging [72].

An example of a highly-flexible software network stack is Google's Snap networking system [53]. 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 hardware [47], [49], 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 [4], [16], [66], or in the case of RDMA, require extensive engineering to make work in practice [6], [65].

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

979-8-3503-5057-9/24/$31.00 ©2024 IEEE
DOI 10.1109/MICRO61859.2024.00037

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 Beehive and 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 Beehive and 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 Beehive that 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.

II. Motivation

We motivate direct-attached accelerators by investigating their latency benefits over CPU-attached accelerators. Prior work has shown benefits over the Linux network stack [13], [74]. However, cutting-edge systems aiming for the lowest possible latency typically use a DPDK network stack, which can achieve single digit microsecond latencies [39], [73], [79].

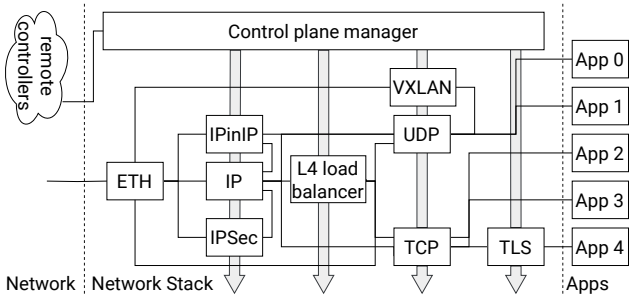


Fig. 2: A high-level diagram of the type of network stack Beehive targets. Along with multiple transport protocols, this stack has IP-in-IP and VXLAN for network virtualization and a component for an L4 load balancer. The downward arrows represent control-plane communication, which potentially needs access any module internal to the network stack.

TABLE I: Comparison of median and p99 round-trip time of a UDP echo across different configurations. Client machines use software networking. Beehive represents the configuration in Figure 1(b); Linux and DPDK to Accel. represent Figure 1(c).

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

Our experiment compares the direct-attached configuration in Figure 1(b) and the software-hosted configuration in Figure 1(c). 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 [73], a DPDK network stack, as 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 [67] 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 (Linux or F-Stack).

We report median and 99^{th} percentile (p99) RTTs in Table I. As expected, trampolining every RPC 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. 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. When the network stack is at user level on both the client and server, scheduling variance is reduced as the server CPU busy-waits for incoming requests, at the cost of higher CPU overhead. However, the relative benefit of Beehive is similar, with 1.5× better median and p99 tail latency relative to redirection through the CPU.

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. Realizing this benefit requires a hardware network stack that can be flexibly reconfigured to meet the needs of datacenter network management.

III. Design Goals

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

A. 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 [53] or 1RMA [2]) 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 [27]. 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.

	Std. Protocols	Modular	Scalable	Performant	Mgmt. Features	Open Source
Limago [66]	✓	✳	✗	✓	✗	✓
PANIC [49]	✳	✓	✗	✓	✗	✳
ClickNP [47]	✳	✓	✳	✓	✓	✗
LTL [13]	✗	✗	✳	✓	✳	✗
Beehive	✓	✓	✓	✓	✓	✓

TABLE II: Beehive and prior work versus the goals in Section III-A. The stars indicate partially meeting the goal.

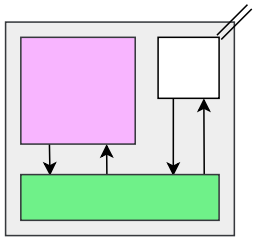


Fig. 3: Architecture of a tile.

B. Comparing versus related work

As shown in Table II, 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 VII-G. Unfortunately, we were unable to run Limago on FPGA using their code [32] 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 VII-C, 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.

