NICA: An Infrastructure for Inline Acceleration of Network Applications

Haggai Eran, *Technion–Israel Institute of Technology & Mellanox Technologies;*
Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein,
Technion–Israel Institute of Technology

https://www.usenix.org/conference/atc19/presentation/eran

This paper is included in the Proceedings of the
2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

NICA: An Infrastructure for Inline Acceleration of Network Applications

Haggai Eran[1,2], Lior Zeno[1], Maroun Tork[1], Gabi Malka[1], and Mark Silberstein[1]

[1]*Technion – Israel Institute of Technology* [2]*Mellanox Technologies*

Abstract

With rising network rates, cloud vendors increasingly deploy FPGA-based SmartNICs (F-NICs), leveraging their *inline* processing capabilities to offload hypervisor networking infrastructure. However, the use of F-NICs for *accelerating general-purpose server applications in clouds* has been limited.

NICA is a hardware-software co-designed framework for inline acceleration of the application data plane on F-NICs in multi-tenant systems. A new *ikernel* programming abstraction, tightly integrated with the network stack, enables application control of F-NIC computations that process *application* network traffic, with minimal code changes. In addition, NICA's virtualization architecture supports fine-grain time-sharing of F-NIC logic and provides I/O path virtualization. Together these features enable cost-effective sharing of F-NICs across virtual machines with strict performance guarantees.

We prototype NICA on Mellanox F-NICs and integrate ikernels with the high-performance VMA network stack and the KVM hypervisor. We demonstrate significant acceleration of real-world applications in both bare-metal and virtualized environments, while requiring only minor code modifications to accelerate them on F-NICs. For example, a transparent key-value store cache ikernel added to the stock `memcached` server reaches 40 Gbps server throughput (99% line-rate) at 6 µs 99th-percentile latency for 16-byte key-value pairs, which is 21× the throughput of a 6-core CPU with a kernel-bypass network stack. The throughput scales linearly for up to 6 VMs running independent instances of `memcached`.

1 Introduction

SmartNICs with integrated FPGAs (*F-NICs*) [19, 68, 89, 119] are an appealing platform for accelerating I/O intensive network applications. They have been increasingly deployed in data centers and public clouds [33, 66], e.g., in each MS Azure server, enabling line-rate throughput and low, predictable latency at high power efficiency [33]. Many hardware vendors, including Intel, have already announced F-NICs in their future offerings [98].

Data-center F-NICs are used mainly to accelerate infrastructure tasks, such as network functions [61, 72, 80, 119] and software-defined networking [33, 66]. These tasks leverage the F-NIC's *inline processing* capabilities, where data is processed while being transferred between the host and the network, without CPU involvement. In addition, F-NICs are often repurposed as standalone FPGAs running complete applications, e.g., search or deep learning [20, 24, 60, 86].

This paper explores new acceleration opportunities emerging from the growing deployment of F-NICs in clouds, beyond infrastructure tasks and monolithic applications. We seek to leverage F-NICs for inline acceleration of data plane processing in network-intensive applications. For example, F-NICs may run datacenter tax tasks, such as deserialization, hashing, and authentication, which reportedly consume over a quarter of the CPU cycles in data centers [48]. An F-NIC may serve as an extra caching layer for key-value stores, responding directly in case of a hit and eliminating the CPU involvement. We show, for example, that this architecture achieves near line-rate throughput (40 Gbps) for stock `memcached` (§6.2.1). Promising results for application-specific traffic steering, packet transformation, and network stack offloading have been reported in prior work [50, 83]. We discuss these and other applications in §3.

Unfortunately, building such F-NIC-accelerated applications today is hard. First, there are **no adequate operating system abstractions** for inline acceleration of general-purpose applications on F-NICs. Such abstractions should associate F-NIC tasks with the application process, and they should provide well-defined execution boundaries and isolated per-task state while supporting easy integration of F-NIC functionality with the application logic. OpenCL and CUDA provide general lookaside acceleration support, but they are a poor match for F-NICs because they require explicit kernel invocation and data transfers that are irrelevant for the inline processing scenario. Floem [83] provides language-level constructs to accelerate applications on SmartNICs, but it targets CPU-based rather than FPGA-based SmartNIC architectures, and requires application refactoring to use its data flow model, complicating acceleration of legacy workloads. SmartNIC-accelerated networking frameworks such as AccelNet, eBPF-XDP, and DPDK rte_security [33, 39, 84] are domain specific and lack application-level abstractions. Systems for data plane acceleration, e.g. P4 and FlexNIC [13, 50], expose packet-level match-action rules for F-NIC management, but lack abstractions for application-level semantics.

Second, F-NICs provide **no virtualization support**, preventing their sharing among cloud tenants. Existing virtualization mechanisms for FPGAs [18, 23, 34, 51, 101, 118] and GPUs [49, 73] rely on space partitioning or coarse-grain time sharing of the compute fabric. The former, however, results in hardware underutilization [51, 112, 116], whereas the latter may affect processing latency due to slow context switching and FPGA reconfiguration times [49, 51, 73, 91], making it unsuitable for latency-sensitive tasks. More fundamentally, F-NICs lack *I/O path virtualization* to isolate and protect per-application I/O across shared buses between the network, the FPGA and the host CPU. Thus, current F-NICs cannot guarantee performance isolation for co-located applications.

We introduce **NICA**, a system for FPGA-based **NIC** Server **A**cceleration. NICA introduces new software abstractions and co-designed F-NIC hardware runtime for application acceleration in cloud systems. NICA manages one or more *Accelerator Functional Units (AFUs)* [42, 101] – application-specific hardware accelerators hosted on an F-NIC. Such AFUs can be developed by users, or provided by cloud vendors and deployed on-demand.

OS abstraction. We introduce a novel *ikernel* (inline kernel) abstraction, which represents an AFU in a user program. An application dynamically *attaches* the ikernel to one or more transport layer sockets, activating the respective AFU. Subsequently, all traffic sent and received via these sockets is processed by the AFU without CPU invocation. To communicate via the sockets, the CPU may use standard POSIX sockets API calls, or a high-performance zero-copy interface for application-level messages. The ikernel abstraction is private to a process and provides protection for the AFU application and network state. We discuss the ikernel abstraction, its network stack integration, and FPGA runtime support in §4.1.

AFU virtualization. NICA supports sharing of AFUs among multiple virtual machines (VMs) while guaranteeing state protection and quality of service (QoS). We address two primary requirements: (1) *AFU I/O channel virtualization*, including host and network traffic, by adding anti-spoofing, classification, and packet schedulers for the I/O sent and received by AFUs; and (2) *fine-grain AFU time-sharing*, which uses a hardware task scheduler that switches contexts at a fine granularity, thus allowing better hardware utilization for latency-sensitive applications. We describe AFU virtualization in §4.2 and show how it enables performance isolation in §6.

NICA provides necessary on-FPGA services for accelerating applications on F-NICs in a multi-tenant setting, including an FPGA-resident network transport layer, compute and I/O scheduling blocks, and AFU state isolation. However, the development of high-throughput network-focused AFUs on FPGAs is beyond the scope of this paper. Fortunately, some promising solutions are emerging, such as template libraries with optimized building blocks for network processing [32]. In addition, we believe that cloud providers will increasingly offer AFUs using an "app marketplace" deployment model [4, 17, 40], with a variety of AFUs ready to be used on their infrastructure (see §3).

We prototype NICA[1] on Mellanox Innova F-NICs [68] with a Xilinx FPGA and 2GB of onboard memory. We implement the ikernel API, integrate it with the VMA kernel-bypass network stack [69], and implement the AFU virtualization support in the KVM hypervisor. We also co-design the FPGA hardware support for the software abstractions and AFU virtualization, and we integrate full UDP and partial TCP layer implementation in FPGA.

We evaluate the system with microbenchmarks and accelerate two real-world applications: a `memcached` server and a Node.js-based IoT monitoring server, by implementing the respective AFUs on the F-NIC. Enabling F-NIC acceleration required minimal software changes: 107 additional lines of C and 20 additional lines of JavaScript respectively.

A transparent hot-item cache AFU integrated with `memcached` serves GET hits at 6 µs 99th-percentile latency and 40.3 Mtps throughput for 16B keys/values, 99% of the 40 Gbps line rate and 21.6× faster than the 6-core CPU baseline. For a Zipf(0.99)-distributed workload with 0.2% SETs, NICA acceleration results in a 4.6× speedup.

NICA allows sharing of an F-NIC among multiple VMs while providing significant performance gains. It introduces negligible throughput and latency overheads while maintaining a fair bandwidth allocation, controllable by the hypervisor.

In summary, we make the following contributions:

- We introduce an ikernel OS abstraction for inline acceleration of applications on F-NICs.
- We design an F-NIC virtualization framework that supports I/O QoS and low-latency time sharing of compute resources.
- We implement NICA for Mellanox F-NICs, analyze its performance, and demonstrate the development simplicity and performance benefits for accelerating `memcached` and a Node.js-based IoT server.

2 Background

We describe the F-NIC architecture and survey FPGA programming principles and sharing mechanisms.

2.1 F-NIC architecture

We describe bump-in-the-wire F-NICs, focusing on Mellanox Innova, but others [19, 80, 89] are similar.

[1]https://github.com/acsl-technion/nica

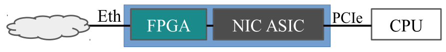


Figure 1: A bump-in-the-wire F-NIC

Bump-in-the-wire. A typical F-NIC (Figure 1) combines a commodity network ASIC (e.g., ConnectX-4 Lx NIC) with an FPGA and local DRAM. The FPGA is located *between* the ASIC and the network port, interposing on all Ethernet traffic in and out of the NIC. The FPGA and the ASIC communicate directly via an internal bus (e.g., 40 Gbps Ethernet), and a PCIe bus connects the ASIC to the host.

