

Snap: a Microkernel Approach to Host Networking

Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli*, Michael Dalton*, Nandita Dukkupati, 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
Google, Inc.
sosp2019-snap@google.com

Abstract

This paper presents our design and experience with a microkernel-inspired approach to host networking called Snap. Snap is a userspace networking system that supports Google's rapidly evolving needs with flexible modules that implement a range of network functions, including edge packet switching, virtualization for our cloud platform, traffic shaping policy enforcement, and a high-performance reliable messaging and RDMA-like service. Snap has been running in production for over three years, supporting the extensible communication needs of several large and critical systems.

Snap enables fast development and deployment of new networking features, leveraging the benefits of address space isolation and the productivity of userspace software development together with support for transparently upgrading networking services without migrating applications off of a machine. At the same time, Snap achieves compelling performance through a modular architecture that promotes principled synchronization with minimal state sharing, and supports real-time scheduling with dynamic scaling of CPU resources through a novel kernel/userspace CPU scheduler co-design. Our evaluation demonstrates over 3x Gbps/core improvement compared to a kernel networking stack for RPC workloads, software-based RDMA-like performance of up to 5M IOPS/core, and transparent upgrades that are largely imperceptible to user applications. Snap is deployed to over half of our fleet of machines and supports the needs of numerous teams.

CCS Concepts • **Networks** → **Network design principles**; Data center networks; • **Software and its engineering**; • **Computer systems organization** → *Maintainability and maintenance*;

*Work performed while at Google.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-6873-5/19/10.

<https://doi.org/10.1145/3341301.3359657>

Keywords network stack, datacenter, microkernel, RDMA

ACM Reference Format:

Marty and De Kruijf, et al. 2019. Snap: a Microkernel Approach to Host Networking. In *ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19)*, October 27–30, 2019, Huntsville, ON, Canada. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3341301.3359657>

1 Introduction

The host networking needs of large-scale Internet services providers are vast and rapidly evolving. Continuous capacity growth demands novel approaches to edge switching and bandwidth management, the rise of cloud computing necessitates rich virtualization features, and high-performance distributed systems continually seek more efficient and lower latency communication.

Our experiences realizing these needs with conventional kernel-based networking were hampered by lengthy development and release cycles. Hence, several years ago we started an effort to move networking functionality out of the kernel and into userspace modules through a common framework. This framework, Snap, has evolved into a rich architecture supporting a diverse set of host networking needs with high performance, and critically, high developer productivity and release velocity.

Before Snap, we were limited in our ability to develop and deploy new network functionality and performance optimizations in several ways. First, developing kernel code was slow and drew on a smaller pool of software engineers. Second, feature release through the kernel module reloads covered only a subset of functionality and often required disconnecting applications, while the more common case of requiring a machine reboot necessitated draining the machine of running applications. Mitigating this disruption required us to severely pace our rate of kernel upgrades. In practice, a change to the kernel-based stack takes 1-2 months to deploy whereas a new Snap release gets deployed to our fleet on a weekly basis. Finally, the broad generality of Linux made optimization difficult and defied vertical integration efforts, which were easily broken by upstream changes.

The Snap architecture has similarities with a *microkernel* approach where traditional operating system functionality is hoisted into ordinary userspace processes. However, unlike prior microkernel systems, Snap benefits from multicore

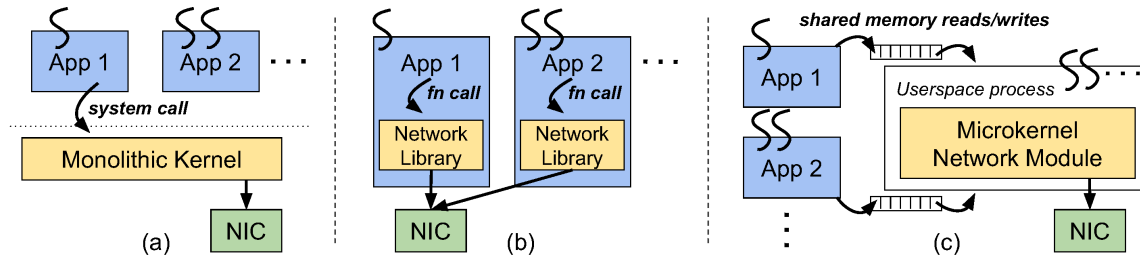


Figure 1. Three different approaches to organizing networking functionality: (a) shows a traditional monolithic kernel where applications make system calls, (b) shows a library OS approach without centralization and with application-level thread scheduling of processing, and (c) shows the Snap microkernel-like approach leveraging multicore for fast IPC.

hardware for fast IPC and does not require the entire system to adopt the approach wholesale, as it runs as a userspace process alongside our standard Linux distribution and kernel. Yet Snap retains the advantages of a centralized OS for managing and scheduling resources, which makes our work distinct from other recent work in userspace networking that put functionality into uncoordinated application libraries [1, 14, 30, 51]. Our approach also allows Snap to eschew the complexity of full compatibility with existing interfaces like POSIX and socket system calls, and to instead favor opportunities for vertical integration with our applications. Through Snap, dataplane interaction occurs over custom interfaces that communicate via lock-free shared memory queues. Additionally, for use cases that integrate with existing kernel functionality, Snap supports an internally-developed driver for efficiently moving packets between Snap and the kernel.

Unlocking higher levels of performance in networking is critical as Moore’s Law slows and as faster storage devices continue to emerge. Through Snap, we created a new communication stack called *Pony Express* that implements a custom reliable transport and communications API. Pony Express provides significant communication efficiency and latency advantages to our applications, supporting use cases ranging from web search to storage.

Snap’s architecture is a composition of recent ideas in userspace networking, in-service upgrades, centralized resource accounting, programmable packet processing, kernel-bypass RDMA functionality, and optimized co-design of transport, congestion control and routing. We present our experience building and deploying Snap across Google, and as a unified and production-hardened realization of several modern systems design principles:

- Snap enables a high rate of feature development with a microkernel-inspired approach of developing in userspace with transparent software upgrades. It retains the benefits of centralized resource allocation and management capabilities of monolithic kernels while also improving upon accounting gaps with existing Linux-based systems.
- Any practical evolution of operating system architecture requires interoperability with existing kernel network functions and application thread schedulers. Snap implements

a custom kernel packet injection driver and a custom CPU scheduler that enables interoperability without requiring adoption of new application runtimes and while maintaining high performance across use cases that simultaneously require packet processing through both Snap and the Linux kernel networking stack.

- Snap encapsulates packet processing functions into composable units called “engines”, which enables both modular CPU scheduling as well as incremental and minimally-disruptive state transfer during upgrades.
- Through Pony Express, Snap provides support for OSI layer 4 and 5 functionality through an interface similar to an RDMA-capable “smart” NIC. In concert with transparent upgrade support, this enables transparently leveraging offload capabilities in emerging hardware NICs as a means to further improve server efficiency and throughput.
- Minimizing I/O overhead is critical to scaling modern distributed services. We carefully tuned Snap and the Pony Express transport for performance, supporting 3x better transport processing efficiency than the baseline Linux kernel and supporting RDMA-like functionality at speeds of 5M ops/sec/core.

2 Snap as a Microkernel Service

Unlike monolithic operating systems, Snap implements host networking functionality as an ordinary Linux userspace process. However, compared to other userspace networking approaches that assume a library OS model [14, 30, 31, 51, 59], Snap maintains the centralized coordination and manageability benefits of a traditional OS. The host grants Snap exclusive access to device resources by enabling specific Linux capabilities for Snap (Snap does not run with root privileges) and applications communicate with Snap through library calls that transfer data either asynchronously over shared memory queues (fast path) or synchronously over a Unix domain sockets interface (slow path).