IV. Design

A. Beehive's architecture

The basic component in Beehive is the tile, shown in Figure 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

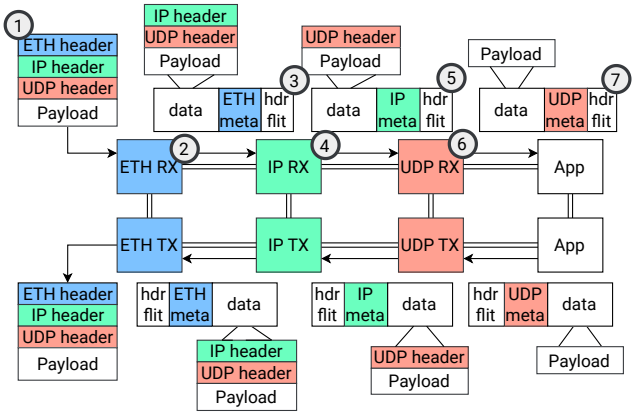


Fig. 4: The flow through which a packet is processed or constructed in Beehive.

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.

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 [49]. We discuss this decision in more detail in Section IV-D. 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 IV-E.

B. Processing a packet

Figure 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 VI.

C. 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 [22], and makes it easier to chain offloads together. ClickNP [47] and PANIC [49], 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.

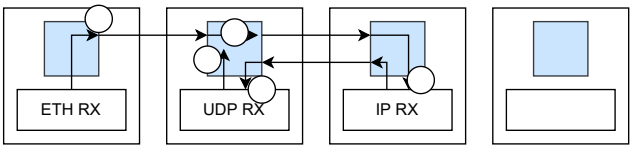
We prefer a NoC interconnect for two main reasons related to our goal of scalability. First, we can take advantage of the multiplexing provided by the NoC routers. Certain tiles may interact with many other tiles, e.g. if we instantiate multiple copies of the same component or common services such as 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.

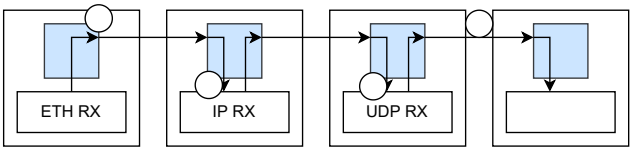
These scalability benefits apply both to the data plane and control plane. We discuss the benefits further for the control plane specifically in Section IV-F.

D. 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



(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.

Fig. 5: An example of how tile assignment affects deadlock. Beehive takes advantage of protocol layer ordering, so a packet always acquires NoC resources in the same order.

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 packets. 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.

E. 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 [3], [21], [46], [58], [68].

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 [21]. 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 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 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 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 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 [46], [71]. These buffers give space for the NoC to drain into, freeing routing resources.

F. 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 V-E.

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.

G. 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 V-D.

V. 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 VI.

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 [31], 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.

A. Network-on-chip (NoC)

We use the 2D mesh NoC from OpenPiton [7] 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.

B. 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. 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 V-D.

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 IV-F. 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.

C. 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.

D. TCP engine

To evaluate how Beehive can 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 [10]. Currently, it does not support selective acknowledgments, initiating connections, or congestion control. Full TCP offload functionality has been demonstrated by previous work [66] 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.

E. 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.

F. 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 [9], 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.

G. 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

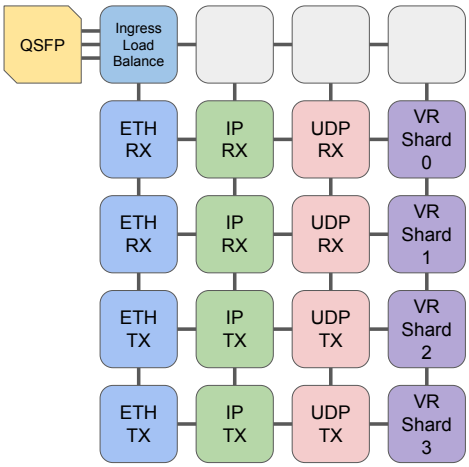


Fig. 6: Beehive tile layout for Viewstamped Replication.

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 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 6 is a visualization of the layout generated by the XML file for the consensus witness design in Section VI-B.

VI. Integrating with Beehive

A. Erasure coding

To demonstrate the benefits of Beehive for 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 [34], [41], [64]. 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 Beehive as 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.

B. 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 [14], [51], [60] 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 [82].