The bump-in-the-wire design reuses the existing data and control planes between the CPU and the NIC ASIC, with its QoS management, and virtualization support (SR-IOV), mature DMA engines, and software stack.

F-NIC programming. The development of an F-NIC-accelerated application involves both hardware logic on FPGA and associated software on the CPU. F-NIC vendors provide a lightweight *shell IP*: a set of low-level hardware interfaces for basic operations, including link-layer packet exchange with the network and the host, onboard DRAM access, and control register access. However, the vendor SDK leaves it to customers to implement higher level features such as FPGA network stack processing or virtualization support.

2.2 FPGA concepts

Field Programmable Gate Arrays (FPGAs) are "a sea" of logic, arithmetic, and memory elements, which users can configure to implement custom compute circuits. FPGA compute capacity is determined by the *area* available for the circuits.

FPGA development. FPGAs can be seen as "software-defined" hardware. The software definition, *a design*, is implemented using register transfer languages (RTL) such as Verilog. Additionally, designers can use high-level synthesis (HLS) tools to generate RTL, e.g., from a restricted version of C++ [67]. However, HLS C++ programs are different from CPU programs, and must follow certain rules, including explicit exposure of fine-grain pipeline- and task- parallelism to achieve high performance. Implementation tools then compile the design into an *FPGA image* targeting specific hardware.

Finally, users can load the image onto an FPGA (slow, up to a few seconds), entirely replacing the previous design. Some FPGAs support *partial reconfiguration* to replace only a subset of the entire FPGA, a much faster process (milliseconds), which unfortunately incurs significant area overheads [51].

FPGA sharing. There are three ways to share an FPGA: space partitioning, coarse-grain, and fine-grain time sharing.

Space partitioning divides FPGA resources into disjoint sets used by different AFUs [18, 20, 51]. If shared I/O interfaces (memory, PCIe bus) are securely isolated and multiplexed, this method enables low-overhead FPGA sharing among mutually distrustful AFUs but requires larger FPGAs to fit them all. *Coarse-grain time sharing* dynamically switches AFUs via full or partial reconfiguration [20, 51]. It incurs high switching latency and thus is not suitable for F-NICs' latency-sensitive applications. *Fine-grain time sharing* allows multiple CPU applications to use the same AFU [44]. The AFU implements the context switch internally, in hardware. Packet processing applications such as AccelNet [33] use this approach to process each packet in the context of its associated flow. Such AFUs oversee switching between the contexts; therefore this type of sharing requires AFUs to be *trusted* to ensure fair use and state isolation between their users.

NICA combines both space sharing for untrusted AFUs, and fine-grain time sharing for trusted AFUs, to achieve maximum utilization under area constraints of F-NICs.

3 Motivation

We consider emerging opportunities for application acceleration by using F-NICs in clouds.

3.1 F-NICs in data centers

Microsoft has been among the first to deploy F-NICs at large scale, having installed the Catapult F-NICs in over a million Azure servers. Their recent work [33] analyzes the cost, power, and performance trade-offs of F-NICs in data centers and decisively shows their benefits. Following Azure, other data centers, such as China Mobile [115], Tencent [66], Huawei [88], and Selectel [94], are deploying F-NICs, and leading hardware vendors are adding F-NICs to their offerings [98]. These technology trends suggest that F-NICs will become a commodity, motivating our goal to broaden the scope of their applications.

3.2 Use cases for F-NIC acceleration

What sets F-NICs apart from stand-alone FPGAs is their ability to *interpose and process the network traffic* to and from the host with low overhead. For application acceleration, the application data plane can be partitioned between the F-NIC and the CPU, even for latency-sensitive fine-grain tasks.

We identify several common task categories in the server data plane that benefit from F-NIC acceleration.

Filtering. F-NICs may execute compute-intensive processing, such as per-message stateless authentication (e.g., JSON web token validation [47]), and filter invalid requests before reaching the CPU. We evaluate this example in §6.2.2.

Such filtering patterns arise in many server applications. For example, F-NICs may implement high-performance, simplified versions of popular services to accelerate common behavior (fast path), falling back to the CPU for corner cases (slow path). We show in §6.2.1 how an F-NIC-hosted key-value store cache reduces server load.

Transformation. F-NICs may convert data formats, perform (de)serialization, compression, encryption, or similar datacenter tax tasks [48]. They can change data layout, e.g., transpose matrices [35], sample, or realign data [2, 38] for efficient CPU/GPU processing or storage. As F-NICs may run a (potentially limited) network transport layer, they may speed up CPU transport layer processing [50], as we show in §6.

Transformation is often combined with filtering. For example, to accelerate the log-structured merge (LSM)-trees [22, 79], the F-NIC may store the tree's first level in its local memory, executing updates without interrupting the CPU, batching and sorting them before sending them to the host.

Steering. F-NICs may improve server performance using application-specific packet steering and inter-core load balancing [50, 89], processing complex steering policies at line-rate, e.g., using heavy-hitter approximation sketches [62].

Generation. Applications may offload the transmission of outgoing messages to multiple destinations. Examples include data replication and erasure coding in storage systems [38, 53, 77], and the shuffle stage in distributed analytics engines.

3.3 AFUs in the cloud

AFUs are custom accelerators that can be instantiated on any compatible FPGA and used via a companion software library. There are two deployment models for cloud AFUs: in the FPGA-as-a-Service (FaaS) model, tenants use their own AFUs on cloud infrastructure [3, 5, 41], whereas with the *app marketplace* model, cloud providers offer common AFUs for on-demand deployment [40].

For example, while Amazon provides FaaS, its Marketplace offers third-party AFUs [4]. Similarly, Microsoft deploys its own cloud hardware microservices [17, 24, 33]. The marketplace model opens more opportunities for better F-NIC utilization. As cloud providers develop or audit these AFUs, they can trust them to allow fine-grain sharing. By co-locating tenants that request the same AFU, cloud providers may increase their infrastructure utilization, thereby increasing power efficiency [108] and reducing costs. Pre-designed AFUs are less flexible than customer-provided AFUs, but vendors can offer them at a lower cost due to the more aggressive sharing.

NICA's design supports both deployment models.

4 Design

NICA overview. Figure 2 shows the main NICA components with a single physical AFU. NICA comprises three layers: application-visible OS abstractions and services inside a VM integrated with the network stack (§4.1); the hypervisor layer for managing F-NIC resources and QoS (§4.2); the hardware layer which includes the support for OS abstractions, physical AFU logic (pAFU), a virtualization framework exposing virtual AFUs (vAFUs), and a hardware runtime with network I/O services for application-level message processing on AFUs.

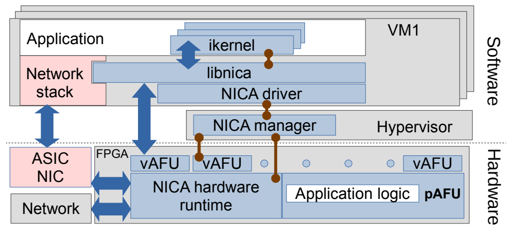


Figure 2: NICA overview. Light blue shapes are NICA components. Blue and brown lines are the data and control path respectively. vAFU: virtual AFU, pAFU: physical AFU.

4.1 Abstractions for inline acceleration

Among our primary goals is to simplify the use of inline accelerators in existing applications with minimal changes. Our abstractions thus provide a general interface for AFU management, which is compatible with standard network I/O interfaces. They allow application control of AFU execution and efficient communication between the host and the AFU.

4.1.1 The ikernel abstraction

An *ikernel* is an OS object that represents an AFU in a user program. An owner process creates an ikernel and controls it exclusively. Essentially, the ikernel extends the process abstraction into the AFU, and NICA protects the ikernel state from other CPU processes and ikernels.

To invoke an AFU, it must be associated with an active network flow. Thus, applications activate the AFU by *attaching* one or more sockets to its ikernel, thereby rerouting the associated traffic through the AFU. The ikernel stops processing the socket's traffic when the application detaches or closes the socket, keeping the AFU state intact for later invocations. Dynamic attachment adds flexibility by enabling software involvement in connection establishment and session preparation, thereby allowing applications to activate an ikernel only for specific clients or request types, for example.

The attachment semantics depends on the protocol. For a UDP socket, the ikernel receives all incoming packets destined to the socket's listening port. For TCP sockets, the effect of attachment depends on the socket state. Attaching a *connected* TCP socket migrates its state to the AFU hardware network layer. After a process attaches a listening TCP socket to an ikernel, the AFU handles new connection requests, as applications with a high connection rate may benefit from accelerating the connection establishment process. Nevertheless, NICA notifies the host network stack about new connections, off the critical path, to provide application control over these connections from the host.

A process may create several shared-nothing ikernels of the same AFU, e.g., to keep different cryptographic contexts for a crypto-AFU, but our intended usage is one ikernel per AFU per process. Multiple threads of a process may attach

Function	Purpose
`ikernel* ik_create(uuid_t, int dram_size)`	Allocate an ikernel
`void ik_destroy(ikernel*)`	Deallocate an ikernel
`int ik_attach(ikernel*, int sock)`	Attach ikernel and socket
`int ik_detach(ikernel*, int sock)`	Detach ikernel and socket
`int ik_command(ikernel*, cmd* desc)`	Invoke RPC command
`cr* ik_create_cr(ikernel*)`	Allocate a ring
`void ik_destroy_cr(cr*)`	Deallocate a ring
`int cr_post_recv/send(cr*, buf*)`	Pass buffers to the ikernel
`int cr_poll(cr*, wc*, int n)`	Poll ring for completion

Table 1: Control (top) and data plane (bottom) ikernel API.

their sockets to the same ikernel, thereby sharing the AFU state among them.

For now, a socket can be attached only to a single ikernel, but we plan to enable ikernel chaining in the future (§4.4).

