We Need Kernel Interposition over the Network Dataplane

Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S. Berger,[‡•]
James C. Hoe, Aurojit Panda,[†] Justine Sherry
Carnegie Mellon University [‡] Microsoft Research [•] University of Washington [†] New York University

Abstract

Kernel-bypass networking, which allows applications to circumvent the kernel and interface directly with NIC hardware, is one of the main tools for improving application network performance. However, allowing applications to circumvent the kernel makes it impossible to use tools (e.g., `tcpdump`*) or impose policies (e.g., QoS and filters) that need to interpose on traffic sent by different applications running on a host. This makes* maintainability and manageability *a challenge for kernel-bypass applications. In response, we propose Kernel On-Path Interposition (KOPI), in which traditional kernel dataplane functionality is retained but implemented in a fully programmable SmartNIC. We hypothesize that KOPI can support the same tools and policies as the kernel stack while retaining the performance benefits of kernel bypass.*

ACM Reference Format:
Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S. Berger, James C. Hoe, Aurojit Panda, Justine Sherry. 2021. We Need Kernel Interposition over the Network Dataplane. In *Workshop on Hot Topics in Operating Systems (HotOS '21), May 31-June 2, 2021, Ann Arbor, MI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3458336.3465281

1 Introduction

Network throughput and latency dictates the performance of many applications including web servers, big data engines, and deep learning frameworks. While network line rates are growing rapidly, the OS software stack has emerged as a bottleneck when accessing the network. Consequently, kernel-bypass architectures are gaining popularity as a means to 'speed up' network access [12, 21, 38, 56]. This move to kernel-bypass architectures is not limited to the network interfaces alone, and other I/O devices, such as disks, have moved in the same direction for much the same reason [9, 21, 27, 47]. The key idea behind kernel bypass is to allow applications to interface *directly* with I/O devices and hence avoid the overheads of context switching and copying data between the kernel and applications [7, 25, 38]. Kernel bypass designs are quickly becoming the *de rigeur* approach to designing high throughput, low-latency networked systems [14, 22, 36, 40].

Unfortunately, kernel bypass architectures have brought a maintenance and manageability nightmare for administrators. For example, system admins are accustomed to setting security and QoS policies like, 'only application A can send packets on port 22' or 'application A has priority access to the network over application B'; when applications are given raw I/O access the administrator can no longer enforce such policies. Similarly, developers are used to debugging networked applications by inspecting traffic traces intercepted in the kernel using `tcpdump`; a kernel-bypass approach means that interception can only be performed within the application (which is not very helpful when the question at hand is *which* application is acting up or misbehaving in the first place). In the absence of mechanisms to enforce these policies administrators must rely on ad-hoc and kludgy solutions (such as deploying one virtual machine per application) which have high costs, additional performance overheads, and management complexity.

The root problem is that kernel bypass implies that no single, privileged component has global visibility into network traffic *and* source applications. Enforcing policies like the QoS example above or providing administrative tools like `tcpdump` traditionally involve OS interposition on the dataplane, but current approaches to interposition entail undesirable performance overheads.

These overheads result, broadly, from the need for data movement between the application, interposition layer, and the NIC itself. *Virtual movement* occurs when network traffic must traverse an isolation boundary on the same core, e.g., moving from userspace to the kernel in the OS stack, which introduces well-known overheads [25, 38, 46]. A few recent proposals such as IX [3] and Snap [32] replace this virtual movement with *physical movement*, placing a dataplane interposition layer on a dedicated, independent processor core and routing traffic through this secondary core. Unfortunately, physical movement also induces overheads due to coherence traffic or copies [11, 37]. The core performance benefits of kernel bypass stem from reducing data movement

HotOS '21, May 31-June 2, 2021, Ann Arbor, MI, USA

ACM ISBN 978-1-4503-8438-4/21/05.
https://doi.org/10.1145/3458336.3465281

when sending or receiving packets, from two transfers (application, to interposition layer, to NIC) to one (application to NIC).