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 [20], [35], [37]. 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 [38], 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 [37], [38] 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 Beehive can 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 [63]. VR witnesses are integrated into Beehive as 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.

VII. 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.

A. Setup

We use Vivado 2021.2 for building our FPGA images. Beehive is 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 [76]. We poll these counters every second to calculate energy over the benchmarking period.

B. 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 [24], 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 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 [79], 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.

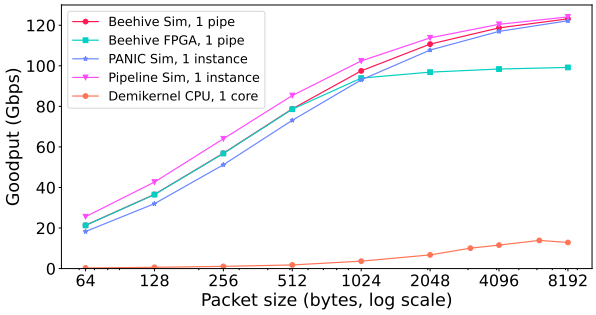
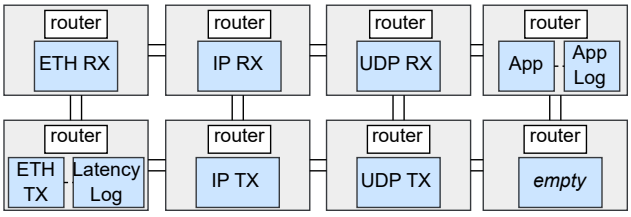
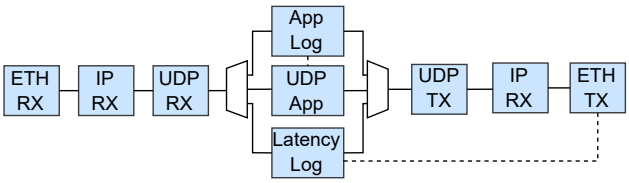


Fig. 7: Packet size vs. goodput for a UDP echo application. Beehive and CALM perform almost identically across all packet sizes and outperform Demikernel.



(a) Setup for Beehive's UDP stack



(b) Setup for the fixed pipeline UDP stack

Fig. 8: UDP stacks for the echo microbenchmark

C. UDP echo

Throughput: We compare UDP echo goodput for Beehive (shown in Figure 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 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 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

TABLE III: Energy consumption and goodput for Reed-Solomon encoding using Beehive versus CPU for 1, 2, 3 and 4 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×

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 [59]. 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 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.

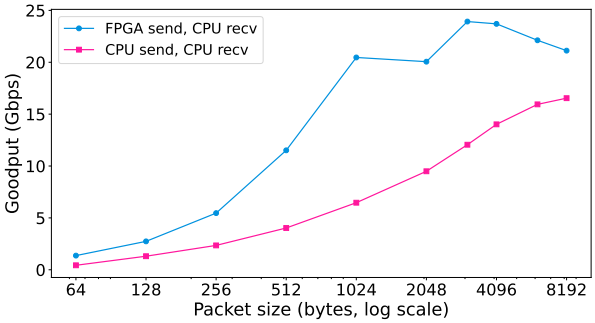


Fig. 9: Packet size vs. goodput for Beehive and Linux TCP send. The (CPU send/FPGA receive) is omitted, as it is approximately the same as (CPU send/CPU receive) due to the CPU send path being the bottleneck.

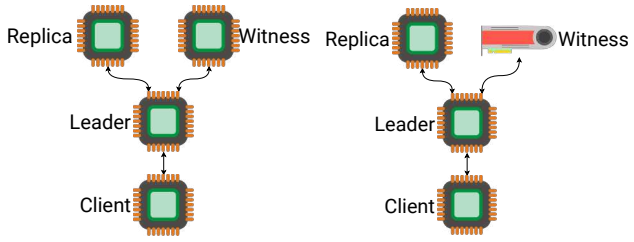
D. TCP throughput

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.

We vary whether the sender or the receiver is the FPGA or the CPU. The results are shown in Figure 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 Beehive achieves 2666 KReq/s versus the CPU's 843 KReq/s, a 3.2× speedup.