Alternatives. We choose the ikernel abstraction because it captures the intuitive application-level semantics of inline network processing. We also considered using match-action rules, as in FlexNIC [50] and DPDK [84]. These are not associated with sockets, but rather with packet header rules, e.g., selecting packets of a specific five-tuple. Such an interface suits packet processing but is too low-level for application logic offloading. The ikernel socket-level abstraction hides the details of the hardware-resident network stack and allows simpler integration with existing applications.

4.1.2 Control plane

The control APIs (Table 1) allow initialization, teardown, and access to the AFU-resident application state. Under the hood, they interact with the network stack on the host and on the F-NIC to coordinate resource allocation and AFU processing.

Initialization and attachment. An `ik_create` call initializes an ikernel given an AFU's UUID. When the ikernel attaches to a socket it updates the F-NIC network stack. Once the ikernel is attached, the NICA driver tracks the socket state, detaching the flow when the socket is closed. The application may also detach an ikernel before the connection terminates.

The `ik_create` call may initialize a pre-loaded AFU or load it at runtime using partial reconfiguration. The ikernel abstraction hides the AFU hardware initialization details from the user, leaving the OS in charge of manipulating the FPGA-AFU allocation, similarly to AmorphOS [51].

Application state. Applications may access the ikernel state in the AFU. The hardware could expose the state in two ways: (1) as shared memory between the host and the F-NIC; or (2) using remote procedure calls (RPC) from the CPU to the AFU that retrieve/set the state. Shared memory might not be efficient, however. First, FPGA logic can keep frequently-accessed data in private memory, such as registers or block RAM, for efficiency. This memory is not exposed to the CPU. Further, access to the shared state requires explicit synchronization, which is costly over PCIe. Therefore, we chose the RPC model, which allows the AFU to implement arbitrary atomic transactions, including e.g., getting a snapshot of its state. Internally, NICA also uses the same mechanism to control transport layer and QoS parameters.

Error handling. An AFU that encounters an error exposes it to an ikernel runtime which periodically checks for errors via the RPC mechanism. In addition, the ikernel may abort the connection or detach itself from the respective sockets and forward packets without offloading.

4.1.3 Data plane

NICA provides two ways to perform network I/O with inline acceleration: POSIX API and *custom rings*.

POSIX networking API. After attaching an ikernel to a socket, the application may use standard I/O calls, e.g., `send`, `recv`, and `epoll_wait`, while the AFU transparently processes the data in-flight. We currently support the POSIX APIs only for UDP sockets.

Custom rings. POSIX I/O interfaces incur the overhead of extra data copies into user buffers [81] and host-side network stack processing. On the other hand, an AFU may need to exchange *application-level* messages with the host application. For example, a deserialization AFU may send ready-to-use data objects to the application. Furthermore, an AFU may need to steer the processed messages to different CPU cores, i.e., for application-aware load balancing.

NICA introduces a *custom ring*[2] (CR) abstraction that provides a zero-copy API for sending/receiving *application messages*, bypassing the host network stack. Each ikernel may create multiple associated CRs to enable message steering for multi-core systems.

The CR interfaces are similar to VIA/RDMA verbs [30] (Table 1). Specifically, each CR comprises a queue pair (QP) and a completion queue (CQ). The application allocates its communication memory buffers and registers them with the CR. It then posts the send/receive requests to the respective queue in the QP. The request completions show up in the CQ.

Custom rings vs. random access. FPGA acceleration frameworks [37, 43, 101] and some I/O intensive AFUs [29, 38, 60] allow random access to CPU memory from the AFU, which is useful for fine-grain sharing of data-structures between the CPU and the AFU. NICA currently focuses on the AFU tasks that communicate with the CPU via a streaming I/O pattern, which is much easier to implement using a producer-consumer CR interface. We leave support for random host memory access for future work.

Synchronization. In the most common application scenarios, networking or custom ring operations implicitly synchronize the CPU application and the AFU processing. In more complex cases, when the AFU accumulates the application state (e.g., for network I/O monitoring or consensus), the ikernel

[2]The hardware uses a descriptor *ring* buffer just like a regular NIC, but the buffer contents are application messages rather than raw packets.

RPC interface allows AFU developers to provide application-specific mechanisms to safely access ikernel state.

4.1.4 Usability

We expect adding ikernels to existing applications to require relatively small design or code changes. In case of filtering (see §3.2), an application may still use POSIX sockets as before, while receiving only the filtered data. For example, `memcached` requires no changes to its data processing to use the KVS cache AFU (§6). Data transformation tasks, such as deserialization, may use custom rings to obtain or send back the data in an application-friendly form. Steering applications may use per-core custom rings to get the contents directly to the correct application thread or a GPU. A generation application, e.g., replication, may send only one data copy via the custom ring, while the AFU will distribute it to pre-configured destinations.

4.2 Virtualization

To support fine-grain sharing of AFUs, as required for low latency applications, we introduce the notion of a *virtual AFU, vAFU*, which represents a single isolated hardware entity on the F-NIC. Each vAFU provides state protection and performance isolation across all the shared resources on the F-NIC. To clarify, a vAFU is a hardware entity, whereas an ikernel is an OS object that belongs to a process. Connecting multiple ikernels to the same vAFU might be possible, i.e., allowing in-VM resource allocation policy enforcement, yet we do not support it in our prototype.

One F-NIC may host multiple physical AFUs via space sharing, whereas each such AFU may support multiple vAFUs via fine-grain time sharing, as explained below. For example, our key-value-store cache AFU supports 64 vAFUs, allowing concurrent acceleration of up to 64 different `memcached` servers on the same F-NIC (§6).

Fine-grain AFU sharing. Supporting multiple vAFUs on a single physical AFU requires low-overhead hardware context switching mechanism. The vAFU context includes the ikernel state in DRAM and registers and the contexts of the sockets connected via that vAFU. Each received packet may belong to a different vAFU so slow context switch would not only increase application latency but also increase the required NICA internal buffer space.

To support fine-grain sharing, we store the vAFU context by reserving fast memory for each vAFU rather than evict/reload it to/from slow DRAM memory. Specifically, the AFU registers are replicated to store data for all concurrently active vAFU contexts. Each vAFU is associated with a hypervisor-chosen tag. The AFU switches to the context requested by the scheduler by updating the active tag register. Such a context switch can be extremely fast, e.g., up to 3 clock cycles in our prototype.

However, the number of vAFUs that can be supported is constrained due to the limited size of fast memory on the FPGA. For more vAFUs, AFUs may use DRAM to store the contexts and use latency hiding techniques, i.e., increased concurrency. Our current prototype uses fast memory, yet it is enough to host up to 64 vAFUs for the evaluated applications.

4.2.1 State protection

NICA protects the vAFU state in DRAM, fast memory, and hardware registers. For the DRAM, we use a segment-based MMU for simplicity. Similarly, we protect the control registers of the RPC interface by including a vAFU tag.

Additionally, NICA ensures correct steering of network traffic to and from the vAFU via its on-NIC network stack (§5.3). In particular, it guarantees that a vAFU will not perform network spoofing attacks toward the host and will receive only the packets destined to that vAFU. These two aspects are essential for supporting untrusted AFUs in NICA.

4.2.2 Performance isolation

NICA supports isolation of I/O channels and compute resources. The compute scheduling is necessary only among the vAFUs of the same physical AFU. The FPGA loads different physical AFUs into different partitions, and thus they do not share FPGA compute resources. DRAM bandwidth partitioning is left for future work.

I/O bandwidth sharing. The bandwidth allocation between tenants is often implemented inside a virtual switch or in the NIC internal switch. However, in a bump-in-the-wire architecture the F-NIC sends vAFU-generated messages directly to the network, bypassing these policies. Therefore, NICA provides its own bandwidth allocation mechanisms, similar to the traffic class (TC) mechanisms used in NICs [10].

To control the vAFU egress bandwidth, both towards the CPU and towards the network, we add a set of TC queues (see Figure 3). Packets are classified to these queues and scheduled. We use a work-conserving deficit round robin (DRR) scheduler [95] to allocate bandwidth, but more complex policies can be used. NICA's bandwidth scheduler is trusted and used by all the vAFUs on the F-NIC.

The vAFU recognizes when the TC queues are full and may drop the packets or propagate the contention if possible. For example, it may slow down the host by using custom ring flow control or slow down the sender through explicit congestion notification (ECN).

NICA does not manage the ingress bandwidth into the vAFU from the network or the host, as the sender (TOR or host virtual switch) already shapes ingress traffic.

AFU compute sharing. An AFU must determine which vAFU to activate at any given time, and which packets to serve first. We considered two design options: a general compute scheduler for all AFUs (similar to the I/O scheduler) or an internal AFU-specific scheduler for each AFU. These two options represent an inherent trade-off between FPGA resource consumption and design generality.

A generic scheduler in front of the vAFUs could reorder packets according to a global policy, simplifying the AFU design. However, such a scheduler requires deep input queues, therefore increasing consumption of F-NIC fast memory. Further, the need for queuing is protocol-dependent. For example, TCP has its own input queues to receive out-of-order packets, so extra scheduling queues would be wasteful. Moreover, AFUs may customize queue contents to save resources, e.g. by keeping parsed requests instead of full packets.

We thus decided to implement a custom, application-specific scheduler in each AFU.

4.3 AFU development

AFUs implement hardware interfaces to receive/transmit transport layer and custom ring data, configuration and control interfaces for RPC, and, optionally, provide vAFU scheduling and virtualization.

All the packets passing through an AFU are tagged with metadata that identifies the associated ikernel and flow, which can be used by the AFU for ikernel state isolation. The AFU receives per-TC usage levels and CR flow control (see §5.2).

While designing such FPGA hardware can be difficult, we try to simplify the development by using high-level synthesis to design our AFUs in C++, and use the `ntl` class library [32] to implement common modules such as AFU schedulers and control-plane interfaces. In addition, the NICA hardware runtime handles some common tasks such as transport processing and egress scheduling, thus simplifying AFU development.