Figure 1 illustrates the differences among (a) the traditional approach of placing all networking functionality in kernel-space, (b) the library OS approach of accessing hardware directly from application threads, and (c) the Snap approach

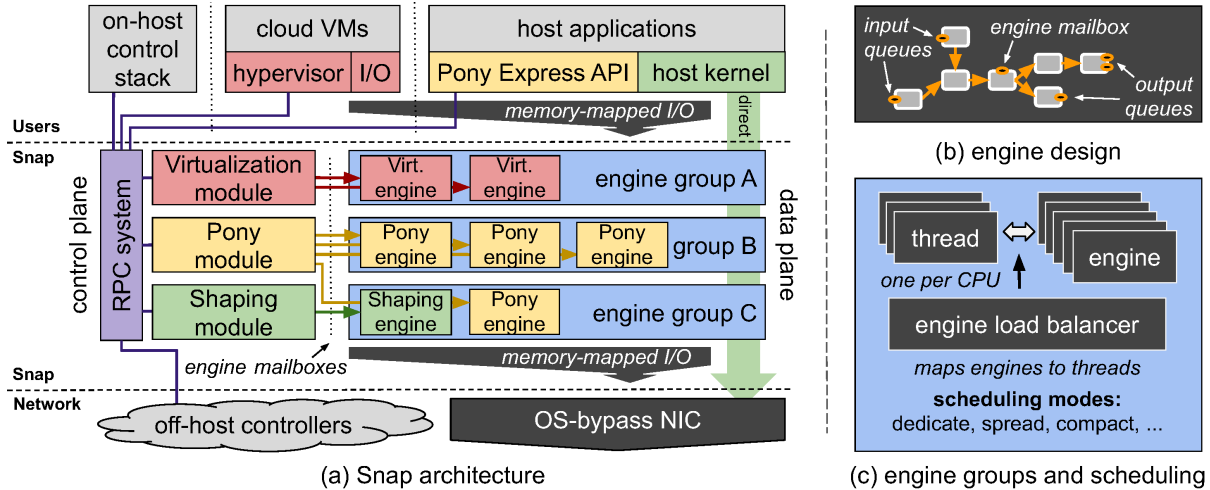


Figure 2. Snap is divided into a control plane, centered around RPC serving, and a data plane, centered around engines. Engines encapsulate packet processing pipelines that communicate through memory-mapped I/O. Engine groups dictate the method of scheduling engines to cores. Except for host kernel interactions, all code executes in userspace.

of leveraging fast multicore IPC to implement network functionality as a user-level service.

Approach (c) provides a number of benefits. First, it combines the centralization benefits of (a) with the userspace development benefits of (b). Second, it uniquely decouples the release of new networking functionality from *both* kernel and application binary release cycles. Third, unlike the library OS implementations of approach (b) that achieve low latency but typically rely on spin-polling application threads to do so, approach (c) decouples application threading and CPU provisioning from network services. This enables managing networking latency and spin-polling capability as a system-level resource, which is essential for production machines that typically run dozens of independent applications.

Over the years, Snap has incrementally absorbed networking functionality from the host kernel while other OS functionality like memory management, thread scheduling, existing kernel networking, and non-networking I/O continues to leverage the rich Linux ecosystem. This aspect is not like a traditional microkernel. However, Snap still maintains the microkernel benefits of address space isolation between the networking component and other kernel components, developer and release velocity as bugs found during development and testing do not bring down the machine, and centralization, which enables rich scheduling and management policies that are lost with traditional OS-bypass networking systems.

While early microkernel work saw significant performance overheads attributed to inter-process communication (IPC) and address space changes [15, 16, 20], such overheads are less significant today. Compared to the uniprocessor systems of the 80s and 90s, today’s servers contain dozens of cores, which allows microkernel invocation to leverage inter-core IPC while maintaining application cache locality. This approach can even *improve* overall performance when there is

little state to communicate across the IPC (common in zero-copy networking) and in avoiding ring switch costs of system calls. Moreover, recent security vulnerabilities such as Melt-down [43] force kernel/user address space isolation, even in monolithic kernels [25]. Techniques like tagged-TLB support in modern processors, streamlined ring switching hardware made necessary with the resurgence of virtualization, and IPC optimization techniques such as those explored in the L4 microkernel [29], in FlexSC [58], and in SkyBridge [44], further allow a modern microkernel to essentially close the performance gap between direct system calls and indirect system calls through IPC.

2.1 Snap Architecture Overview

Figure 2(a) shows the architecture of Snap and its interactions with external systems. It shows Snap separated into “control plane” (left) and “data plane” (right) components, with *engines* as the unit of encapsulation for data plane operations. Examples of engines include packet processing for network virtualization [19], pacing and rate limiting (“shaping”) for bandwidth enforcement [39], and a stateful network transport like Pony Express (presented in Section 3).

Figure 2(a) also illustrates the separate communication paradigms employed across distinct component types: on the left, communication between control plane components and external systems is orchestrated through RPC; on the right, communication to and from data plane components occurs through memory-mapped I/O; and in the middle, control plane and data plane components interact through a specialized unidirectional RPC mechanism called an *engine mailbox*. Finally, the figure shows engines organized into groups that share a common scheduling discipline. Engines are shown handling all guest VM I/O traffic, all Pony Express traffic, and a subset of host kernel traffic that needs Snap-implemented traffic

shaping policies applied. The following sections elaborate on each of these aspects in turn.

2.2 Principles of Engine Design

Engines are stateful, single-threaded tasks that are scheduled and run by a Snap engine scheduling runtime. Snap exposes to engine developers a bare-metal programming environment with libraries for OS-bypass networking, rate limiting, ACL enforcement, protocol processing, tuned data structures, and more, as well as a library of Click-style [47] pluggable “elements” to construct packet processing pipelines.

Figure 2(b) shows the structure of a Snap engine and its interaction with external systems through queues and mailboxes. An engine’s inputs and outputs can include userspace and guest applications, kernel packet rings, NIC receive/transmit queues, support threads, and other engines. In each of these cases, lock-free communication occurs over memory-mapped regions shared with the input or output. Occasionally, an engine may also communicate with outputs via interrupt delivery (*e.g.* by writing to an `eventfd`-like construct) to notify of inbound work, and interrupts may also be received on inputs under some engine scheduling policies (see Section 2.4). However, these are the only forms of communication generally performed by engines. In particular, to maintain real-time properties, Snap prohibits forms of communication that rely on blocking synchronization.

From the perspective of the Snap framework, it is the role of engine developers to determine how many engines are needed, how they are instantiated—either statically or dynamically—and how work is distributed among engines (utilizing NIC steering functionality as needed). Sections 3.1 and 6.4 provide some discussion of this topic, but it is otherwise not a focus of this paper.

2.3 Modules and Control-to-Engine Communication

Snap modules are responsible for setting up control plane RPC services, instantiating engines, loading them into engine groups, and proxying all user setup interactions for those engines. For example, in the case of Pony Express, the “Pony module” shown in Figure 2(a) authenticates users and sets up memory regions shared with user applications by exchanging file descriptors over a local RPC system. It also services other performance-insensitive functions such as engine creation/destruction, compatibility checks, and policy updates.

Some control actions in the service of the above, such as setting up user command/completion queues, shared memory registration, rotating encryption keys, etc. require control components to synchronize with engines. To support Snap’s real-time, high performance requirements, control components synchronize with engines lock-free through an *engine mailbox*. This mailbox is a queue of depth 1 on which control components post short sections of work for synchronous execution by an engine, on the thread of the engine, and in a manner that is non-blocking with respect to the engine.