Prior work, such as Arrakis [38] and SplitFS [21], have argued that kernel policy enforcement should be carried out entirely in the control plane, i.e., policies should merely dictate (and limit) the physical resources (NIC queues, file system blocks, etc.) that an application can access. The dataplane, on the other hand, should be unhindered by the OS. *We argue that, in the context of networking, restricting kernel policies in this manner makes it impossible to maintain and manage deployed networked applications.*

In this paper, we instead advocate a natural solution to the tension between dataplane interposition and performance: *implement the interposition layer on the NIC*. This avoids any additional physical or logical data movement (since the NIC is on the data path) and has visibility over all of the traffic sent by the host, regardless of what application produces the traffic. A key challenge with this approach, as we discuss in §3 lies in ensuring that a NIC-based implementation can support a rich set of evolving policies. Kernel developers expect an unrestricted platform for enforcing arbitrary types of policies that can also evolve quickly. In the past year alone, the Linux kernel filtering stack (`net/netfilter`) registered 377 commits, and the Linux network scheduler (`net/sched`) registered 249 commits. For this reason, 'fixed function offloads' such as TCP-offload NICs *cannot* meet the demands of developers [52]. Instead, today's technology trends provide a new opportunity: programmable SmartNICs—which include embedded CPUs, FPGAs or other programmable elements—for the first time allow an on NIC interposition layer to change and evolve as demanded by developers. We refer to this approach of implementing interposition logic on the NIC as Kernel On-Path Interposition (KOPI) and describe it in this paper.

Recent work [13] moves *hypervisor* functionality (including network isolation and traffic filtering) into the SmartNIC and is motivated by the same desire to avoid virtual and physical data movement. Given this, one might wonder whether our approach is in fact new, or even necessary? The problem with relying on the hypervisor for interposition is that policies and tools such as flow-scheduling, debugging (using tools like `iptables`, and `netstat`) require access not just to network traffic but also to other kernel datastructures including the process table. Hypervisors, which are logically isolated from the OS, cannot implement these policies and tools. As a result we focus on developing OS-integrated KOPI approaches.

We begin our exposition by looking at how the use of kernel-bypass networking impacts manageability and maintainability (§2), before laying out requirements for KOPI (§3). We then describe early work on our KOPI-based OS called Norman (§4), before concluding with a discussion (§5), and related work (§6).

2 What Kernel Bypass Has Cost Us

In this section, we consider Alice, a system administrator managing a server with users Bob and Charlie.[1] Alice is in a bare-metal deployment environment (e.g., deployments at Facebook [43, 48], containers at Google [5, 54], etc. [55]). Her server has a single high performance NIC which must be shared by Bob and Charlie's processes. We consider four common management scenarios which require dataplane interposition: debugging, partitioning ports across users and processes, process scheduling, and QoS. As we will see, all of these examples require that an interposition layer (a) have a 'global view' of traffic crossing *multiple* applications (not just one application at a time), and (b) have a 'process view' of users, processes, and on-host datastructures. Because of the need for *both* a global view and a process-level view, alternative approaches to dataplane interposition that do not directly involve the kernel are kludgy at best and impossible at worst. Dataplane interposition implemented by applications lack a global view; interposition implemented using the network (e.g., P4 [20] or a middlebox [6]) or by introducing a hypervisor switch (e.g., AccelNet [13]) lack the process-level view.

Debugging: Alice uses RSS [15] custom hashing to partition her NIC into two 'virtual interfaces' each with its own IP address—one for Bob, and one for Charlie. Alice notices a flood of ARP requests in her network with an unknown source MAC address. Without kernel bypass, Alice can inspect her server's ARP cache and `ifconfig` to determine if her server is the source of the problem. In the kernel bypass setup, however, each application is responsible for generating their own ARP traffic. Unfortunately, Alice has no *global view* of her server's network traffic and cannot trace traffic to the correct source process. Similarly. interposing at the network or hypervisor level does not help tracing traffic to a specific process. Instead, Alice must manually inspect every application installed by Bob and Charlie, one by one, to trace down the buggy ARP sender—which is tedious and scales poorly as the number of applications grows.[2]

Partitioning Ports: In another scenario, Alice assigns a single IP address to the server and wants to ensure that (a) only Postgres instances run by Bob can send or receive traffic on port 5432, and (b) only MySQL instances run by Charlie can send or receive traffic on port 3306. When using the kernel network stack, Alice can enforce this policy by adding `iptables` rules that match on `cmd-owner` (to match on process name) and `uid-owner` (to match on user ID). In a kernel bypass setup, Alice cannot enforce such a policy, and

[1] We note that the problems we present, for the most part also apply to machines used by a single user who is also the administrator. Individual administrators often run several programs on a single server and hence Alice, Bob, and Charlie may represent different levels of privilege assigned to applications owned by a single human operator.