E. Reed-Solomon encoding acceleration

To evaluate Beehive's scaling architecture, we evaluate a duplicated Reed-Solomon (RS) encoding accelerator on Beehive versus a CPU implementation of the same algorithm in Table III. 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 III. For comparison, we use the open-source Reed-Solomon encoding implementation from BackBlaze [5] running on CPUs which we then duplicate across cores.



(a) CPU-only setup (b) Beehive witness setup

Fig. 10: Experimental setups for VR evaluation

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

F. 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 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 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 eventual 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 [15], [52], [55].

Results: We plot latency versus throughput for differing numbers of shards in Figure 11. We increase offered load by increasing the number of client threads sending requests to the leader. The results are shown in Figure 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 11. These results are shown in Table IV. 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.

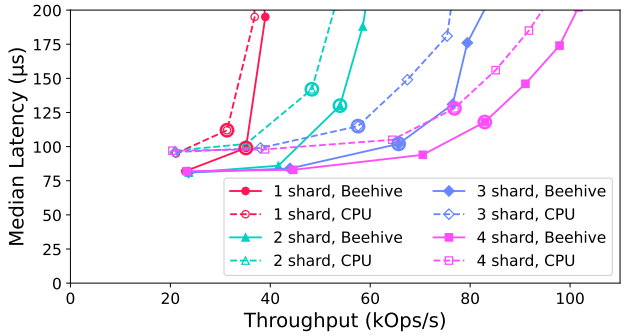


Fig. 11: Latency vs. throughput for the VR key-value store workload varying the number of shards and client threads. The FPGA witness consistently outperforms the equivalent CPU cores in both latency and throughput.

TABLE IV: Energy per operation (measured at the witness) and performance metrics (measured at the clients) at the circled points in Figure 11.

Shards	1	2	3	4
CPU Energy (mJ/op)	1.51	1.03	0.90	0.70
Beehive Energy (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
Beehive Throughput (kOps/s)	35	54	66	83
Speedup	1.12×	1.12×	1.14×	1.08×
CPU Median Latency (μs)	112	142	115	128
Beehive Median Latency (μs)	99	130	102	118
Improvement	1.13×	1.09×	1.13×	1.08×
CPU p99 Latency (μs)	273	372	339	412
Beehive p99 Latency (μs)	281	334	304	394
Improvement	0.97×	1.11×	1.12×	1.05×

G. Hardware resource utilization

The hardware utilization of the Beehive infrastructure is shown in Table V. For the UDP stack used in Section VII-C, 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.

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

H. Flexibility

As a quantitative proxy for flexibility, we count the lines of code (LoC) required to insert an additional instance of an

TABLE V: 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

TABLE VI: 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

implemented service (network function or application) into the design for our three designs. Results are shown in Table VI.

I. 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 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 [1], [78] now support hardened NoC resources and could improve the quality of results.

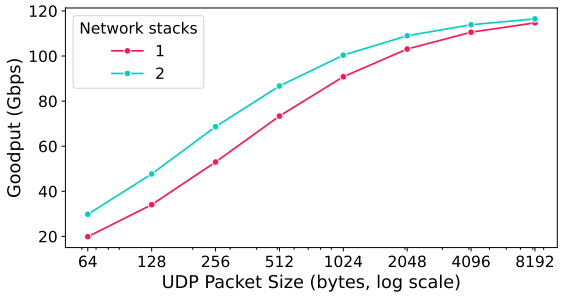


Fig. 12: Packet size vs. goodput for a UDP echo application, running on Beehive with multiple network stacks instantiated. 2 network stacks maxes out the load balancer's throughput.

VIII. Related Work

A. Packet processing

PANIC [49] 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, limiting scalability. While PANIC does not directly address deadlocks, its central scheduler drops packets when it runs out of buffer space, preventing deadlock. However, this makes integrating RPC/TCP applications into PANIC is challenging, since it assumes that operations occur on a packet level and the scheduler may drop an acknowledged packet, violating TCP semantics.