4.4 Discussion

F-NIC transport layer. An inline AFU requires transport layer services to process data at the application layer; it may terminate flows or generate and send new messages. Our current design uses a full implementation of UDP and TCP logic in hardware. With this solution, the F-NIC effectively runs its own complete network stack.

A complete TCP/IP stack in hardware simplifies AFU development but increases F-NIC resource consumption and maintenance difficulty [75]. To eliminate NIC transmission buffers, an AFU could generate retransmissions on-demand or use host memory [85, 97]. If packet reordering is rare, an AFU may process received data only in-order, deferring out-of-order packets to the CPU [85]. A resource-efficient TCP design for inline AFUs warrants further research, so we choose a simple solution to evaluate the ikernel abstraction compatibility with TCP.

Virtual switch offloading. F-NICs intercept the inbound network traffic *before* it reaches the CPU. As a result, it becomes difficult to handle hypervisor policy and virtual networking rules, e.g., as in Open vSwitch, because they are typically handled by the hypervisor's virtual switch software running on CPU. This issue is not unique to NICA and exists with standard SR-IOV NICs [33]. Typical solutions pass the first packet to software and offload per-flow policy to hardware match-action rules [33, 59, 78]. While this may take significant area of the F-NIC's FPGA [33], future F-NIC designs may be able to harden this functionality [20, 31].

Multi-AFU support and services. Our design provides all the necessary mechanisms to run multiple AFUs on the F-NIC: packet schedulers, steering, RPC and MMU isolation modules. Currently, a single socket may only be attached to a single AFU. However, there are use cases for chaining several AFUs in a single application to accelerate various aspects of the server's traffic [16, 56, 117]. Multi-AFU chaining requires extensions to resource isolation mechanisms and software interfaces, which we plan to explore in the future.

5 Implementation

We implement NICA for the Mellanox Innova F-NIC and integrate it with the KVM/QEMU hypervisor and VMA user-space networking library [69].

5.1 AFU virtualization

NICA implements hardware virtualization of the physical AFUs, exposing virtual AFUs (vAFUs in Figure 2) to VMs. Currently, the hypervisor allocates one vAFU for each requested ikernel. NICA isolates the vAFU I/O channels in hardware and requires no software mediation.

We utilize the NIC's SR-IOV functionality to virtualize the data path (both POSIX and custom rings). SR-IOV enables unmediated overhead-free access from the guest to the NIC hardware. In general, implementing SR-IOV in custom accelerators is quite challenging, but the bump-in-the-wire architecture of our F-NIC allows reusing the existing NIC hardware SR-IOV mechanism. For the control plane, which is less sensitive to performance, NICA uses para-virtualization.

5.2 Software

We implement the NICA API in the `libnica` library. It integrates with the VMA user-space networking library, providing the POSIX socket API with kernel bypass and direct hardware access. We modify VMA to support the ikernel abstraction.

The NICA VM driver mediates between `libnica` and the hypervisor's NICA manager daemon, using a para-virtual device (virtio-serial). The NICA manager runs in the hypervisor and controls AFU hardware through the F-NIC kernel driver. NICA software stack is about 2,200 LOC.

Custom ring using RoCE. We use the F-NIC's RoCE support [105] to implement the CR, employing the ASIC NIC

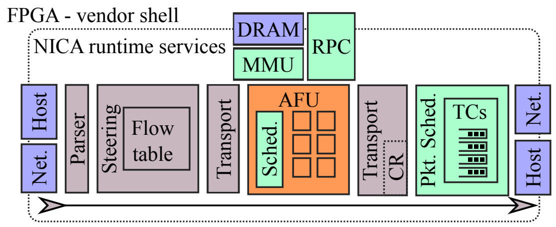


Figure 3: NICA hardware runtime (only ingress is shown, 1 AFU). Isolation modules are green. Each AFU supports multiple vAFUs. Sched.=Scheduler.

hardware and software layers using the bump-in-the-wire architecture. The implementation associates CRs with RoCE unreliable connected (UC) queue pairs (QPs). To send to a specific CR, NICA's transport layer generates RoCE packets to the host, targeting the appropriate QP. The ASIC RoCE engine writes the data directly to the application buffers, providing address translation, DMA, and completion notifications.

In our bump-in-the-wire F-NIC, the FPGA logic does not have a direct end-to-end flow control mechanism with the host, and UC does not provide such a mechanism either. Therefore, NICA adds a credit-based flow control mechanism between the AFU and the CPU application. The custom ring APIs transparently invoke this mechanism.

5.3 Hardware runtime

Figure 3 shows our FPGA processing pipeline. For clarity, we describe ingress (from network to host) only. The FPGA runtime provides the hardware support for inline programming abstractions and the essential services for inline acceleration. These include: (1) the custom rings and RPC mechanism to support efficient data and control plane primitives for ikernels; (2) a memory management unit (MMU) for memory isolation; (3) a network processing stack to support application-level processing in the AFU, which includes the parser, flow steering, and the transport layer; and (4) a virtualization layer, implementing AFU and packet schedulers.

We develop NICA and the evaluated AFUs in HLS [114] and Verilog. Table 2 shows the FPGA resources and number of code lines. NICA operates the FPGA at 216.25 MHz.

TCP/IP implementation. Our prototype includes full support for UDP and partial support for TCP. The UDP/IP service splits/combines the header and the payload. As the CR utilizes RoCE over UDP, it also uses the UDP/IP service.

The TCP implementation builds on an existing 10 Gbps FPGA TCP/IP stack [96]. Its integration with NICA is incomplete, as it lacks virtualization and socket migration support (though existing techniques apply [8, 27]). It is included primarily to validate how NICA abstractions hold with TCP.

Table 2: FPGA utilization and lines of code. LUTs: lookup tables, FFs: flip-flops, RAMB18: block RAM units.

		Area (% of total)			LOC	
	Module	LUTs	FFs	RAMB18	HLS	Verilog
System	NICA	13%	9%	13%	6643	1736
	TCP stack	6%	4%	13%	15303	1110
	Vendor shell	51%	32%	7%		
Apps	NICA-KVcache	5%	2%	2%	975	
	IoT server	10%	7%	8%	646	1627

5.4 Limitations

Our prototype may run only two physical AFUs, where one is a minimal AFU that passes through unmodified traffic. This is not a design limitation but stems from the FPGA area constraints (see Table 2). Further, NICA does not yet support virtual switch offloading, and our current CR implementation does not transmit, only receives. In addition, our F-NIC does not support partial reconfiguration. We hope the next generation of the F-NIC [31] will resolve these limitations, as it is expected to have a larger FPGA with more space and hardened network virtualization support.

NICA performance drops dramatically when crossing NUMA links. We are investigating a potential hardware bug.

6 Evaluation

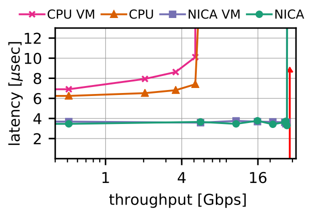
Hardware setup. We use four machines with Intel® Xeon® E5-2620 v2 2.1 GHz CPUs, connected via a Mellanox SN2100 40 Gbps switch. Three (clients) use Mellanox ConnectX®-4 Lx EN NIC, and one (server) uses a 40 Gbps Mellanox® Innova™ Flex 4 Lx EN (1st gen.) F-NIC, equipped with a Xilinx XCKU060 FPGA. The server is a dual socket NUMA machine with 64 GB RAM. Hyper-threading and power saving settings are disabled.

CPU baseline. We use VMA [69] user-level network stack with kernel bypass, optimized by Mellanox and broadly used for high-performance networking [33]. We use commodity NICs with the same ASIC as our F-NIC but without the FPGA. Due to the NUMA performance issue of the current prototype (§5.4), to allow a fair comparison, we constrain our experiments to the NUMA node closer to the NIC.

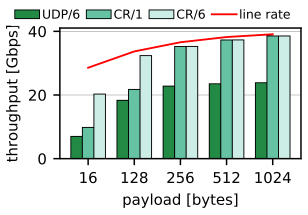
F-NIC maximum power consumption. The F-NIC consumes up to 30 W [68] vs. 14.2 W [70] for the client NICs.

Performance measurement. We use `sockperf` [71], a benchmarking tool optimized for VMA. To reliably measure performance, we use performance counters on NICA's FPGA runtime, the NIC, and the switch. We run each experiment 5 times, each 60 second long.

NICA configuration. We set a max. of 4 TCs, 64 ikernels, VMs, and custom rings, 1K UDP ports, and 10K TCP flows.



(a) Throughput vs. latency (99th-percentile) for `echo`. Vertical arrow: line rate, CPU: kernel-bypass.



(b) UDP on AFU. UDP: CPU baseline (POSIX + kernel-bypass), CR: custom ring/number of cores.

Figure 4: Microbenchmarks

6.1 Microbenchmarks

We use several microbenchmarks to evaluate the benefits of NICA acceleration through filtering and transport layer acceleration and to estimate virtualization overheads.

Experiment 1: Virtualization performance. Figure 4a shows the throughput-latency comparison of bare-metal and virtualized echo server AFU vs. the CPU baseline for 64-byte packets. We measure no overheads of the AFU virtualization.

At 5 Gbps, the latency of the virtualized AFU is 2×/2.8× lower than bare-metal/in-VM CPU server respectively. At 6.7 Gbps, the baseline latency spikes to 38 µs, while the AFU achieves up to 27.6 Gbps at 4 µs latency, above which we see packet drops. The stable low latency at high throughput is a valuable property of F-NIC accelerators.

Experiment 2: UDP performance. We run a pass-through AFU that receives UDP packets and transfers them to the host via CRs, saving the host UDP processing. The CPU baseline uses VMA for POSIX API kernel bypass, with 6 CPU cores. Figure 4b shows the throughput for different packet sizes. Offloading UDP processing to the AFU boosts the throughput from 2.9× and 1.7× for small and large packets respectively. For larger packets, a single-core CR outperforms 6-core UDP.