[2] This example is in fact based on a true story from our research lab!

violations can occur through simple application misconfiguration or bugs. Interposing at the network or hypervisor level also cannot enforce this policy since neither is able to determine what process a packet originated at, or what user started the process. As a result, this policy is *unenforceable* when using kernel-bypass libraries and requires an on-host interposition layer.

Process Scheduling: Charlie and Bob run multiple applications which access the network intermittently. The applications rely on blocking I/O operations and are willing to sleep until the OS 'wakes' the application on the arrival of data. Linux supports both blocking and non-blocking operations. With kernel bypass the blocking option is not available since the kernel is not able to detect packet arrivals in the dataplane to 'wake' an application. As a consequence, Charlie and Bob are forced to use non-blocking operations and poll for packets, 'burning' CPU cores unnecessarily for applications that might be better served by blocking operations. Although interposition at the hypervisor or network level does have visibility into packet arrivals, neither can signal and unblock processes.

QoS: Alice notices that both Bob and Charlie occasionally SSH into the server to play an online-multiplayer game, and she decides to apply traffic shaping to the game's network bandwidth, so that more productive applications are unaffected. If the game uses the kernel network stack, Alice can move the game to its own control group (`cgroup`) and then use `tc` and `qdisc` to enforce a shaping policy. In a kernel bypass setup, Alice cannot enforce her shaping policy. Applications cannot individually enforce any work-conserving shaping policy (such as weighted fair queuing [10]) without viewing all rates from all competing traffic sources. Interposition at the hypervisor and network level, where one can observe all competing traffic sources, is also challenging as the game server uses different ports in each session, hence, one cannot simply set a policy to 'de-prioritize game traffic on port 1234 when it competes with traffic from applications on port 6789.' Instead, similarly to the port partition scenario, enforcing QoS requires visibility over which users and processes are generating the traffic.

3 Kernel On-Path Interposition

To *completely* and *simply* implement the kinds of features discussed in §2, we propose an intuitive design for implementing dataplane interposition: embed a *kernel-managed* dataplane in a *fully programmable* SmartNIC. We refer to this approach as Kernel On-Path Interposition and it is illustrated in Figure 1. As in systems like Arrakis [38], a KOPI operating system allows applications to open socket-like connections by requesting permission from the kernel. The kernel then enables the application to route traffic directly to/from the NIC; packets in the dataplane do not pass through the software kernel. However—unlike prior work—the kernel can install code on the NIC that also monitors, manipulates, and filters traffic in the dataplane. In what follows, we identify a few properties that are necessary for a modern interposition layer to support the kinds of functionality discussed in §2 and why KOPI uniquely meets these requirements.

The interposition layer must be isolated from the application. In the port partitioning and QoS examples from §2, we saw that policies are often designed to control/limit the activity of a particular user or process; implementing interposition to enforce policies in the applications leads to a design that is easily evaded by a malicious or compromised application. Because KOPI is implemented fully on the NIC and managed by the kernel, applications cannot evade policies enforced by the interposition layer.

The interposition layer must be able to interpose on cross-application traffic. In the debugging and QoS examples, we also saw that some administrative actions require a *global view* of traffic across multiple applications; instrumentation within a single application hence cannot implement a global traffic shaping policy and debugging using application-level interposition can be tedious because it requires inspecting every application individually. Sitting on the NIC, KOPI can view the flow of traffic over *all* processes that share the same interface—enabling effective traffic shaping/QoS and debugging.

The interposition layer must be integrated with the OS. Many interposition tasks require knowledge of processes, their ownership and privileges, and how to signal/interrupt them—as we saw in the process scheduling and partitioning ports example. Because hypervisor switches are separated from these OS-level data, they cannot implement these features (and neither can an in-network solution such as a P4 switch or a middlebox). As we will discuss in §4.3, a KOPI should be implemented with explicit signaling mechanisms between the interposition layer and the kernel.

The interposition layer must avoid unneeded data movement. We will not belabor this point, as it is already the subject of a sizable literature [11, 25, 37, 38, 56]: unnecessary transfers of data—whether virtual (involving software copies or context switches) or physical (involving moving data between cores)—lead to performance overheads that are considered unacceptable for today's network workloads. Implementing interposition on a SmartNIC avoids such data movement.