ClickNP [47] is an FPGA-accelerated packet processing framework that also supports the integration of arbitrary processing elements. However, it does not use a NoC. Instead, components are directly connected via FIFOs, which makes it harder to replicate elements. Since ClickNP aims to accelerate software network functions, it 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 [42] is an FPGA framework for middleboxes. It uses an interconnect to connect custom processing elements they call reconfigurable processing units (RPUs) that 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 inter-RPU traffic has a fairly significant latency penalty.

A more restrictive approach leverages reconfigurable match-action tables. 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. Typically, there is a pipeline of these processing elements [11], [30], [40]. However, match-action style processing is not well-suited for highly stateful processing [62] typical of application-level offloads. Other models have been proposed for stateful packet processing. Flowblaze uses an FSM-based model [62]. However, they specifically

say that workloads above the transport layer are out of scope. hXDP proposed a processor for eBPF bytecode [12] 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.

B. Transport protocol offloads

Another related vein of work are transport protocol offloads. Most of these are TCP offload engines available as custom chips [16] or encrypted IP cores for FPGAs [25], [56], [77]. They generally do not support the full range of functions found in datacenter network stacks.

Some TCP offload engines could potentially support modification. Limago [66] 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 [4] 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 [69] 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 [13], 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 [30]. However, it is unknown if these are ever combined and if so, how it would support new protocols or network functions.

IX. 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 accelerators to reduce network overhead. In this paper, 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. We have open-sourced Beehive for reuse at https://github.com/beehive-fpga/beehive.

Acknowledgments

We thank the anonymous reviewers for feedback that substantially improved the paper. This work was supported by grants from VMware Research, the National Science Foundation (CNS-2104548, No. 2213387), the University of Washington Center for the Future of Cloud Infrastructure (FOCI), the Intel TSA center, and the NSF Graduate Research Fellowship. Pratyush Patel assisted in an early draft of this paper. We would like to thank Hugo Sadok for his invaluable help with getting Ensō running. We also thank Alexey Lavrov and Jonathan Balkind for feedback on various drafts of this paper.

Artifact Appendix

A. Abstract

Beehive is a NoC-based network stack for direct attached accelerators designed to enable flexible construction of complex network functionality in hardware. This appendix outlines the steps to access Beehive. All files necessary to build the various artifacts for the experiments in the paper are available in the Beehive repository.

Full evaluation and reproduction of evaluation results requires testbed access to a 100 Gb switch, at least 4 100 Gb NICs, and an Alveo U200 FPGA. It also requires access to Vivado 2021.2 to build the hardware designs. Simulation requires access to Python and ModelSim 2019.2.

B. Artifact check-list (meta-information)

- **Program:** Vivado 2021.2, ModelSim 2019.2
- **Compilation:** Follow the directions inside `beehive/README.md`
- **Hardware:** Alveo U200/U250
- **How much disk space required (approximately)?:** 200 MB
- **Publicly available?:** Yes, see below
- **Code licenses (if publicly available)?:** BSD 3-Clause
- **Data licenses (if publicly available)?:** BSD 3-Clause
- **Archived (provide DOI)?:** **https://doi.org/10.5281/zenodo.13308868**

C. Description

1) How to access: Beehive is accessible at https://github.com/beehive-fpga/beehive

D. Installation

Please refer to the instructions inside beehive/README.md

E. Evaluation and expected results

Each hardware build needs some sort of input data to run on. You are expected to connect to the hardware via TCP or UDP and send the appropriate packets to drive the device and to fetch evaluation information from the device. If simulating, the simulation driver will drive the design. In this case, you should see outputs relating to that run.

F. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae

REFERENCES

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

[2] A. Akella, A. Vahdat, A. Singhvi, B. Montazeri, D. Gibson, H. Wassel, J. Scherpelz, M. M. K. Martin, M. C. Wong-Chan, M. Mclaren, P. Chandra, R. Cauble, S. Clark, S. Sabato, and T. 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*, New York, NY, USA, 2020, p. 708–721.