2.4 Engine Groups and CPU Scheduling

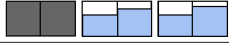


Snap seeks to balance the scheduling latency, performance isolation, and CPU efficiency of engines. However, this balance is different for different types of engines: some engines care most about low tail latency, while others seek maximum fairness given a fixed CPU budget, while still others aim to prioritize efficiency above all else.

Snap accommodates each of these cases with support for bundling engines into groups with a specific scheduling *mode*, which dictates a scheduling algorithm and CPU resource constraints. Figure 2(c) shows the composition of an engine group and Figure 3 classifies three broad categories of scheduling modes supported by Snap.

Dedicating cores: In this mode, engines are pinned to dedicated hyperthreads on which no other work can run. Although this mode does not allow CPU utilization to scale in proportion to load, it can be suitable when the CPU allowance is a fixed budget of the total machine, where it can then also minimize latency via spin polling, optionally invoking a userspace `mwait` [9] to conserve power. Due to static provisioning, however, this system can strain under load or face high cost from overprovisioning. When CPU constrained, the scheduler attempts to fair-share CPU time between engines in order to provide reasonable performance isolation and robustness under high load conditions.

Spreading engines: This mode scales CPU consumption in proportion to load, with a focus on minimizing scheduling tail latency. The Snap implementation binds each engine to a unique thread that schedules only when active and blocks on interrupt notification when idle. Interrupts are then triggered either from the NIC or from an application via a system call to schedule an engine. Given a sufficient number of schedulable cores, this mode can provide the best tail latency properties for two reasons. First, it is not subject to scheduling delays caused by multiplexing the work of potentially many engines onto a small number of threads or cores. Second, by leveraging our internally-developed real-time kernel scheduling class (more detail in Section 2.4.1), engines bypass the default Linux CFS kernel scheduler [5] and quickly schedule, with priority, to an available core upon interrupt delivery. With interrupts, however, there are system-level interference effects that must be carefully managed; in Section 5.3 we elaborate on some of the schedulability challenges that may arise when a thread is scheduled to a core that is either in a low-power sleep state or is in the midst of running non-preemptible kernel code.

Compacting engines: This mode collapses work onto as few cores as possible, combining the scaling advantages of interrupt-driven execution with the cache-efficiency advantages of dedicating cores. However, it relies on periodic polling of engine queueing delays to detect load imbalance instead of relying on instantaneous interrupt signaling as with the “spreading engines” scheduling mode above. The speed of rebalancing is thus constrained by the latency in polling

scheduling mode	CPU resources	scheduling latency		CPU efficiency	visualization
		median	tail		
dedicating cores	static	0-1 μ s	0*-100+ μ s	poor	
spreading engines	dynamic	3-10 μ s	10-30** μ s	good	
compacting engines	dynamic	0-5 μ s	50-100** μ s	excellent	

* 0 μ s tail scheduling latency under “dedicating cores” possible only when running a single engine per core

** assumes minimal tail latency impact due to low-power sleep states and/or possible preemption failure

Figure 3. Snap supports three different engine scheduling modes with different resource, latency, and efficiency properties. In the visualization column, a rectangle is a physical core that contains two logical cores, with portions consumed by Snap shown in dark gray, non-Snap shown in light blue, and idle time shown in white.

for these queueing delays, which, in our current design, is non-preemptive and requires engine tasks to return control to the scheduler within a fixed latency budget. The latency delay from polling engines, in addition to both the delay for statistical confidence in queueing estimation and the delay in actually handing off an engine to another core, can be higher than the latency of interrupt signaling.

In terms of implementation, this scheduling mode executes engines on a single thread while the CPU-bottlenecked queueing delay of all engines—measured using an algorithm similar to Shenango [48]—stays below some configured latency SLO. Then, interleaved with engine execution, a rebalancing function periodically performs one of several actions. One action is to respond to queue build-up by scaling out an engine to another thread (awoken if necessary). Another action is to detect excess capacity and migrate back an engine that has sufficiently low load. Other actions include engine compaction and engine swaps to effectively bin-pack work and maximize efficiency within the constraints of the SLO. The algorithm estimates the queueing load of engines by directly accessing concurrently-updated shared variables in memory, with any subsequent load balancing decision synchronizing directly with affected threads through a message passing mechanism similar to the engine mailbox, but non-blocking on both sides. Although fundamentally driven by polling, this mode also supports blocking on interrupt notification after some period of idleness in order to scale down to less than a full core.

2.4.1 MicroQuanta Kernel Scheduling Class

To dynamically scale CPU resources, Snap works in conjunction with a new lightweight kernel scheduling class called *MicroQuanta* that provides a flexible way to share cores between latency-sensitive Snap engine tasks and other tasks, limiting the CPU share of latency-sensitive tasks and maintaining low scheduling latency at the same time. A *MicroQuanta* thread runs for a configurable *runtime* out of every *period* time units, with the remaining CPU time available to other CFS-scheduled tasks using a variation of a fair queueing

algorithm for high and low priority tasks (rather than more traditional fixed time slots).

MicroQuanta is a robust way for Snap to get priority on cores runnable by CFS tasks that avoids starvation of critical per-core kernel threads [7]. The sharing pattern is similar to *SCHED_DEADLINE* or *SCHED_FIFO* with real-time group scheduling bandwidth control. However, *MicroQuanta* schedules with much lower latency and scales to many more cores. While other Linux real-time scheduling classes use both per-CPU tick-based and global high-resolution timers for bandwidth control, *MicroQuanta* uses only per-CPU high-resolution timers. This allows scalable time slicing at microsecond granularity.

2.5 CPU and Memory Accounting

Strong CPU and memory accountability is important in datacenter and cloud computing environments because of the heavy and dynamic multiplexing of VMs and jobs onto machines. For example, prior work [12, 36] demonstrates the shortcoming of soft-interrupt accounting with the kernel networking stack, as *softirqs* steal CPU time from whatever application happens to be running on the core irrespective of whether *softirq* processing is for data destined to that application. Snap maintains strong accounting and isolation by accurately attributing both CPU and memory consumed on behalf of applications to those applications using internally-developed Linux kernel interfaces to charge CPU and memory to application containers. This allows Snap to scale Snap CPU processing and per-user memory consumption (for per-user data structures) without oversubscribing the system.

2.6 Security

Snap engines may handle sensitive application data, doing work on behalf of potentially multiple applications with differing levels of trust simultaneously. As such, security and isolation are critically important. Unlike a monolithic kernel, Snap runs as a special non-root user with reduced privilege, although care must still be taken to ensure that packets and

payloads are not mis-delivered. Applications establishing interactions with Snap authenticate its identity using standard Linux mechanisms.

One benefit of the Snap userspace approach is that software development can leverage a broad range of internal tools, such as memory access sanitizers and fuzz testers, developed to improve the security and robustness of general user-level software. The CPU scheduling modes of Section 2.4 also provide options to mitigate Spectre-class vulnerabilities [37] by cleanly separating cores running engines for certain applications from those running engines for different applications.

3 Pony Express: A Snap Transport

Over the course of several years, the architecture underpinning Snap has been used in production for multiple networking applications, including network virtualization for cloud VMs [19], packet-processing for Internet peering [62], scalable load balancing [22], and *Pony Express*, a reliable transport and communications stack that is our focus for the remainder of this paper. Our datacenter applications seek ever more CPU-efficient and lower-latency communication, which Pony Express delivers. It implements reliability, congestion control, optional ordering, flow control, and execution of remote data access operations.