Experiment 3: TCP performance. We evaluate NICA's preliminary TCP support by accelerating a monitoring server microbenchmark. The server receives integers as 18-byte messages (4-byte integers with a 14-byte sockperf header) and computes their average, alerting the user when the received values are above a given threshold. With NICA, the AFU maintains the average and sends only the messages above the threshold via the custom ring (bypassing the host TCP stack).

For 6 flows from 6 clients, the AFU consumes 34.8M messages/sec, 3× faster than the baseline's 11.5M messages/sec (single core). The AFU benefits diminish as the portion of the messages sent to the host increases, down to a modest 11% throughput improvement. This indicates that *the F-NIC transport layer processing contributes much less than filtering to the overall performance benefits*.

Experiment 4: I/O isolation overheads. We evaluate the egress scheduler when using two AFUs: a traffic generator AFU and a pass-through AFU. The former generates messages to the network at maximum throughput. The latter transfers messages between the host and the network. These AFUs share the network egress I/O channel and are assigned to separate traffic classes. We set the scheduler quantum to 1 KB.

We measure the latency of a few 64-byte packets sent via the pass-through AFU while the generator AFU sends 1514-byte packets. At 38.4 Gbps load, the low-latency pass-through packets suffer a 1 µs overhead to 99th-percentile latency compared to an empty system. This result demonstrates that the *I/O isolation mechanism achieves low overhead even under heavy contention*.

6.2 Application benchmarks

We accelerate two large applications: `memcached` and a Node.js-based IoT server. We build a transparent cache AFU for the former and an authentication AFU for the latter, integrating both into the CPU software.

6.2.1 Transparent `memcached` cache

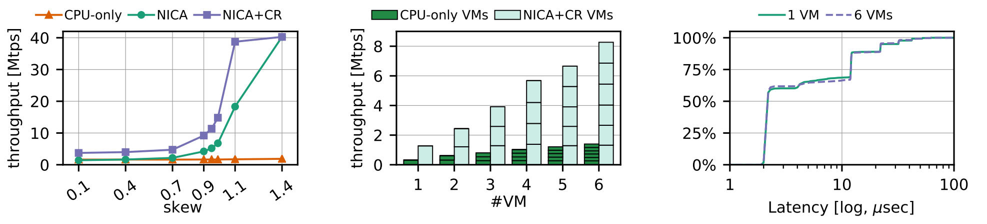
We prototype a transparent look-through cache for `memcached`, called NICA-KVcache. The AFU parses `memcached`'s ASCII UDP protocol and serves GETs directly from its F-NIC DRAM-resident cache. The AFU passes GET misses and other update requests to the host. Upon update, the AFU *invalidates* the respective cache entry. The AFU populates the cache by intercepting GET *responses* from the host, ensuring coherence even if the host drops the updates due to overload. The AFU caches keys/values of up to 16-byte and uses a direct-mapped cache for simplicity.

We implement two designs: one with POSIX API and another with CRs. The former requires changing `memcached` to instantiate the ikernel and attach sockets. The latter introduces CR polling to the `memcached` worker thread event loop. Adding the F-NIC acceleration support required 107 and 135 LOC for the POSIX API and CR versions respectively.

Workload. We initialize the CPU server with 32 M 16-byte keys and values (4 GB with overheads) and set the AFU cache to store 2 M keys per-ikernel (128 MB RAM). The CPU baseline uses an unmodified `memcached` with the VMA network stack. Clients generate a YCSB-like [25] workload with varying skew using `sockperf`.

Bare-metal performance. Figure 5a shows that for lower skews (high miss rate), the CPU (6 cores) is the bottleneck. With Zipf(0.99) distribution (YCSB's default), NICA+CR achieves 9× speedup. For 100% hit-rate, the AFU becomes network-bound (99% of 40 Gbps line-rate), resulting in 21× higher throughput than the baseline.

The cache hit-rate also dictates the latency distribution (not shown). We observe a mixture of two distributions: cache hits and cache misses. With Zipf(0.99) distribution and 1 Mtps load, the F-NIC serves cache hits at a stable 2.1 µs. Misses, served by the host, are 6 µs at the 99th-percentile, versus



(a) Bare-metal throughput, varying Zipf skew.

(b) Multi-VM scaling, Zipf(0.9) distribution.

(c) Latency CDF with virt., Zipf(0.9) distribution.

Figure 5: NICA-KVcache results, CPU+VMA (kernel-bypass) vs. NICA with/without a custom ring (CR).

Table 3: NICA-KVcache throughput [Mtps] with 0.2% SETs.

Skew	0.90	0.95	0.99	1.10
Baseline (CPU-only with kernel-bypass)	1.55	1.55	1.55	1.55
NICA with custom ring	5.98	6.51	7.10	8.28

10.5 µs in the baseline. *The latency improvement is due to the reduced CPU load as a result of filtering.*

Table 3 shows the throughput with 0.2% SETs (common in Facebook [15]). At Zipf(0.99), NICA is 4.6× faster than the baseline. With 10% SETs (not shown), CPU throughput dominates, thus NICA shows no performance improvement.

Other KVS implementations. NICA-KVcache offers significant advantages even when used with highly optimized CPU-only KVS implementations, such as MICA [63], which achieve line-rate throughput using CPU cores alone. In this case, NICA-KVcache reduces the required number of CPU cores by filtering all the cache hits and leaving only the misses to the CPU, thereby improving the overall system efficiency. More specifically, for a given hit rate in the NICA-KVcache, achieving line-rate requires the CPU throughput to be $line\_rate \cdot (1 - hit\_rate)$ transactions per second.

For example, MICA [63] reaches 5 Mtps on a single CPU core with 100% GETs for 32 M 16-byte keys and values (1GB of data) with a Zipf(0.99) distribution. Optimistically assuming perfect scaling, MICA would reach line-rate (59.5 Mtps) with 12 cores, without NICA-KVcache acceleration. In contrast, with NICA-KVcache of size 128MB, running the same Zipf(0.99) key distribution results in 75% hit-rate, thus the CPU only handles 14.9 Mtps, utilizing just 3 CPU cores.

This result demonstrates that the use of NICA for accelerating key-value stores is cost-effective, considering that a single CPU core is reportedly more expensive than a SmartNIC [33].

Accelerated KVS. Floem [83] implemented a similar key-value store cache on a Cavium SmartNIC and reported a 3.6× performance improvement with 100% hit-rate with write-back, and no benefits for 10% SETs write-through, as in NICA. Rather than `memcached`, Floem required a custom KVS server, however. KV-Direct [60] with small requests achieves comparable performance to NICA (with 100% hits) but reaches 180 Mtps using client-side batching. Unlike NICA-KVcache, its data-plane is fully implemented in hardware, and it only uses the host for slab allocation.

Contribution of network-stack processing. Figure 5a shows that using CR for low cache hit rates results in 2.2× speedup over the CPU baseline. In this case, the use of CR eliminates the network stack processing on the host but keeps the application processing on the CPU. Naturally, higher hit rates result in a higher portion of the requests handled by the AFU, and much higher speedups. This experiment suggests that *the network stack offloading alone is not enough to reach the full performance potential of the F-NIC acceleration.*

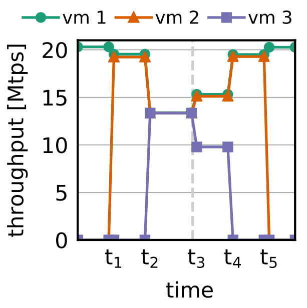
Virtualization performance. We evaluate the performance with a varying number of VMs. Each VM uses 5 GB of server RAM, 1 dedicated CPU core, a vAFU, and 2 M keys worth of vAFU cache. For Zipf(0.9), Figure 5b shows near-linear scaling, consistently achieving a 5.6× speedup over the CPU. Further, we observe no measurable negative impact of virtualization on vAFU latency. The system achieves similar results with 64 M keys per VM, utilizing most of the 64 GB RAM of our machine.

Figure 5c shows the latency distribution of a single VM and 6 VMs executions under 1.3 Mtps load, for a Zipf(0.90) workload. The latency increases for the top 40% of the requests, which matches the expected hit-rate. We observe that the VM CPU latency is much higher than the bare-metal latency reported above, but cache hits are served at the same latency with and without virtualization.

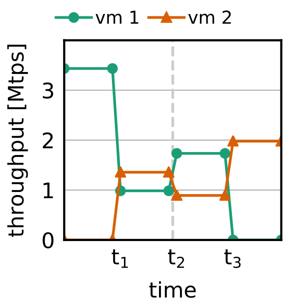
This experiment confirms that *AFU fine-grain sharing is feasible and effective.*

Network bandwidth isolation. We use 3 VMs, associated with 3 TCs, and initially configure them to share the egress bandwidth equally. We use a Zipf(1.4) distribution (99.9% hit-rate), and a 20 Mtps load on each VM, to stress the scheduler.

Figure 6a shows the throughput of each VM over time. At first, only VM 1 is active, using the whole AFU. When VM 2's clients start, the combined egress throughput is barely above the AFU's maximum (39 Mtps), and the clients process 19 Mtps each. When VM 3's clients start, the combined throughput surpasses the maximum, and the scheduler divides the bandwidth equally (13 Mtps per VM). At t_3, we change the bandwidth allocation to 40%/40%/20% and observe an asym-



(a) NICA-KVcache w. egress bandwidth management.



(b) IoT authentication throughput w. internal scheduler.

Figure 6: QoS experiments

Table 4: Node.js goodput (valid req. received) under DoS.

Valid packet ratio	40%	60%	80%	100%
Baseline (req/sec)	1489	2294	3131	3960
NICA (req/sec)	5165 (3.4×)	5165 (2.3×)	5231 (1.6×)	5181 (1.3×)

metric allocation. This confirms the *NICA egress isolation is successful in allocating bandwidth among the tenants*.