[3] K. Anjan and T. M. Pinkston, "An efficient, fully adaptive deadlock recovery scheme: DISHA," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, 1995, pp. 201–210.

[4] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff, "Enabling Programmable Transport Protocols in High-Speed NICs," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, February 2020.

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

[6] W. Bai, S. S. Abdeen, A. Agrawal, K. K. Attre, P. Bahl, A. Bhagat, G. Bhaskara, T. Brokhman, L. Cao, A. Cheema, R. Chow, J. Cohen, M. Elhaddad, V. Ette, I. Figlin, D. Firestone, M. George, I. German, L. Ghai, E. Green, A. Greenberg, M. Gupta, R. Haagens, M. Hendel, R. Howlader, N. John, J. Johnstone, T. Jolly, G. Kramer, D. Kruse, A. Kumar, E. Lan, I. Lee, A. Levy, M. Lipshteyn, X. Liu, C. Liu, G. Lu, Y. Lu, X. Lu, V. Makhervaks, U. Malashanka, D. A. Maltz, I. Marinos, R. Mehta, S. Murthi, A. Namdhari, A. Ogus, J. Padhye, M. Pandya, D. Phillips, A. Power, S. Puri, S. Raindel, J. Rhee, A. Russo, M. Sah, A. Sheriff, C. Sparacino, A. Srivastava, W. Sun, N. Swanson, F. Tian, L. Tomczyk, V. Vadlamuri, A. Wolman, Y. Xie, J. Yom, L. Yuan, Y. Zhang, and B. Zill, "Empowering Azure Storage with RDMA," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 49–67. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/bai

[7] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. 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*, ser. ASPLOS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 217–232. [Online]. Available: https://doi.org/10.1145/2872362.2872414

[8] J. R. Ballard, I. Rae, and A. Akella, "Extensible and Scalable Network Monitoring Using OpenSAFE," *INM/WREN*, vol. 10, 2010.

[9] S. Bishop, M. Fairbairn, H. Mehnert, M. Norrish, T. Ridge, P. Sewell, M. Smith, and K. Wansbrough, "Engineering with Logic: Rigorous Test-Oracle Specification and Validation for TCP/IP and the Sockets API," *J. ACM*, vol. 66, no. 1, dec 2018. [Online]. Available: https://doi.org/10.1145/3243650

[10] E. Blanton, D. V. Paxson, and M. Allman, "TCP Congestion Control," RFC 5681, Sep. 2009. [Online]. Available: https://www.rfc-editor.org/info/rfc5681

[11] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, p. 99–110, aug 2013. [Online]. Available: https://doi.org/10.1145/2534169.2486011

[12] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco, "hXDP: Efficient Software Packet Processing on FPGA NICs," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 973–990. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/brunella

[13] A. Caulfield, E. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A Cloud-Scale Acceleration Architecture," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, October 2016, pp. 1–13.

[14] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos Made Live - An Engineering Perspective (2006 Invited Talk)," in *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, 2007. [Online]. Available: http://dx.doi.org/10.1145/1281100.1281103

[15] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker, "Making gnutella-like p2p systems scalable," in *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2003, pp. 407–418.

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

[17] I. Cho, A. Saeed, J. Fried, S. J. Park, M. Alizadeh, and A. Belay, "Overload Control for u-scale RPCs with Breakwater," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 299–314. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/cho

[18] Cilium. L7 load balancing and url re-writing. https://docs.cilium.io/en/latest/gettingstarted/servicemesh/envoy-traffic-management/. Accessed: 2022-6-28.

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

[20] N. Crooks, Y. Pu, L. Alvisi, and A. Clement, "Seeing is Believing: A Client-Centric Specification of Database Isolation," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, ser. PODC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 73–82. [Online]. Available: https://doi.org/10.1145/3087801.3087802

[21] Dally and Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Transactions on Computers*, vol. C-36, no. 5, pp. 547–553, 1987.