Rather than reimplement TCP/IP or refactor an existing transport, we started Pony Express from scratch to innovate on more efficient interfaces, architecture, and protocol. The application interface to Pony Express is based on asynchronous operation-level commands and completions, as opposed to a packet-level or byte-streaming sockets interface. Pony Express implements both (two-sided) messaging operations as well as *one-sided* operations of which RDMA is one example. A one-sided operation does not involve any remote application thread interaction and thereby avoids invoking the application thread scheduler for remote data access.

As a contrast to the Linux TCP stack, which attempts to keep transport processing affinity local to application CPUs [8, 21], Pony Express runs transport processing in a thread separate from the application CPU through Snap, which affinityizes its engine threads to the NUMA node of the PCI-attached NIC. Through Snap, Pony Express predominantly shares CPU with other engines and other transport processing work rather than application work. This enables better batching, code locality, reduced context switching, and opportunities for latency reduction through spin polling. The overarching design principle is that, given zero-copy capability, NIC NUMA node locality and locality within the transport layer are together more important than locality with the application thread.

3.1 Architecture and Implementation

The architecture of Pony Express is illustrated in Figure 4. Client applications contact Pony Express over a Unix domain socket at a well-known address through the Pony Express client library API. As previously described in Section 2.3,

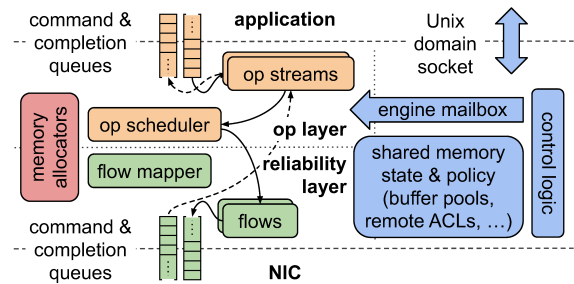


Figure 4. The architecture of a Pony Express engine on the left and associated control logic on the right.

this socket bootstraps shared memory regions between applications and Pony Express, using the ancillary data features of domain sockets to pass tmpfs-backed file descriptors between processes. One such shared memory region implements the command and completion queues for asynchronous operations. When an application wishes to invoke an operation, it writes a command into the command queue. Application threads can then either spin-poll the completion queue, or can request to receive a thread notification when a completion is written. Other shared memory regions map to application regions for zero-copy request/response payloads.

Pony Express implements custom memory allocators to optimize the dynamic creation and management of state, which includes streams, operations, flows, packet memory, and application buffer pools. These structures are appropriately charged back to application memory containers using the mechanisms discussed in Section 2.5.

Transport design: Pony Express separates its transport logic into two layers: an upper layer implements the state machines for application-level operations and a lower layer implements reliability and congestion control. The lower layer implements reliable *flows* between a pair of engines across the network and a *flow mapper* maps application-level connections to flows. This lower layer is only responsible for reliably delivering individual packets whereas the upper layer handles reordering, reassembly, and semantics associated with specific operations. The congestion control algorithm we deploy with Pony Express is a variant of Timely [45] and runs on dedicated fabric QoS classes. The ability to rapidly deploy new versions of Pony Express significantly aided development and tuning of congestion control.

With Pony Express, we periodically extend and change our internal wire protocol while maintaining compatibility with prior versions during the time horizon where multiple release versions may exist in our fleet. Although our weekly release cycle makes this time horizon small, we nonetheless still require interoperability and backwards compatibility as we transition users. We currently use an out-of-band mechanism (a TCP socket) to advertise the wire protocol versions available when connecting to a remote engine, and select the

least common denominator. Once the fleet has turned over, we subsequently remove the code for unused versions.

Engine operation: A Pony Express engine services incoming packets, interacts with applications, runs state machines to advance messaging and one-sided operations, and generates outgoing packets. When polling NIC receive queues, the maximum number of packets processed is configurable to vary the latency vs. bandwidth trade-off—our current default is 16 packets per batch. Polling application command queues similarly uses a configurable batch size. Based on incoming packets and commands from applications, as well as the availability of NIC transmit descriptor slots, the engine produces new packets for transmission. This just-in-time generation of packets based on slot availability ensures we generate packets only when the NIC can transmit them (there is no need for per-packet queueing in the engine). Throughout, work scheduling provides tight bounds on engine execution time, implements fairness, and opportunistically exploits batching for efficiency.

Applications using Pony Express can either request their own *exclusive* engines, or can use a set of pre-loaded *shared* engines. As discussed in Section 2, engines are the unit of scheduling and load balancing within Snap, and thus, with application-exclusive engines, an application receives stronger performance isolation by not sharing the fate of CPU and scheduling decisions with other applications, but at potentially higher CPU and memory cost. Applications use shared engines when strong isolation is less important. This has economic advantages and also reduces the challenges associated with scaling to a high number of engines, which is an area of ongoing work.

3.2 One-Sided Operations

One-sided operations do not involve any application code on the destination, instead executing to completion entirely within the Pony Express engine. Avoiding the invocation of the application thread scheduler (i.e., Linux CFS) to dispatch, notify, and schedule a thread substantially improves CPU efficiency and tail latency. An RDMA Read is a classic example of a one-sided operation, and this section covers how Pony Express enables similar benefits.

Unlike RPC systems, which enable applications to write handlers that implement arbitrary user logic, one-sided operations in Pony Express must be pre-defined and pre-installed as part of the Pony Express release. We do not allow arbitrary application-defined operations and instead follow a feature request and review process. Since the one-sided logic executes in the address space of Snap, applications must explicitly share remotely-accessible memory even though their threads do not execute the logic. While it is certainly possible to avoid the invocation of the application thread scheduler for every operation by using spin-polling application threads in conjunction with two-sided messaging, there are downsides to a

spin-polling thread for every application (discussed further in Section 6.1).

The software flexibility of Pony Express enables richer operations that go beyond basic remote memory read and write operations. As noted in prior work [33, 38], the efficiency advantages of RDMA can disappear with remote data structures that require multiple network round-trips to traverse. As an example, we support a custom *indirect read* operation, which consults an application-filled indirection table to determine the actual memory target to access. Compared to a basic remote read, an indirect read effectively doubles the achievable operation rate and halves the latency when the data structure requires a single pointer indirection. Another custom operation is a *scan and read*, which scans a small application-shared memory region to match an argument and fetches data from a pointer associated with the match. Both of these operations are used in production systems.

3.3 Messaging and Flow Control

Pony Express provides send/rcv messaging operations for RPC and non-RPC use cases. As with HTTP2 and QUIC [40], we provide a mechanism to create message streams to avoid head-of-line blocking of independent messages. Rather than a per-connection receive window of bytes (as found in TCP sockets), flow control is based on a mix of receiver-driven buffer posting as well as a shared buffer pool managed using credits, for smaller messages.

Flow control for one-sided operations is more subtle, as there are no application-visible mechanisms to stop initiators from hitting a machine with a never-ending stream of read requests, for example. That is, unlike byte streaming sockets and two-sided messaging, an application cannot pause a sender by refusing to read from a socket or post memory to receive a message. While one-sided clients can ameliorate server overload by limiting outstanding operations on a connection or with other client-side backoff strategies, we ultimately address this problem with the Snap CPU scheduling and accounting mechanisms discussed in Section 2. In particular, users of one-sided operations instantiate a Pony Express engine that receives a fair allocation of CPU time. If clients overrun this engine with one-sided operations, inevitably request packets get dropped and congestion control backs off. Thus, while our one-sided users typically avoid these scenarios, one-sided operations fall back to relying on congestion control and CPU scheduling mechanisms rather than higher-level flow control mechanisms for fair sharing.