The interposition layer must be fully programmable. Many off-the-shelf NICs incorporate a variety of fixed offload engines e.g., for TCP segmentation [17, 18], TCP offload [8], or filtering [17, 18]. Is implementing a useful interposition layer simply a matter of developing the right collection of hardware accelerators? Unfortunately, implementing complex logic in fixed function hardware necessarily limits the evolution of new protocols and policies (as new hardware

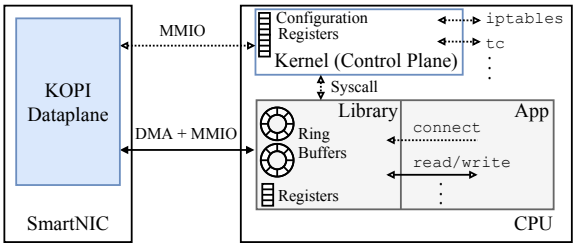

Figure 1. Norman Architecture.

comes out on timescales measured in years), where software developers working on the kernel network stack are accustomed to issuing hundreds of updates per year—as previously mentioned, in 2020 alone `net/netfilter` registered 377 commits and `net/sched` had 249 commits in the Linux kernel. This lack of ability to upgrade is part of the reason that the OS community has resoundingly rejected complex offloads such as TCP Offload Engines [2, 52]. Programmability is hence necessary to ensure that new networking policies and features continue to be developed and adopted at the same rate as they are today; a *Smart*NIC is capable of this where a fixed-function NIC is not.

Until recently, most servers had no on-path location that met these requirements: On most servers, data was generated in the application, and then transferred to NICs which were not programmable, before leaving the host. Today, we finally have a new opportunity which meets *all* of the above requirements: modern SmartNICs.

4 Norman Design Sketch

We are developing Norman, an operating system which implements KOPI. We first describe the hardware targeted by Norman, and then give an overview of Norman's architecture and some of the mechanisms required to realize it.

4.1 Hardware Considerations

Norman targets servers that include FPGA-based SmartNICs, where the FPGA is 'on-path' [19], i.e., SmartNICs where all packets traverse the FPGA by default. Our use of an FPGA in the dataplane means that Norman includes components written in C (running on the host CPU) and in RTL (running on the FPGA NIC). We chose FPGAs since (a) they are fully/generically programmable; (b) there exist SmartNICs which include FPGAs on-path;[3] and (c) recent work [1, 4, 13, 29, 39, 57] has demonstrated exciting performance results for network processing on FPGA-based SmartNICs, and we hope to achieve similarly high performance.

4.2 Norman Architecture

In Figure 1 we show an overview of Norman's architecture. Our goal in designing Norman was to provide the same separation as the kernel: the kernel should have complete control over the NIC, and expose a narrow interface (similar to the socket interface) for applications. Norman is comprised of the following three components:

First, the **on-SmartNIC dataplane** implements all of the interposition logic including packet filters, queueing disciplines, congestion control, and packet sniffing. The dataplane sends and receives packets directly to/from userspace (§4.3), but receives control plane signaling through a separate interface with the kernel (§4.4).

The **Norman library** provides abstractions that allow applications to interface with the network. It provides both POSIX APIs—so that applications can be easily portable between Norman and other systems—as well as more efficient abstractions that prevent unnecessary copies [38, 56]. As we describe below, the Norman library interfaces with both the in-kernel control plane and the on-SmartNIC dataplane to implement its functionality. The library also implements dataplane functionality that does not require privileged interposition.

Finally, the **in-kernel control plane** is responsible for allocating network resources to applications and for configuring the on-SmartNIC dataplane. Tools such as `tc`, `iptables` and `tcpdump` also call into the in-kernel control plane, which updates the SmartNIC dataplane.

4.3 Applications' Dataplane Interface

The Norman library allows applications to use the familiar sockets interface to communicate over the network, while ensuring that operations which send or receive data do not go through any off-path interposition logic. In order to do so, the library calls into the in-kernel control plane when handling calls that establish a new connection (e.g., `connect(2)` and `accept(2)`). In response to these calls, the in-kernel control plane allocates (and pins) memory for a pair of per-connection ring-buffers that the application uses to send and receive data from the connection, and configures the NIC so that packets belonging to the connection are written to the appropriate ring buffer. The kernel also grants the application access to SmartNIC MMIO registers that store the head and tail pointer for each ring buffer. Using these MMIO registers and ring buffers, the application can directly send and receive data by merely accessing memory. Our approach to implementing connection setup in the kernel and allowing applications direct access to NIC resources for sending and receiving data is similar to the approach adopted by Arrakis [38] and TAS [25], however we can support richer dataplane policies.