[22] W. J. Dally and B. Towles, "Route Packets, Not Wires: On-Chip Inteconnection Networks," in *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 684–689. [Online]. Available: https://doi.org/10.1145/378239.379048

[23] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCabooter, M. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat, "Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 373–387. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/dalton

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

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

[26] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A Fast and Reliable Software Network Load Balancer," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, 2016, pp. 523–535. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud

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

[28] European Parliament and Council of the European Union. (2016) Regulation (EU) 2016/679 of the European Parliament and of the Council. [Online]. Available: https://data.europa.eu/eli/reg/2016/679/oj

[29] D. 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)*. Boston, MA: USENIX Association, Mar. 2017, pp. 315–328. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/firestone

[30] D. Firestone, A. Putnam, H. Angepat, D. Chiou, A. Caulfield, E. Chung, M. Humphrey, K. Ovtcharov, J. Padhye, D. Burger, D. Maltz, A. Greenberg, S. Mundkur, A. Dabagh, M. Andrewartha, V. Bhanu, H. K. Chandrappa, S. Chaturmohta, J. Lavier, N. Lam, F. Liu, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, K. Vaid, and D. A. Maltz, "Azure Accelerated Networking: SmartNICs in the Public Cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2018.

[31] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen, "Corundum: An Open-Source 100-Gbps NIC," in *28th IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2020.

[32] E. Z. 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.

[33] P. Goyal, P. Shah, K. Zhao, G. Nikolaidis, M. Alizadeh, and T. E. Anderson, "Backpressure Flow Control," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 779–805.

[34] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure Coding in Windows Azure Storage," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, Jun. 2012, pp. 15–26. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/huang

[35] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, Jun. 2010. [Online]. Available: https://www.usenix.org/conference/usenix-atc-10/zookeeper-wait-free-coordination-internet-scale-systems

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

[37] Z. István, D. Sidler, and G. Alonso, "Caribou: Intelligent Distributed Storage," *Proc. VLDB Endow.*, vol. 10, no. 11, p. 1202–1213, Aug. 2017. [Online]. Available: https://doi.org/10.14778/3137628.3137632

[38] Z. István, D. Sidler, G. Alonso, and M. Vukolic, "Consensus in a box: Inexpensive coordination in hardware," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 425–438.

[39] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter RPCs can be General and Fast," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, MA: USENIX Association, February 2019, pp. 1–16.

[40] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy, "High Performance Packet Processing with FlexNIC," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 67–81. [Online]. Available: https://doi.org/10.1145/2872362.2872367

[41] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. 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: USENIX Association, Feb. 2012. [Online]. Available: https://www.usenix.org/conference/fast12/rethinking-erasure-codes-cloud-file-systems-minimizing-io-recovery-and-degraded

[42] M. Khazraee, A. Forencich, G. C. Papen, A. C. Snoeren, and A. 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*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 586–605. [Online]. Available: https://doi.org/10.1145/3582016.3582067

[43] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[44] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang, "Network Virtualization in Multi-tenant Datacenters," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 203–216. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/koponen

[45] G. Kumar, N. Dukkipati, K. Jang, H. M. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan *et al.*, "Swift: Delay is Simple and Effective for Congestion Control in the Datacenter," in *Proceedings of the ACM SIGCOMM 2020 Conference*, 2020, pp. 514–528.

[46] A. Lankes, T. Wild, A. Herkersdorf, S. Sonntag, and H. 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*, 2010, pp. 17–24.

[47] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1–14. [Online]. Available: https://doi.org/10.1145/2934872.2934897

[48] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu, "HPCC: High Precision Congestion Control," in *Proceedings of the ACM SIGCOMM 2019 Conference*, 2019, p. 44–58.

[49] J. Lin, K. Patel, B. E. Stephens, A. Sivaraman, and A. Akella, "PANIC: A High-Performance Programmable NIC for Multi-tenant Networks," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 243–259. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/lin

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

[51] B. Liskov and J. Cowling, "Viewstamped Replication Revisited," MIT, Tech. Rep. MIT-CSAIL-TR-2012-021, Jul. 2012.

[52] H. Lu, S. Sen, and W. Lloyd, "Performance-Optimal Read-Only Transactions," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 333–349.