3.4 Hardware Offloads

The pursuit of offloads and other accelerators is strategically important as Moore's Law slows. Pony Express exploits stateless offloads, including the Intel I/OAT DMA device [2] to offload memory copy operations. We developed a kernel module to make this capability available to Snap and found the asynchronous interactions around DMA to be a natural fit for Snap, with its continuously-executing packet processing

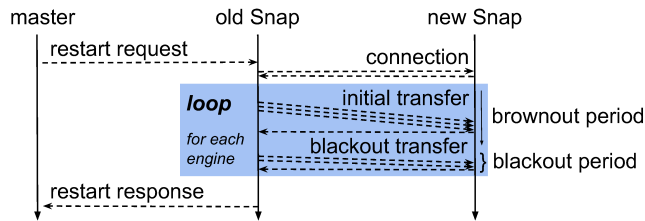


Figure 5. Transparent upgrade sequence diagram.

pipelines, compared to Linux’s synchronous socket system calls. Section 5.1 quantifies the overall CPU efficiency improvement from copy offload for Pony Express. Pony Express also exploits other stateless NIC offloads; one example is an end-to-end invariant CRC32 calculation over each packet. Section 6.5 discusses the use of stateful offloads and how we see them interacting with Pony Express.

4 Transparent Upgrades

Snap substantially enhances our ability to innovate in host networking. A critical enabler is our ability to release new versions of Snap without disrupting running applications. This section describes our approach.

An initial goal was to afford developers maximum flexibility in making substantial changes between releases (keeping API and wire format compatibility issues in mind). This precluded schemes that kept binary state in memory while changing out code. Instead, during upgrades, the running version of Snap serializes all state to an intermediate format stored in memory shared with a new version. As with virtual machine migration techniques [18], the upgrade is two-phased to minimize the *blackout* period when the network stack is unavailable. The initial *brownout* phase performs a preparatory background transfer that has minimal performance impact.

A second goal was to minimize disruption to applications, both in terms of writing code to handle upgrade and loss of connectivity. We target a blackout period of 200msecs or less, and in order to meet this target, Snap performs upgrades incrementally, migrating engines one at a time, each in its entirety. As the number of engines running in production increased, this approach became necessary to protect a single engine from experiencing prolonged blackout due to the transfer of other engines. Beyond this, with a gradual upgrade process across a cluster, we have found that our existing applications do not notice hundred millisecond blips in communication that occur once per week. Packet loss may occur during this blackout period, but end-to-end transport protocols tolerate this as if it were congestion-caused packet loss. Importantly, authenticated application connections remain established (on both ends) and state such as bidirectional message streams remain intact.

Figure 5 illustrates the upgrade flow. First, a Snap “master” daemon launches a second instance of Snap. The running Snap instance connects to it and then, for each engine one at a

time, suspends control plane Unix domain socket connections and transfers them in the background along with shared memory file descriptor handles. This is accomplished using the ancillary fd-passing feature of Unix domain sockets. While control plane connections are transferred, the new Snap re-establishes shared memory mappings, creates queues, packet allocators, and various other data structures associated with the new instance of the engine while the old engine is still operating. Upon completion, the old engine begins the blackout period by ceasing packet processing, detaching NIC receive filters, and serializing remaining state into a tmpfs shared memory volume. The new engine then attaches identical NIC filters and deserialize state. Once all engines are transferred this way, the old Snap is terminated.

5 Evaluation

This section evaluates the performance and dynamic scalability of Snap, focusing on the Pony Express communication stack. We measure the isolation and fairness properties of Snap running Pony Express (“Snap/Pony”) and also demonstrate our incremental state transfer for transparent upgrades. All of our results below report CPU usage with hyperthreading enabled such that a single reported “core” or “CPU” refers to a single hardware thread context. In Sections 5.1–5.3, we compare against the Linux kernel TCP/IP stack, not only because it is the baseline at our organization but also because kernel TCP/IP implementations remain, to our knowledge, the only widely-deployed and production-hardened alternative for datacenter environments. In Section 5.4, we also provide qualitative comparisons against a hardware RDMA technology available within our organization.

5.1 Baseline Throughput and Latency

To establish the baseline performance of Snap/Pony, we start with performance measurements between a pair of machines connected to the same top-of-rack switch. The machines are equipped with Intel Skylake processors and 100Gbps NICs. We compare against kernel TCP sockets using the Neper [3] utility (similar to Netperf but with support for flowing data over multiple simultaneous streams), while for Snap/Pony we use a custom (non-sockets) benchmark configured similarly. All benchmarks employ a single application thread for sending and receiving—we look at scaling performance with multiple processes and threads in the following section. The Pony Express engine always spins and for measuring end-to-end latency we vary whether or not the application thread spins.

Table 1 shows that the baseline, single-stream throughput of TCP is 22Gbps on our kernel configuration using Neper. Also shown is the measured average core utilization across both application and kernel, which is about 1.2 cores. By contrast, Snap/Pony delivers 38Gbps using 1.0 Snap cores and 0.05 application cores. Recently we have also started to

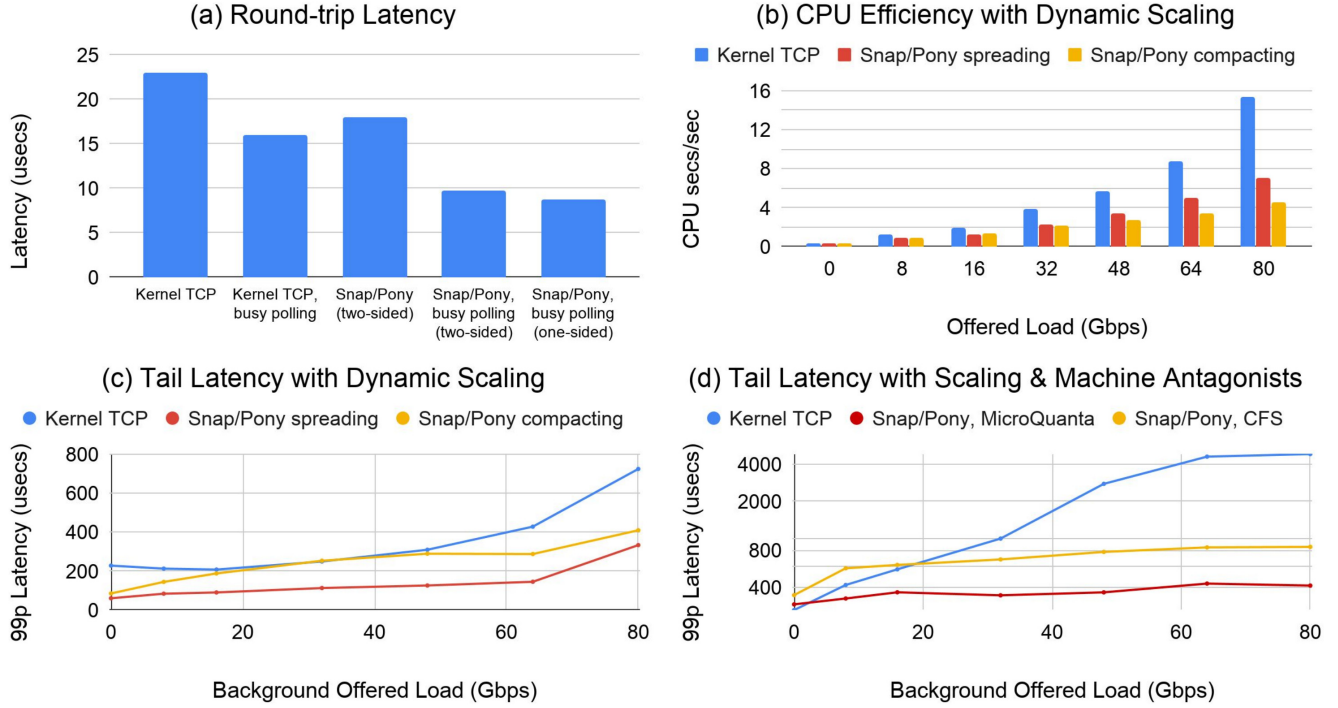


Figure 6. Benchmark data. Graph (a) shows mean latency between two machines connected to the same top-of-rack switch. Graphs (b), (c), and (d) show per-machine CPU time, probe tail latency, and the latency impact of background machine antagonists, respectively, for an all-to-all RPC benchmark as offered load is increased. Graphs (b) and (c) compare scheduling modes while (d) compares MicroQuanta versus optimized Linux CFS. All graphs compare against Linux TCP and in all graphs Gbps is bidirectional, since each job both requests and responds to RPCs.