A second concern that Norman's architecture needs to address is how to provide blocking system calls. Similar to kernel-bypass approaches, our use of memory reads and writes is at odds with blocking for I/O. The Norman dataplane therefore allows connections to be configured so that the NIC adds notification to a shared notification queue when

[3] Some SmartNICs include FPGAs as a 'lookaside' [35] but we do not use these platforms to avoid unnecessary physical data movement.

packets are added to a queue (allowing blocking receive calls) or when a queue is drained (allowing blocking for sends). A process's notification queue is accessible to both the process and the kernel, and the Norman kernel control plane is responsible for monitoring notifications sent to blocked threads, and unblocking the thread when necessary. The control plane on the kernel can also choose to enable interrupts for notification queues with low activity. This allows Norman to support both blocking and non-blocking I/O while making efficient use of CPU cycles.

4.4 Configuring the On-SmartNIC Dataplane

Only the kernel has privilege to configure the SmartNIC, so any runtime configuration made using utilities like `iptables` or `tc` continue to be routed through the kernel. Under the hood, the kernel applies the new configurations to the on-SmartNIC dataplane in two main ways. Some changes, like inserting a new firewall rule, simply require injecting new data into memory on the SmartNIC and are made using commands passed through MMIO registers. However, some changes require changing functionality *on the fly*, such as applying a new queueing policy. For these changes we adopt a new approach to designing FPGA programs called an *overlay* [4, 28]. An overlay can be thought of as a custom, potentially non-Turing complete processor with a domain-specific instruction set (e.g., an instruction set for defining traffic shaping policies). To load a new policy, one do not need to change the underlying hardware, but load a new 'program' into the overlay.

In addition to configuration updates, one may wish to install an entirely new bitstream to the FPGA—that is, to rewrite the hardware. These operations take seconds or longer, and can be thought of as the equivalent to upgrading the kernel itself; Norman might require such an update to, e.g., add support for eBPF to a dataplane which previously did not offer eBPF, but should not require such an update for most configuration changes including ones required by tools such as `tc` or `iptables`.

5 Open Challenges

We are currently implementing Norman using a fork of the Linux kernel and a Stratix 10 MX FPGA. Realizing the design approach we have set out in the previous section is continually revealing to us new systems challenges and we outline several of our current open questions as follows.

How high can a per-connection application interface scale? In §4.3 we discussed how Norman allocates a pair of ring buffers for each connection, and configures the NIC to direct packets from a connection to the appropriate ring buffer. Our current implementation fails to sustain full (100Gbps) throughput when there are more than 1024 concurrent connections, although a single IP address should be able to support millions of connections! One possible reason for this is that DDIO, which Intel uses to improve I/O performance, can only use a fixed fraction of LLC cache space [31, 53], and can slow down I/O if more cache space is necessary. We suspect that the number of active ring buffers is outstripping the DDIO cache thus impacting our performance. Beyond performance concerns, resource limits on the NIC might hold us back from scaling to more connections: NICs have relatively little on-board memory [23, 45, 57], and prior work has shown that the need for per-connection state at the NIC can be a scalability bottleneck [23, 45]. One can reduce state requirements by sharing buffers across connections, but this brings its own challenges and might require changing application abstractions. At present, it is simply not clear whether per-connection semantics are feasible, or if sustaining high throughput will necessitate sharing ring buffers between connections from the same application.

Is an FPGA reconfigurable enough to support online configuration updates? We chose to use an FPGA based SmartNIC, in part because of FPGAs' success in achieving high performance for a range of networking applications [1, 4, 13, 29, 39, 57]. However, they have one distinct disadvantage relative to CPU and NPU based NICs: changing the installed functionality on the dataplane takes longer for an FPGA than it does to change the instructions running on the CPU/NPU. In section 4, we discussed our proposal to use 'overlays' to swap out different custom behaviors, e.g., for queueing disciplines, online. However, designing overlays that are resource efficient (using limited memory and logic gates), achieve high performance, *and* are flexible is an active area of research on its own [28, 51]. It is yet unclear that our FPGA overlay approach will prove superior to a similar CPU/NPU based design to implement KOPI.