6.2.2 IoT authentication

We prototype an IoT monitoring server using Node.js with JSON web token (JWT) stateless authentication. The JavaScript-based server exposes an endpoint to which IoT devices publish their measurement using the CoAP protocol [12], similarly to the SAMSUNG Artik IoT cloud API [92]. The payload of each request contains an authentication token, which includes the device ID and a timestamp, and signed using HMAC-SHA256. Invalid requests are discarded.

Our prototype authentication AFU parses received packets, extracts the token, verifies the signatures (using a SHA-256 accelerator [99]) and drops requests with invalid tokens. Valid requests are passed to the CPU and only undergo token expiration check there.

We evaluate our IoT authentication accelerator against a software-only Node.js server. Adding NICA support using POSIX APIs required 20 JavaScript LOC and 34 lines for the libnica generic Node.js module, demonstrating *the simplicity of integrating the ikernel abstraction with complex software*.

In this experiment, we simulate a Denial of Service (DoS) attack by sending a varying number of invalid tokens with incorrect signatures in the input stream. Table 4 shows the goodput, in requests/sec, as a function of the valid packet ratio. While the baseline degrades linearly, NICA maintains a constant goodput by filtering the invalid packets.

One may wonder whether optimizing the Node.js server (e.g., rewriting it in C) would diminish the AFU acceleration benefits. We argue that this is not the case. The AFU hardware achieves the throughput of 3.5 Mtps, about 3 orders of magnitude higher than the software throughput. As long as the rest of the CPU processing pipeline remains the bottleneck, the AFU remains effective. Additionally, compute acceleration alone results in only 30% speedup. The remaining speedup is due to filtering invalid packets, which would be helpful in the CPU-optimized version too.

AFU compute sharing. The AFU's throughput can be bounded by its SHA-256 hashing units and depends on the input JWT token sizes. To fairly share the hashing units among vAFUs, we introduce a custom DRR scheduler (§4.2.2)) that controls the per-VM utilization of the AFU hashing units.

We use 2 VMs to demonstrate the performance isolation. Clients send 10 Mtps of invalid requests to each VM, but VM 2 receives requests with 40% larger tokens. We start the experiment with the scheduler disabled and enable it mid-run.

Figure 6b shows the throughput of each VM over time. At first, only VM 1 clients are active, allowing the AFU to process at max speed (3.5 Mtps). When VM 2 begins receiving at t_1, VM 1 processes only 28% of the requests, which is below its fair share. With the scheduler enabled, at t_2, both VMs receive half of their respective maximum throughput. We observe that *NICA's compute performance isolation is essential to allow sharing of compute-bound AFUs*.

7 Related work

NIC-based acceleration. Commodity NICs have been offering network stack offloads ranging from checksum calculations, segmentation, and receive-side-scaling (RSS) to RDMA [9, 82, 87, 105, 110] and TCP offload engines [75]. Such offloads are limited to network and transport layer processing, while NICA focuses on the application layer.

Our work builds upon previous attempts to accelerate general purpose applications through inline processing in SmartNICs. Early work on Network Processing Units (NPUs) [1, 113] programming abstractions [16, 56] has shown the potential of customizing the I/O path for applications. More recently, FlexNIC [50] has proposed an RMT-based [14] NIC for inline acceleration of application packet processing, showing how to leverage RMT hardware for application acceleration. Floem [83] aids design of NPU accelerated applications. sPIN [38] offers inline acceleration of high-performance computing (HPC) tasks such as tag-matching, data transformation, or replication, but the Portals 4 host abstraction is unsuitable for socket applications.

While we also consider inline acceleration, our goals, design, platform, and evaluation methodology are different. FlexNIC focuses on applications of SmartNIC RMT acceleration, whereas NICA offers convenient OS abstractions for integrating inline accelerators into user applications. FlexNIC targets RMT SmartNICs with constrained functionality, whereas NICA targets more flexible bump-in-the-wire FPGAs. These may run large parts of application logic, necessitating more expressive interfaces for state and execution management, such as host-NIC network stack interaction. As RMT devices are designed to work at line-rate, performance isolation of concurrent application pipelines is unnecessary; conversely, we show that QoS support is essential to expose F-NICs in cloud systems.

Packet processing frameworks such as DPDK and eBPF-XDP [39] include inline acceleration mechanisms, e.g., for cryptographic protocols such as IPSec [84] or offloading eBPF programs to SmartNICs [52]. However, these target system-wide packet processing tasks, so they lack a transport layer, network stack integration, and multiple application support.

Linux also supports attaching eBPF programs to sockets [26], similarly to ikernels, to perform inline packet processing. However, such programs cannot process transmitted packets or generate new ones, and use a POSIX API data-path, whereas ikernels enable zero-copy application messaging.

C-CORE [56] proposes the stream handlers abstraction for inline processing, but unlike ikernels, they provide no virtualization mechanisms. Streamline [16] is an OS subsystem for tailoring application I/O path that uses UNIX pipes as an abstraction, but it does not allow dynamic attachment and configuration of filters.

Some F-NIC vendors have proprietary APIs for inline application development. Solarflare AOE allows low latency TCP transmission [97] from an F-NIC. Unlike NICA, it only offloads transmissions. Maxeler MPC-N supports inline UDP/TCP application acceleration [7]. All the above lack virtualization support, and their proprietary host application abstractions are too hardware specific.

SmartNIC applications. Eden [6] and AccelNet [33] accelerate network functions on data-center end-nodes with SmartNICs. However, these are loosely coupled with host applications, whereas NICA's model couples the AFU logic with the host server logic.

Hardware accelerators for Network Function Virtualization (NFV) [18, 34, 117] target the NFV domain and hence do not provide abstractions for general purpose applications, lack host-accelerator network stack integration provided by ikernels, and provide no I/O path virtualization to/from the accelerator.

Several works have accelerated specific applications on F-NICs [24, 57, 60, 64, 106, 107]. NICA provides an infrastructure for building such AFUs in the clouds.

Languages for SmartNIC AFU development. P4 [13] is a DSL for implementing network functions with implementations for FPGAs [100, 111]. The Click [55] router has been ported to F-NICs [61, 90]. Emu [102] enables the development of network functions on NetFPGA using HLS. These can be used to simplify AFU development for NICA, but do not provide application-level abstractions.

Floem [83] is a DSL for NPU-accelerated applications. However, it requires refactoring applications to its DSL, while ikernel abstraction is less intrusive.

FPGA virtualization and sharing. AmorphOS [51] improves FPGA utilization by sharing an FPGA among multiple AFUs, and dynamically switching AFUs. Its hull isolates different AFUs used by different applications. We apply similar mechanisms to F-NIC. However, AmorphOS does not isolate FPGA network interfaces, and its context switching mechanism is not suitable for latency-sensitive networking applications.

Multes [44] shares an FPGA among tenants using a single pipeline. AccelNet [33] allows flow-context switching on a packet-by-packet basis. NICA's fine-grained time-sharing design is similar, but its goal is to virtualize inline accelerators for application layer, rather than a standalone FPGA application or cloud network/transport layers.

Remote/distributed FPGA frameworks [7, 19, 104] share FPGAs over the network with a remote CPU. Other have virtualized local look-aside accelerators [23, 36, 101, 118]. In contrast, NICA virtualizes local inline networking AFUs.

Standalone FPGAs, GPUs, or switches. Our choice of FPGA-based SmartNICs has been motivated by prior works on accelerating networking applications [11, 21, 45, 76, 103, 109]. Unlike NICA, they focus on standalone FPGAs.

Other inline acceleration techniques let GPU kernels control communication using GPU-centric networking abstractions [28, 54, 58, 74], or process data in transit on programmable switches or network accelerators [46, 62, 65, 93]. Conversely, NICA provides tighter integration of server software and AFUs. This simplifies integration with legacy programs and makes acceleration transparent for clients.

8 Conclusion and future work

As F-NICs are becoming common in data centers, new use cases for application layer inline acceleration are starting to emerge. NICA provides the ikernel OS abstraction to easily integrate F-NIC-based accelerators into applications and introduces virtualization mechanisms to share them securely and fairly in cloud systems. NICA's real-world prototype demonstrates the significant performance potential for inline acceleration of virtualized server systems, with minimal software development effort.

We believe NICA's inline abstractions are suitable beyond F-NICs and plan to investigate their use in CPU-FPGA systems and non-FPGA SmartNICs. NICA raises a range of research topics, such as distributed heterogeneous architectures, accelerator chaining, and reliable transport offloading, which we will explore in the future.

Acknowledgments

We thank Chris Rossbach, Michael Swift, Ada Gavrilovska, Aleksandar Dragojevic, and our shepherd Scott Rixner for their valuable feedback. We also gratefully acknowledge the support of the Israel Science Foundation (grant No. 1027/18), the Israeli Innovation Authority Hiper Consortium, the Technion Hiroshi Fujiwara Cybersecurity center, as well as Mellanox hardware donations and technical support.

References

[1] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson. The next generation of Intel IXP network processors. *Intel Technology Journal*, 6(3):6–18, 2002.

[2] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck. Rhythm: harnessing data parallel hardware for server workloads. In *ASPLOS '14*. ACM, 2014, pp. 19–34.

[3] Alibaba Cloud. Instance type families: f1, compute optimized type family with FPGA. (Accessed: Jan. 2019). URL: https://www.alibabacloud.com/help/doc-detail/25378.htm%5C#f1.

[4] Amazon. AWS Marketplace – F1 search results. (Accessed: Dec. 2018). URL: https://aws.amazon.com/marketplace/search/results?x=0&y=0&searchTerms=F1&page=1&ref_=nav_search_box.

[5] Amazon Web Services. Amazon EC2 F1 instances. (Accessed: Jan. 2019). 2016. URL: https://aws.amazon.com/ec2/instance-types/f1/.

[6] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea. Enabling end-host network functions. In *SIGCOMM '15*. ACM, 2015, pp. 493–507.