[53] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, M. Ryan, E. Rubow, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat, "Snap: a Microkernel Approach to Host Networking," in *In ACM SIGOPS 27th Symposium on Operating Systems Principles*, New York, NY, USA, 2019.

[54] 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.

[55] D. L. Mills, "Internet time synchronization: the network time protocol," *IEEE Transactions on communications*, vol. 39, no. 10, pp. 1482–1493, 1991.

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

[57] B. Montazeri, Y. Li, M. Alizadeh, and J. 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*. New York, NY, USA: ACM, 2018, p. 221–235.

[58] S. Murali, P. Meloni, F. Angiolini, D. Atienza, S. Carta, L. Benini, G. De Micheli, and L. 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*, 2006, pp. 158–163.

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

[60] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. USA: USENIX Association, 2014, p. 305–320.

[61] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri, "Ananta: Cloud Scale Load Balancing," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, p. 207–218, aug 2013. [Online]. Available: https://doi.org/10.1145/2534169.2486026

[62] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Siracusano, "FlowBlaze: Stateful Packet Processing in Hardware," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 531–548. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/pontarelli

[63] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy, "Designing Distributed Systems Using Approximate Synchrony in Data

Center Networks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[64] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. 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: USENIX Association, Jun. 2013. [Online]. Available: https://www.usenix.org/conference/hotstorage13/workshop-program/presentation/rashmi

[65] B. Rothenberger, K. Taranov, A. Perrig, and T. Hoefler, "ReDMArk: Bypassing RDMA security mechanisms," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[66] M. Ruiz, D. Sidler, G. Sutter, G. Alonso, and S. López-Buedo, "Limago: An FPGA-Based Open-Source 100 GbE TCP/IP Stack," in *29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 286–292.

[67] H. Sadok, N. Atre, Z. Zhao, D. S. Berger, J. C. Hoe, A. Panda, J. Sherry, and R. Wang, "Enso: A Streaming Interface for NIC-Application Communication," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 1005–1025. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/sadok

[68] C. Seiculescu, S. Murali, L. Benini, and G. 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)*. IEEE, 2010, pp. 1625–1628.

[69] R. Shashidhara, T. Stamler, A. Kaufmann, and S. Peter, "FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 87–102. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/shashidhara

[70] N. Shirokov and R. 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.

[71] Y. H. Song and T. Pinkston, "A progressive approach to handling message-dependent deadlock in parallel computer systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 3, pp. 259–275, 2003.

[72] P. Tammana, R. Agarwal, and M. Lee, "Simplifying datacenter network debugging with PathDump," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 233–248.

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

[74] J. Weerasinghe, F. Abel, C. Hagleitner, and A. 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)*. IEEE, 2016, pp. 9–17.

[75] D. Wragg. Unimog - cloudflare's edge load balancer. https://blog.cloudflare.com/unimog-cloudflares-edge-load-balancer/. Accessed: 2021-10-26.

[76] Xilinx. Alveo card management solution subsystem product guide. https://docs.xilinx.com/r/en-US/pg348-cms-subsystem/Register-Space. Accessed: 2023-11-29.

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

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

[79] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar, P. H. Penna, M. Demoulin, P. Choudhury, and A. Badam, "The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 195–211. [Online]. Available: https://doi.org/10.1145/3477132.3483569

[80] Y. Zhang, G. Kumar, N. Dukkipati, X. Wu, P. Jha, M. Chowdhury, and A. Vahdat, "Aequitas: Admission Control for Performance-Critical RPCs in Datacenters," in *Proceedings of the ACM SIGCOMM 2022 Conference*, ser. SIGCOMM '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–18. [Online]. Available: https://doi.org/10.1145/3544216.3544271

[81] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry, "Achieving 100Gbps Intrusion Prevention on a Single Server," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 1083–1100. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng

[82] Y. Zhou, Z. Wang, S. Dharanipragada, and M. Yu, "Electrode: Accelerating Distributed Protocols with eBPF," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 1391–1407. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/zhou