experiment with a larger MTU size on our switches and fabrics. We show that a 5000B MTU¹ increases the single-core throughput of Snap/Pony to over 67Gbps, and enabling I/OAT receive copy offload (to complement zero-copy transmit) further increases throughput to over 80Gbps using a single core. Finally, Table 1 shows that, while TCP performance degrades substantially as the number of simultaneously active streams increases, Snap/Pony demonstrates robust connection scaling by avoiding heavy context switching and maintaining good locality on this benchmark.

Figure 6(a) illustrates the average round-trip latency between two applications sending and receiving a small message (two-sided). In this configuration, *TCP_RR* in Neper shows TCP provides a baseline latency of 23μsecs. A similar application using Snap/Pony delivers latency of 18μsecs. Configuring the application to spin-poll on the Pony Express completion queue reduces latency to less than 10μsecs whereas using the busy-polling socket feature in Linux reduces *TCP_RR* latency to 18μsecs. We also show the Snap/Pony latency achieved with a one-sided access, which further reduces latency to 8.8μsecs.

¹We chose 5000B in order to comfortably fit a 4096B application payload with additional headers and metadata.

	# streams	CPU/sec	Gbps
Linux TCP	1 stream	1.17	22.0
	200 streams	1.15	12.4
Snap/Pony	1 stream	1.05	38.5
	200 streams	1.05	39.1
Snap/Pony w/ 5kB MTU	1 stream	1.05	67.5
	200 streams	1.05	65.7
Snap/Pony w/ 5kB+I/OAT	1 stream	1.05	82.2
	200 streams	1.05	80.5

Table 1. Throughput measured for TCP (using Neper), and Snap/Pony (using an internal tool). All tools use a single application thread to drive the load.

5.2 Scaling and Performance Isolation

This section evaluates Snap/Pony performance when scaling beyond a single core. The experiment uses a rack of 42 machines all connected to the same switch. Each machine is equipped with a 50Gbps NIC and Intel Broadwell processors in a dual-socket configuration. We schedule 10 background jobs on each machine where each job communicates over RPC at a chosen rate with a Poisson distribution. Each RPC chooses one of the 420 total jobs at random as the target and requests a 1MB (cache resident) response with no additional computation. The offered load is varied by increasing the

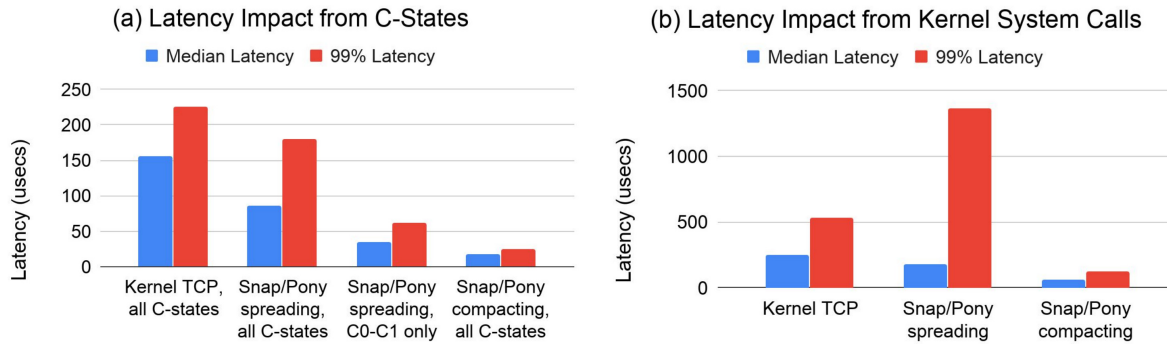


Figure 7. Latency degradation from system-level effects. Graph (a) shows the latency impact of low-power C-states for an RPC benchmark running at low QPS and on an otherwise idle machine, while graph (b) shows the latency impact of a harsh antagonist that spawns threads to continually perform `mmap()` and `munmap()` system calls.

request rate of each job. The measured CPU time includes all application, system, and interrupt time. The MTU size for Snap/Pony is 5000B. For TCP, it is 4096B, which is the analogous “large MTU” setting for TCP at our organization; regrettably, the values are different, but close enough that we believe the data is still representative. In addition to the 10 jobs generating background load with 1MB RPCs, we also schedule a single latency prober job on each machine that similarly chooses a random destination but measures the latency of only a tiny RPC. We report the 99th percentile latency of these measurements. In the case of Snap/Pony, each job requests its own exclusive engine, and we evaluate the two scheduling modes from Section 2.4, “spreading engines” and “compacting engines”.

Figure 6(b) shows the overall per-machine CPU usage and Figure 6(c) shows the 99th percentile prober latency as the total background load increases to 80Gbps per machine (which includes traffic in both directions on the 50Gbps NICs). We first observe that both Snap engine schedulers succeed in scaling CPU consumption in proportion to load. Next, Snap generally shows sub-linear increase in CPU consumption, a consequence of batching efficiencies. Finally, the relative CPU time differences between Snap and TCP increases as dynamic scaling mechanisms are stressed. At an offered load of 8Gbps, CPU time across both TCP and Snap is comparable, but at an offered load of 80Gbps, Snap is over 3x more efficient than TCP. We attribute the performance improvement due to a combination of copy reduction, avoiding fine-grained synchronization, the 5000B vs. 4096B MTU difference, and in hotter instruction and data caches in the case of the Snap compacting scheduler. Notably, for these larger 1MB RPCs, the cost of kernel TCP socket system calls amortizes well and avoiding them does not show performance gain. The Gbps/core of Snap/Pony is less than with the peak single-core throughput of 82Gbps illustrated in Table 1 because work is spread to multiple engines, reducing batching and introducing previously absent scheduler overheads.

While the Snap compacting scheduler offers the best CPU efficiency, the spreading scheduler has the best tail latency as illustrated in Figure 6(c). The lower CPU efficiency under the spreading scheduler comes from time spent in interrupt and system contexts, whereas the majority of scheduler time in the compacting scheduler is in the user context and is overall lower. This experiment does not show performance loss experienced by bystander applications due to context switching. This effect is higher for the spreading scheduler and for TCP softirq processing, which both rely on application interrupts, than for the compacting scheduler, which consolidates work onto the fewest cores and minimizes application interference.

Figure 6(d) shows the impact of loading the machine with background antagonist compute processes, running Snap engines using the spreading scheduler, and comparing the MicroQuanta kernel scheduling class to Linux CFS with a niceness of -20 (its most aggressive setting). The background antagonists run with reduced priority relative to the load-generating network application jobs and continually wake threads to perform MD5 computations. They place enormous pressure on both the hardware (e.g., DRAM, cache) and software scheduling systems, and thus substantially increase the overall tail latency compared to the Figure 6(c) runs without the antagonists, particularly for the non-MicroQuanta cases.