[7] T. Becker, O. Mencer, S. Weston, and G. Gaydadjiev. Maxeler data-flow in computational finance. In, *FPGA Based Accelerators for Financial Applications*, pp. 243–266. Springer, 2015.

[8] M. Bernaschi, F. Casadei, and P. Tassotti. SockMi: a solution for migrating TCP/IP connections. In *PDP 2007*, Feb. 2007, pp. 221–228.

[9] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak. Intel® Omni-Path Architecture: enabling scalable, high performance fabrics. In *HOTI 2015*, Aug. 2015, pp. 1–9.

[10] D. L. Black, Z. Wang, M. A. Carlson, W. Weiss, E. B. Davies, and S. L. Blake. An Architecture for Differentiated Services. RFC 2475. Dec. 1998. URL: https://rfc-editor.org/rfc/rfc2475.txt.

[11] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bär, and Z. István. Achieving 10Gbps line-rate key-value stores with FPGAs. In *HotCloud'13*. USENIX, 2013.

[12] C. Bormann, A. P. Castellani, and Z. Shelby. CoAP: an application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2):62–67, Mar. 2012.

[13] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: programming protocol-independent packet processors. *ACM SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[14] 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. In *SIGCOMM '13*. ACM, 2013, pp. 99–110.

[15] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the social graph. In *USENIX ATC 2013*. USENIX, 2013, pp. 49–60.

[16] W. d. Bruijn, H. Bos, and H. Bal. Application-tailored I/O with Streamline. *ACM Trans. Comput. Syst.*, 29(2):6:1–6:33, May 2011.

[17] D. Burger. Microsoft unveils Project Brainwave for real-time AI. (Accessed: Sep. 2018). 2017. URL: https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/.

[18] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow. FPGAs in the cloud: booting virtualized hardware accelerators with OpenStack. In *FCCM 2014*, May 2014, pp. 109–116.

[19] 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 *MICRO-49*. IEEE Computer Society, Oct. 2016.

[20] A. Caulfield, P. Costa, and M. Ghobadi. Beyond SmartNICs: towards a fully programmable cloud. In *HPSR 2018*, June 2018.

[21] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala. An FPGA memcached appliance. In *FPGA '13*. ACM, 2013, pp. 245–254.

[22] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4, 2008.

[23] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang. Enabling FPGAs in the cloud. In *CF '14*. ACM, 2014, 3:1–3:10.

[24] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. E. Husseini, T. Juhasz, K. Kagi, R. Kovvuri, S. Lanka, F. v. Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Y. Xiao, D. Zhang, R. Zhao, and D. Burger. Serving DNNs in real time at datacenter scale with project Brainwave. *IEEE Micro*, 38(2):8–20, Mar. 2018.

[25] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC '10*. ACM, 2010, pp. 143–154.

[26] J. Corbet. Attaching eBPF programs to sockets. (Accessed: Jan. 2019). 2014. URL: https://lwn.net/Articles/625224/.

[27] J. Corbet. TCP connection repair. (Accessed: Jan. 2019). 2012. URL: https://lwn.net/Articles/495304/.

[28] F. Daoud, A. Watad, and M. Silberstein. GPUrdma: GPU-side library for high performance networking from GPU kernels. In *ROSS '16*. ACM, 2016, 6:1–6:8.

[29] A. Dragojević. The configurable cloud: accelerating hyperscale datacenter services with FPGAs. Presented at MARS'17. (Accessed: Jan. 2019). 2017. URL: https://sites.google.com/site/mars2017eurosys/Program/keynotes/MARS%20alekd%20shared.pdf.

[30] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. *IEEE Micro*, 18(2):66–76, Mar. 1998.

[31] H. Eran, D. Levi, L. Liss, and M. Silberstein. NFV acceleration: the role of the NIC. In *SFMA'18*, 2018.

[32] H. Eran, L. Zeno, Z. István, and M. Silberstein. Design patterns for code reuse in HLS packet processing pipelines. In *FCCM '19*. IEEE Computer Society, 2019.

[33] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *NSDI '18*. USENIX Association, 2018, pp. 51–66.

[34] X. Ge, Y. Liu, D. H. Du, L. Zhang, H. Guan, J. Chen, Y. Zhao, and X. Hu. OpenANFV: accelerating network function virtualization with a consolidated framework in openstack. *ACM SIGCOMM Comput. Commun. Rev.*, 44(4):353–354, Aug. 2014.

[35] J. Gomez-Luna, I.-J. Sung, L.-W. Chang, J. M. González-Linares, N. Guil, and W.-M. W. Hwu. In-place matrix transposition on GPUs. *IEEE Trans. Parallel Distrib. Syst.*, 27(3):776–788, 2016.

[36] L. Gong and X. Zeng. Virtio-crypto: a new framework of cryptography virtio device. KVM Forum. (Accessed: Jan. 2019). 2017. URL: http://events17.linuxfoundation.org/sites/events/files/slides/Introduction%20of%20virtio%20crypto%20device.pdf.

[37] G. Guidi, E. Reggiani, L. D. Tucci, G. Durelli, M. Blott, and M. D. Santambrogio. On how to improve FPGA-based systems design productivity via SDAccel. In *IPDPS Workshops 2016*, May 2016, pp. 247–252.

[38] T. Hoefler, S. D. Girolamo, K. Taranov, R. E. Grant, and R. Brightwell. sPIN: High-performance streaming Processing in the Network. In *SC 2017*, Nov. 2017.

[39] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. The eXpress Data Path: fast programmable packet processing in the operating system kernel. In *CoNEXT '18*. ACM, 2018, pp. 54–66.

[40] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong. Programming and runtime support to blaze fpga accelerator deployment at datacenter scale. In *SoCC '16*. ACM, 2016, pp. 456–469.

[41] Huawei Cloud. FPGA-accelerated cloud server. (Accessed: Jan. 2019). URL: https://www.huaweicloud.com/en-us/product/fcs.html.

[42] Intel. Accelerator functional unit (AFU) developer's guide. (Accessed: Sep. 2018). 2018. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-afu-dev-v1-1.pdf.

[43] Intel. Intel FPGA SDK for OpenCL programming guide. (Accessed: Sep. 2018). 2018. URL: https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html.

[44] Z. István, G. Alonso, and A. Singla. Providing multi-tenant services with FPGAs: case study on a key-value store. In *FPL 2018*, Aug. 2018, pp. 119–1195.

[45] Z. István, D. Sidler, G. Alonso, and M. Vukolic. Consensus in a box: inexpensive coordination in hardware. In *NSDI '16*. USENIX Association, 2016, pp. 425–438.

[46] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. Netcache: balancing key-value stores with fast in-network caching. In *SOSP '17*. ACM, 2017, pp. 121–136.

[47] M. Jones, J. Bradley, and N. Sakimura. JSON Web Token (JWT). RFC 7519. May 2015. URL: https://rfc-editor.org/rfc/rfc7519.txt.

[48] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *ISCA '15*. ACM, 2015, pp. 158–169.

[49] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *USENIX ATC 2011*. USENIX Association, 2011, pp. 2–2.

[50] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. High performance packet processing with FlexNIC. In *ASPLOS '16*. ACM, 2016, pp. 67–81.

[51] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with AmorphOS. In *OSDI 2018*. USENIX Association, Oct. 2018.

[52] J. Kicinski and N. Viljoen. eBPF hardware offload to SmartNICs: cls_bpf and XDP. In *Netdev 1.2*, 2016.

[53] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan. Hyperloop: group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems. In *SIGCOMM '18*. ACM, 2018, pp. 297–312.

[54] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein. GPUnet: networking abstractions for GPU programs. In *OSDI 2014*. USENIX Association, Oct. 2014, pp. 201–216.

[55] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.

[56] S. Kumar, A. Gavrilovska, K. Schwan, and S. Sundaragopalan. C-CORE: using communication cores for high performance network services. In *NCA 2005*, July 2005, pp. 171–178.

[57] M. Lavasani, H. Angepat, and D. Chiou. An FPGA-based in-line accelerator for memcached. *IEEE Comput. Archit. Lett.*, 13(2):57–60, July 2014.

[58] M. LeBeane, K. Hamidouche, B. Benton, M. Breternitz, S. K. Reinhardt, and L. K. John. GPU triggered networking for intra-kernel communications. In *SC '17*. ACM, 2017, 22:1–22:12.

[59] I. Lesokhin, H. Eran, and O. Gerlitz. Flow-based tunneling for SR-IOV using switchdev API. In *Netdev 1.1*, Feb. 2016.

[60] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. KV-Direct: high-performance in-memory key-value store with programmable NIC. In *SOSP '17*. ACM, 2017, pp. 137–152.

[61] 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 *SIGCOMM '16*. ACM, 2016, pp. 1–14.

[62] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with SwitchKV. In *NSDI '16*. USENIX Association, 2016, pp. 31–44.

[63] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: a holistic approach to fast in-memory key-value storage. In *NSDI '14*. USENIX Association, 2014, pp. 429–444.

[64] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin servers with smart pipes: designing SoC accelerators for memcached. In *ISCA '13*. ACM, 2013, pp. 36–47.

[65] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. IncBricks: toward in-network computation with an in-network cache. In *ASPLOS '17*. ACM, 2017, pp. 795–809.

[66] L. L. Luo. Towards converged SmartNIC architecture for bare metal & public clouds. APNet 2018 (Accessed: Jan. 2019). 2018. URL: https://conferences.sigcomm.org/events/apnet2018/slides/larry.pdf.

[67] G. Martin and G. Smith. High-level synthesis: past, present, and future. *IEEE Des. Test. Comput.*, 26(4):18–25, 2009.

[68] Mellanox Technologies. Innova Flex 4 Lx EN adapter card product brief. (Accessed: Jan. 2019). 2017. URL: https://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova_Flex4_Lx_EN.pdf.