Can we prevent a KOPI from being vulnerable to resource exhaustion? SmartNICs inherently have limited memory relative to the amount of available on-host memory. This makes them vulnerable to resource-exhaustion attacks (as has been noted in attempts to deploy TCP offloads [52]). Given the complexity of functionality we aim to offload to the NIC—filtering, queueing, per-connection state, NAT, and everything else the kernel does today—the potential to exhaust NIC resources is all the more dramatic than prior, single-task offloads to SmartNICs. Our hope is that a combination of careful data structure design, as well as the option to route 'low priority' or 'performance non-critical' traffic through a software datapath, will mitigate these challenges. But we have yet to explore the limits of NIC memory on KOPI in detail.

6 Related Work and Conclusion

KOPI can be seen as a natural evolution at the convergence of several lines of research. First, the case for new operating systems which bypass the kernel networking stack has been made elegantly by Peter *et al.* with Arrakis [38], and by Belay

et al. with IX [3]; several other systems follow suit [32, 56]. At the same time, AccelNet [13] makes the case for *hypervisor* dataplane offloads to SmartNICs. Finally, several works have also explored offloading *individual* kernel functionality, e.g., parts of the TCP stack [1, 33, 34, 44, 49], packet steering[24, 42, 50], QoS [13, 30, 50], filtering [13, 24, 30, 49], rate limiting [26, 30, 41, 50], and process scheduling [16].

KOPI is inspired by all of the above. Remove dataplane operation from the software kernel (like Arrakis and IX), offload network processing to a SmartNIC (like AccelNet), and implement rich functionality—beyond the simple switching of AccelNet—in the hardware dataplane (like the many offloads listed above).

Indeed, from the perspective of these authors, the question now is not *whether* KOPI is the right next step for high-performance networking, but *what we need to do to achieve it.* Can current SmartNICs support all of the functionality we require? Do we need to extend SmartNIC hardware somehow, e.g., to better support thousands of concurrent sockets? How do we prevent resource exhaustion on the limited resources of the SmartNIC? Is it really feasible to implement filtering and queueing disciplines in a way that is updatable on demand, resource-efficient, *and* achieves high performance? Hence, our next steps are to continue developing Norman and tackle these challenges head-on.

Acknowledgments

We thank the anonymous reviewers for their great comments and feedback. This work was supported in part by funding from a VMware Systems Research Award, NSF grant 2028832, and by Intel and VMware through the Intel/VMware Crossroads 3D-FPGA Academic Research Center.

References

[1] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *NSDI '20*. 93–109.

[2] Michael C. Bazarewsky. 2017. Why Are We Deprecating Network Performance Features (KB4014193)? https://techcommunity.microsoft.com/t5/core-infrastructure-and-security/why-are-we-deprecating-network-performance-features-kb4014193/ba-p/259053.

[3] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *OSDI '14*. 49–65.

[4] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2020. hXDP: Efficient Software Packet Processing on FPGA NICs. In *OSDI '20*. 973–990.

[5] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Queue* 14, 1 (2016), 70–93.

[6] B. Carpenter and S. Brim. 2002. Middleboxes: Taxonomy and Issues. RFC 3234 (Informational). http://www.ietf.org/rfc/rfc3234.txt

[7] David D. Clark. 1985. The Structuring of Systems Using Upcalls. In *SOSP '85*. 171–180.

[8] Chelsio Communications. 2021. Terminator 5 ASIC. https://www.chelsio.com/terminator-5-asic/.

[9] Jonathan Corbet. 2017. The future of DAX. https://lwn.net/Articles/717953/.

[10] A. Demers, S. Keshav, and S. Shenker. 1989. Analysis and Simulation of a Fair Queueing Algorithm. In *SIGCOMM '89*. 1–12.

[11] Mihai Dobrescu, Katerina Argyraki, Gianluca Iannaccone, Maziar Manesh, and Sylvia Ratnasamy. 2010. Controlling Parallelism in a Multicore Software Router. In *PRESTO '10*. Article 2.

[12] DPDK. 2021. Data Plane Development Kit. https://dpdk.org.

[13] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI '18*. 51–66.

[14] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. *SoftNIC: A Software NIC to Augment Hardware.* Technical Report UCB/EECS-2015-155. EECS Department, University of California, Berkeley.

[15] Ted Hudek. 2017. Introduction to Receive Side Scaling. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling.

[16] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. 2019. Mind the Gap: A Case for Informed Request Scheduling at the NIC. In *HotNets '19*. 60–68.

[17] Intel. 2019. Intel 82599 10 GbE Controller Datasheet.