5.3 System-Level Interrupt Latency Impacts

While Figures 6(c) and (d) shows favorable latency results under the spreading scheduler, this scheduler relies on interrupts to wake on idleness, which is prone to a number of additional system-level latency contributors in practice. First, an interrupt generated by the NIC may target a core in a deep power-saving C-state. Second, our production machines run complex antagonists that can affect the schedulability of even a MicroQuanta thread. We illustrate these effects in Figures 7(a) and 7(b), in which we run the same 42-machine all-to-all RPC benchmark as previously, but running only the prober job, running it only at 1000 QPS (one RPC per millisecond), and with the prober application thread spin-polling to isolate application thread wakeup from transport wakeup.

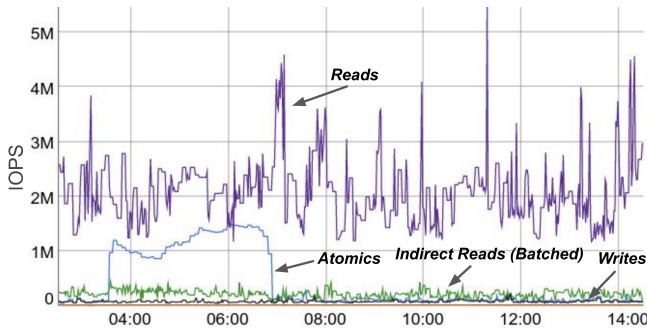


Figure 8. Production dashboard illustrating the rate of IOPS served by the hottest machine over each minute interval. Some intervals show a single Snap/Pony engine and core serving upwards of 5M IOPS.

First, Figure 7(a) shows that, running a low QPS benchmark on an otherwise unloaded machine, both kernel TCP and the Snap spreading scheduler see remarkably worse latency than the prior two-machine (closed loop) ping-pong latency result in 6(a) due to C-state interrupt wakeup latency. The Snap compacting scheduler avoids this wakeup cost because its most compacted, least-loaded state spin-polls on a single core by default (expanding to more cores as needed).

Second, Figure 7(b) shows the impact of running a harsh antagonist that spawns threads to repeatedly `mmap()` and `munmap()` 50MB buffers. While perhaps an uncommon and unrealistic scenario, this demonstrates a pathology found in many Linux kernels in which certain code regions cannot be preempted by any userspace process. Compacting engines provides the best latency because, in this benchmark, engine work compacts down to a single spin-polling core that does not time-share with the antagonist. We further discuss ongoing challenges around CPU scaling in Section 6.3.

5.4 RDMA and One-sided Operations

We see significant application-level performance improvements from leveraging the one-sided operations available in Snap/Pony. Conventional RPC stacks written on standard TCP sockets, like gRPC, see less than 100,000 achievable IOPS/core [4]. Figure 8 shows a production dashboard snapshot of a service using Snap/Pony for one-sided data access. The service supports a large-scale distributed data analytics system. This workload demands up to 5 million remote memory accesses per second served with a single Snap/Pony dedicated core. Many of the operations use a custom batched indirect read operation, in which Snap/Pony consults a table to perform a batch of eight indirections locally rather than requiring those indirections to occur over the network.

Relative to a hardware RDMA implementation that relied on fabric flow control, switching to Snap/Pony doubled the production performance of the data analytics service. This was primarily due to relaxation of rate limits previously put in place to prevent excessive fabric back-pressure when individual RDMA NICs became overloaded during hot-spotting.

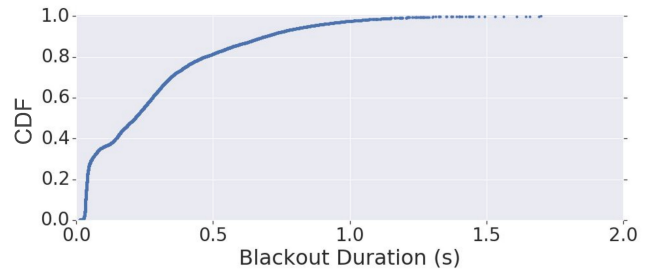


Figure 9. Transparent upgrade blackout duration data from a large and representative production cluster.

Hardware RDMA implementations typically implement small caches of connection and RDMA permission state, and access patterns that spill out of the cache result in significant performance cliffs. A “thrashing” RDMA NIC emits fabric pauses, which can quickly spread to other switches and servers. This led us to implement a cap of 1M RDMA/sec per machine and credits were statically allocated to each client. Switching to Snap/Pony allowed us to remove these caps, to increase IOP rates, and to rely on congestion control on lossy fabrics to handle transient hot-spotting. Then switching to our custom indirect read operation with integrated batching yielded another significant performance boost.

5.5 Transparent Upgrades

Figure 9 shows production statistics for a transparent upgrade performed in one of our production cells, supporting internal services, on 2019-01-18. The median blackout duration is 250ms, which is slightly above the goal of 200ms we set when starting the project. The latency distribution is heavy-tailed, and strongly correlates with the amount of state checkpointed. Internal application owners do not notice the blips that occur once per week, although we are exploring ways to further reduce blackout duration through more incremental transfer steps.

6 Experiences, Challenges, and Future Work

This section discusses our experiences in building and deploying Snap and related networking technologies. We also discuss ongoing work and recent challenges.

6.1 In-Application Transports

An alternative to kernel TCP or Pony Express is a transport that operates entirely in the address space of an application process, using either frameworks such as netmap [55] or DPDK [1] for high-performance use cases, or UDP sockets for lower-performance WAN use cases like QUIC [40]. Such an in-application transport design enables applications to run transport code in their own thread contexts, which can enable faster CPU scale-out in response to bursts and can improve latency by avoiding system calls (kernel TCP) or the cross-core hop between application threads and network threads (Pony Express). Indeed, we experimented with this approach, but found that, despite the benefits, it imposed too many manageability and performance drawbacks.

First and most important, our transparent upgrade approach described in Section 4 relies on running the transport as a separate process. With well-managed weekly releases of Snap, we can quickly make wire protocol changes to systems like Pony Express without needing to speak all wire formats of any client library that is installed anywhere. While some of our techniques could apply to in-application transports with extensions for dynamic linking, binary upgrades would still be at the whim of a large number of individual service owners.

Second, we find that datacenter transports are sensitive to careful CPU provisioning and scheduling, while transports that run in an application container encounter a complicated set of threading and CPU provisioning issues. A spin-polling transport thread with a provisioned core in every application could help ameliorate scheduling unpredictability, but considering that it is common for us to run dozens of individual applications on a single machine, this approach is not practical in general. At the same time, relying on interrupt-driven wakeup of an application transport thread can see unpredictable scheduling delays. Non-responsive transports cause collateral damage, for instance, when well-provisioned servers see excessive transport timeouts and needless retransmissions because packets that have successfully reached a client are stuck waiting for CPU service. Interrupt-driven Snap leverages our MicroQuanta kernel scheduling class to help bound the scheduling delay, but this is a privileged scheduling class that cannot be used by arbitrary internal applications without special policy and arbitration.

6.2 Memory Mapping and Management

One significant challenge with non-kernel networking lies with memory management. This is true not only for the Snap approach, but also for offload stacks like hardware RDMA and for in-application userspace stacks.