[69] Mellanox Technologies. libvma: Linux user-space library for network socket acceleration based on RDMA compatible network adaptors. (Accessed: Jan. 2019). 2018. URL: https://github.com/Mellanox/libvma.

[70] Mellanox Technologies. Mellanox Technologies ConnectX®-4 Lx single 40/50 Gb/s Ethernet QSFP28 port adapter card user manual. (Accessed: Jan. 2019). URL: https://www.mellanox.com/related-docs/user_manuals/ConnectX-4_Lx_Single_40_50_Gbs_Ethernet_QSFP28_Port_Adapter_Card_User_Manual.pdf.

[71] Mellanox Technologies. sockperf: network benchmarking utility. (Accessed: Jan. 2019). 2018. URL: https://github.com/Mellanox/sockperf.

[72] Mellanox Technologies. Whitepaper: Mellanox Innova IPSec: achieve groundbreaking security for VPN, data privacy & data-in-motion, while reducing total cost of ownership (TCO). (Accessed: Jan. 2019). 2018. URL: https://www.mellanox.com/related-docs/whitepapers/WP_Innova_IPsec.pdf.

[73] K. Menychtas, K. Shen, and M. L. Scott. Disengaged scheduling for fair, protected access to fast computational accelerators. In *ASPLOS '14*. ACM, 2014, pp. 301–316.

[74] C. Min, W. Kang, M. Kumar, S. Kashyap, S. Maass, H. Jo, and T. Kim. Solros: a data-centric operating system architecture for heterogeneous computing. In *EuroSys '18*. ACM, 2018, 36:1–36:15.

[75] J. C. Mogul. TCP offload is a dumb idea whose time has come. In *HOTOS'03*. USENIX Association, 2003, pp. 5–5.

[76] R. Müller and K. Eguro. FPGA-accelerated deserialization of object structures. Tech. rep. MSR-TR-2009-126. Microsoft Research Redmond, 2009.

[77] R. Nakhjavani and J. Zhu. A case for common-case: on FPGA acceleration of erasure coding. In *FCCM 2017*, Apr. 2017, pp. 81–81.

[78] Netronome. Agilio OVS firewall software. (Accessed: Jan. 2019). 2017. URL: https://www.netronome.com/media/documents/PB_Agilio_OVS_FW_SW.pdf.

[79] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[80] A. Pant, K. Siva, and N. Tan. IPSec Acceleration: securing your data across the data center. Oracle Open World (Accessed: Jan. 2019). 2017. URL: https://static.rainfocus.com/oracle/oow17/sess/1502318673168001SKY0/PF/OOW%20Technical%20Session%20Final%20100217_1507049724149001WUcf.pdf.

[81] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: the operating system is the control plane. In *OSDI 2014*. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-peter_simon.pdf, Oct. 2014, pp. 1–16.

[82] G. F. Pfister. An introduction to the InfiniBand™ architecture. In, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, part 42. John Wiley & Sons, Inc., 1st ed., 2001.

[83] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson. Floem: a programming system for NIC-accelerated network applications. In *OSDI 2018*. USENIX Association, Oct. 2018, pp. 663–679.

[84] B. Pismenny, D. Doherty, and H. Agrawal. rte_security: enabling hardware acceleration of security protocols. DPDK Summit Userspace. (Accessed: Jan. 2019). 2017. URL: https://dpdksummit.com/Archive/pdf/2017Userspace/DPDK-Userspace2017-Day1-9-security-presentation.pdf.

[85] B. Pismenny, I. Lesokhin, L. Liss, and H. Eran. TLS offload to network devices. In *Netdev 1.2*, 2016.

[86] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *Commun. ACM*, 59(11):114–122, Oct. 2016.

[87] R. J. Recio, P. R. Culley, D. Garcia, B. Metzler, and J. Hilland. A Remote Direct Memory Access Protocol Specification. RFC 5040. Oct. 2007. URL: https://rfc-editor.org/rfc/rfc5040.txt.

[88] Y. Ren. High performance cloud with hardware acceleration. APNet 2018 (Accessed: Sep. 2018). 2018. URL: https://conferences.sigcomm.org/events/apnet2018/slides/yong.pdf.

[89] D. Riddoch and S. Pope. FPGA augmented ASICs: the time has come. In *HCS*, Aug. 2012, pp. 1–44.

[90] T. Rinta-aho, M. Karlstedt, and M. P. Desai. The Click2NetFPGA toolchain. In *USENIX ATC 2012*. USENIX, 2012, pp. 77–88.

[91] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *SOSP '11*. ACM, 2011, pp. 233–248.

[92] SAMSUNG. Samsung ARTIK cloud developer – CoAP. (Accessed: Sep. 2018). 2018. URL: https://developer.artik.cloud/documentation/data-management/coap.html.

[93] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network computation is a dumb idea whose time has come. In *HotNets-XVI*. ACM, 2017, pp. 150–156.

[94] Selectel. FPGA-accelerators go into the clouds [russian]. (Accessed: Jan. 2019). 2018. URL: https://blog.selectel.ru/fpga-uskoriteli-uxodyat-v-oblaka/.

[95] M. Shreedhar and G. Varghese. Efficient fair queuing using deficit round-robin. *IEEE/ACM Trans. Netw.*, 4(3):375–385, June 1996.

[96] D. Sidler, Z. István, and G. Alonso. Low-latency TCP/IP stack for data center applications. In *FPL 2016*, Aug. 2016, pp. 1–4.

[97] Solarflare Communications, Inc. Application nanosecond TCP send (ANTS): from request to response in less than 250ns. (Accessed: Jan. 2019). 2015. URL: https://www.solarflare.com/Media/Default/PDFs/SF-114903-CD-LATEST-Solarflare_Application_Nanosecond_TCP_Send_Paper.pdf.

[98] S. Stanley. Ubiquitous SDN acceleration is coming. (Accessed: Jan. 2019). 2017. URL: https://www.lightreading.com/carrier-sdn/ubiquitous-sdn-acceleration-is-coming/a/d-id/738209.

[99] J. Strömbergson. Secworks/sha256: hardware implementation of the SHA-256 cryptographic hash function. (Accessed: Jan. 2019). 2018. URL: https://github.com/secworks/sha256.

[100] H. Stubbe. P4 compiler & interpreter: a survey. *Future Internet (FI) and Innovative Internet Technologies and Mobile Communication (IITM)*, 47, 2017.

[101] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. CAPI: a coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7:1–7:7, Jan. 2015.

[102] N. Sultana, S. Galea, D. Greaves, M. Wojcik, J. Shipton, R. Clegg, L. Mai, P. Bressana, R. Soulé, R. Mortier, P. Costa, P. Pietzuch, J. Crowcroft, A. W. Moore, and N. Zilberman. Emu: rapid prototyping of networking services. In *USENIX ATC 2017*. USENIX Association, 2017, pp. 459–471.

[103] S. Tanaka and C. Kozyrakis. High performance hardware-accelerated flash key-value store. In *NVMW 2014*, 2014.

[104] N. Tarafdar, N. Eskandari, V. Sharma, C. Lo, and P. Chow. Galapagos: a full stack approach to FPGA integration in the cloud. *IEEE Micro*, 38(6):18–24, Nov. 2018.

[105] The RoCE Initiative. RoCE introduction. (Accessed: Jan. 2019). 2016. URL: http://www.roceinitiative.org/roce-introduction/.

[106] Y. Tokusashi and H. Matsutani. A multilevel NOSQL cache design combining in-NIC and in-kernel caches. In *HOTI 2016*, Aug. 2016, pp. 60–67.

[107] Y. Tokusashi, H. Matsutani, and N. Zilberman. LaKe: the power of in-network computing. In *ReConFig'18*, Dec. 2018, pp. 1–8.

[108] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman. The case for in-network computing on demand. In *EuroSys '19*. ACM, 2019, 21:1–21:16.

[109] D. Tong and V. Prasanna. High throughput sketch based online heavy hitter detection on FPGA. *SIGARCH Comput. Archit. News*, 43(4):70–75, Apr. 2016.

[110] A. Trivedi. Remote Direct Memory Access (RDMA) 101 – quick history lesson and introduction. (Accessed: Sep. 2018). 2011. URL: http://0x8086.blogspot.com/2011/11/remote-direct-memory-access-rdma-101.html.

[111] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon. P4FPGA: a rapid prototyping framework for P4. In *SOSR 2017*. ACM, 2017, pp. 122–135.

[112] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Quality of service support for fine-grained sharing on GPUs. In *ISCA '17*. ACM, 2017, pp. 269–281.

[113] P. Willmann, H.-y. Kim, S. Rixner, and V. S. Pai. An efficient programmable 10 gigabit Ethernet network interface card. In *HPCA-11*, Feb. 2005, pp. 96–107.

[114] Xilinx Inc. Vivado high-level synthesis. (Accessed: Jan. 2019). 2018. URL: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[115] W. Xu. Hardware acceleration over NFV in China Mobile. OPNFV Plugfest. (Accessed: Jan. 2019). June 2018. URL: https://wiki.opnfv.org/download/attachments/20745096/opnfv_Acc.pdf.

[116] T. T. Yeh, A. Sabne, P. Sakdhnagool, R. Eigenmann, and T. G. Rogers. Pagoda: fine-grained GPU resource virtualization for narrow tasks. In *PPoPP '17*. ACM, 2017, pp. 221–234.

[117] K. Zhang, B. He, J. Hu, Z. Wang, B. Hua, J. Meng, and L. Yang. G-NET: effective GPU sharing in NFV systems. In *NSDI '18*. USENIX Association, 2018, pp. 187–200.

[118] Q. Zhao, M. Iida, and T. Sueyoshi. A study of FPGA virtualization and accelerator scheduling. In *ETCD'17*. ACM, 2017, 3:1–3:4.

[119] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: toward 100 Gbps as research commodity. *IEEE Micro*, 34(5):32–41, Sept. 2014.