[18] Intel. 2021. Intel Ethernet Controller X710/XXV710/XL710 Datasheet.

[19] Intel. 2021. Intel Stratix 10 MX FPGA Development Kit. https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/kit-s10-mx.html.

[20] Intel. 2021. Intel Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-booktitle/tofino.html.

[21] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *SOSP '19*. 494–508.

[22] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-Scale Tail Latency. In *NSDI '19*. 345–360.

[23] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs Can Be General and Fast. In *NSDI '19*. 1–16.

[24] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *ASPLOS '16*. 67–81.

[25] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP Acceleration as an OS Service. In *EuroSys '19*. Article 24.

[26] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Jake Adriaens, Steve Gribble, Nate Foster, and Amin Vahdat. 2019. PicNIC: Predictable Virtualized NIC. In *SIGCOMM '19*. 351–366.

[27] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *SOSP '17*. 460–477.

[28] Maysam Lavasani. 2015. *Generating Irregular Data-stream Accelerators: Methodology and Applications.* Ph.D. Dissertation. The University of Texas at Austin.

[29] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *SIGCOMM '16*. 1–14.

[30] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. 2020. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *OSDI '20*. 243–259.

[31] Antonis Manousis, Rahul Anand Sharma, Vyas Sekar, and Justine Sherry. 2020. Contention-Aware Performance Prediction For Virtualized Network Functions. In *SIGCOMM '20*. 270–282.

[32] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *SOSP '19*. 399–413.

[33] Jeffrey C. Mogul. 2003. TCP Offload Is a Dumb Idea Whose Time Has Come. In *HotOS '03*. 5.

[34] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *NSDI '20*. 77–92.

[35] Nvdia. [n.d.]. Nvidia Mellanox Innova-2 Flex Open Adapter Card.

[36] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *NSDI '19*. 361–378.

[37] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *OSDI '16*. 203–216.

[38] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System Is the Control Plane. In *OSDI '14*. 1–16.

[39] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. 2019. FlowBlaze: Stateful Packet Processing in Hardware. In *NSDI '19*. 531–548.

[40] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *SOSP '17*. 325–341.

[41] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. 2014. SENIC: Scalable NIC for End-Host Rate Limiting. In *NSDI '14*. 475–488.

[42] Kaushik Kumar Ram, Jayaram Mudigonda, Alan L. Cox, Scott Rixner, Parthasarathy Ranganathan, and Jose Renato Santos. 2010. sNICh: Efficient Last Hop Networking in the Data Center. In *ANCS '10*. Article 26, 12 pages.

[43] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *SIGCOMM '15*. 123–137.

[44] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. 2015. Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware. In *FCCM '15*. 36–43.

[45] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 2020. 1RMA: Re-Envisioning Remote Memory Access for Multi-Tenant Datacenters. In *SIGCOMM '20*. 708–721.

[46] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *OSDI '10*. 33–46.

[47] SPDK. 2021. Storage Performance Development Kit. https://spdk.io.

[48] Akshitha Sriraman and Abhishek Dhanotia. 2020. Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale. In *ASPLOS '20*. 733–750.

[49] Brent Stephens, Aditya Akella, and Michael M. Swift. 2018. Your Programmable NIC Should Be a Programmable Switch. In *HotNets '18*. 36–42.

[50] Brent Stephens, Aditya Akella, and Michael M. Swift. 2019. Loom: Flexible and Efficient NIC Packet Scheduling. In *NSDI '19*. 33–46.

[51] Ian Taras and Jason H. Anderson. 2019. Impact of FPGA Architecture on Area and Performance of CGRA Overlays. In *FCCM '19*. 87–95.

[52] The Linux Foundation. 2016. TOE. https://wiki.linuxfoundation.org/networking/toe.

[53] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. 2018. ResQ: Enabling SLOs in Network Function Virtualization. In *NSDI '18*. 283–297.

[54] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In *EuroSys '15*. Article 18, 17 pages.

[55] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the Curtains of Serverless Platforms. In *ATC '18*. 133–145.

[56] Irene Zhang, Jing Liu, Amanda Austin, Michael Lowell Roberts, and Anirudh Badam. 2019. I'm Not Dead yet! The Role of the Operating System in a Kernel-Bypass Era. In *HotOS '19*. 73–80.

[57] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. 2020. Achieving 100Gbps Intrusion Prevention on a Single Server. In *OSDI '20*. 1083–1100.