Ultimately, applications wish to send and receive data in and out of their heaps. While the Linux kernel can copy directly to and from any application-specified virtual memory address, and also implement zero-copy by translating through an application's page table and temporarily pinning application buffers, Snap does not have access to application page tables and Linux cannot share arbitrary anonymous-backed heap memory between processes. In Linux, any memory shared between applications and Snap must be backed by tmpfs or memfd [6]. We considered modifying our heap implementation to back all application heap memory with tmpfs, and to then coordinate with Snap to map everything into the Snap address space (and page table), but we did not implement this due to concerns around TLB shootdowns, as any application address space modification can result in inter-processor interrupts targeting the core(s) running Snap. Moreover, our heap implementation frequently returns memory to the kernel to account for phase behavior, which can result in frequent remappings.

Hardware RDMA NICs, along with OS bypass, encounter similar problems because they also do not have access to application page tables and rely on long-lived memory pinning and I/O page tables. While many MPI stacks provide an alternative heap that transparently registers heap memory with NICs, fluid address space modifications in our environment may require excessive coordination with a NIC (or IOMMU).

Pony Express does register some application-shared memory with the NIC for zero-copy transmit; but we do so selectively and some of our current applications take a copy from application heap memory into bounce buffer memory shared and registered with Snap (and vice versa). We are exploring techniques to avoid these copies, for instance, if the cost of a custom system call to translate and temporarily pin the memory is cheaper than a memory copy.

6.3 Dynamic CPU Scaling

Scheduling and scaling of CPU for Snap is an area of ongoing research and development. In this paper we presented three different approaches: dedicated cores, spreading engines, and compacting engines. Our initial Snap deployments used dedicated cores, in part because of simplicity but also because static provisioning meets performance SLOs up to a known load threshold, and thus many deployments continue to use dedicated cores today. However, as we seek to grow usage of Snap, particularly with Pony Express as a general-purpose TCP/IP replacement, dynamic provisioning of host networking CPU resources has become paramount; we find that kernel TCP networking usage is very bursty, sometimes consuming upwards of a dozen cores over short bursts. Although our performance evaluation demonstrated that our compacting engines scheduler provides the best overall CPU efficiency, and provides the lowest latency at low loads and in the presence of system call antagonists, we also showed that spreading engines can offer better tail latency at higher Snap loads because of the delay in reacting to engine queue build-up. We are currently working towards achieving the best of both worlds by combining elements of both types of schedulers, and continuing to refine our kernel support.

6.4 Rebalancing Across Engines

As discussed in Section 2.2, engines are single-threaded, which steers Snap developers away from the overheads of fine-grained synchronization and towards siloing state as much as possible. Pony Express engines, for example, do not currently share flow state between engines, and flows also do not currently rebalance between engines. As a consequence, however, serialization of flow processing across flows within an engine can limit performance. Our client libraries implement a variety of different mechanisms to spread load across multiple engines, ranging from simple sharding of flows across engines to duplicating flows and implementing rebalancing above the Pony Express layer. Nonetheless, we are considering the possible need for fine-grained rebalancing across engines using mechanisms such as work stealing [48, 51].

6.5 Stateful Offloads

Our early experience with an OS-bypass RDMA stack implemented entirely in hardware and directly accessed by uncoordinated application libraries did not scale and did not have the iteration velocity required. Nonetheless as Moore's Law slows we continue to explore the next generation of stateful offloads as they overcome hurdles for datacenter adoption. Going forward, we see several benefits in leveraging Snap as a centralized management layer for accessing future hardware offloads. First, it allows us to deploy workarounds for hardware and firmware bugs frequently encountered in complex, stateful NICs. Second, it can flexibly implement the traditional role of an OS in providing multiplexing and management policies. Third, providing robust and secure virtual functions for complex hardware offloads is challenging and Snap can provide a transparent fallback mechanism when needed. Finally, decoupling applications from specific hardware allows us to transparently migrate applications between machines irrespective of the underlying hardware.

7 Related Work

In the space of networking as a separate host service, TAS [35], IsoStack [57], and SoftNIC [26] dedicate cores for efficiency and low latency, with a focus on performance through minimization, spin-polling, and the elimination of data structure sharing. At the same time, in the space of library OS solutions, IX [14, 52], ZygOS [51], and Shinjuku [31] develop high-performance systems that service a single application in a single address space, integrating application processing handlers into the transport itself and assuming dedicated NIC resources. ZygOS diverges from IX by incorporating task stealing with intermediate buffering, and Shinjuku adds intelligent preemption; however, all three efforts sidestep problems around inter-application sharing and arbitration, and require Dune virtualization [13]. None of the above efforts tackle efficiency and low latency with fine-grained dynamic CPU scaling, transparent upgrades, or achieve the full generality of what Snap provides in a general-purpose, multi-application production and/or cloud environment.

Other recent approaches take advantage of lightweight runtimes and kernel bypass to achieve high performance, both inside academia [30, 32, 42, 48, 50, 53] and outside [1, 55]. Perhaps the most similar to our work is Shenango [48], which tackles dynamic CPU scaling even more aggressively than our work by dedicating an entire core to thread scheduling and packet steering functions. However, Shenango requires a custom application scheduling runtime and requires partitioning cores away from the operating system. Snap uniquely integrates with a general purpose host through its MicroQuanta kernel scheduling class and also uniquely addresses transparent upgrades.

On the transport side, mixed onload/offload approaches have been previously proposed for TCP [54, 56]. There is also abundant recent work on datacenter transports [11, 17,

24, 27, 46, 49], although much of it requires fabric modification, which is not in scope for this work. Like Pony Express running on machines without RDMA offload, FaSST [34] implements RDMA using two-sided datagram exchange, and also like Pony Express, LITE [60] exposes RDMA through a software indirection layer, but implemented in the kernel rather than in userspace.

In terms of kernel design, FlexSC [58] is similar to Snap in leveraging modern multicore hardware and delegating system calls to separate threads from application threads, but it does not propose a microkernel-like approach to doing so. At the same time, microkernels have a decades-long history [10, 15, 16, 20, 23, 28, 29, 41, 61]. Most relevant is the work of Thekkath *et al.* [59], which discusses the trade-offs implementing network protocols in a monolithic kernel versus a microkernel approach. They ultimately dismiss the latter, citing domain switching overheads. However, as previously discussed, these overheads have diminished significantly with multi-core systems and fine-grained memory sharing mechanisms in modern server architectures.

8 Conclusions

We present the design and architecture of Snap, a widely-deployed, microkernel-inspired system for host networking. Snap enables numerous teams to develop high-performance packet processing systems with high developer and release velocity. We describe, in detail, a communication stack based on this microkernel approach and how we transparently upgrade the communication stack without draining applications from running machines. Our evaluation demonstrates up to 3x improvement in Gbps/core efficiency, sub-10-microsecond latency, dynamic scaling capability, millions of one-sided operations per second, and CPU scheduling with a customizable emphasis between efficiency and latency. Snap has been running in production for 3 years supporting the extensible communication needs of several critical systems. Our experience indicates that the Snap release velocity has essentially been a requirement for deploying these services because of the needed iteration speed in production. The performance gains are a corollary benefit, enabled by aggressive deployment of novel application structures that bring to bear advanced one-sided and low-latency messaging features in a way that would otherwise not have been possible.

Acknowledgements

We would like to thank our reviewers, shepherd Irene Zhang, Jeff Mogul, Thomas Wenisch, Aditya Akella, Mark Hill, Neal Cardwell, and Christos Kozyrakis for helpful paper feedback; Luiz Barroso and Sean Quinlan for encouragement throughout; Joel Scherplez and Emily Blem for some of the earliest exploratory work on the project; We also thank Bill Vareka, Larry Greenfield, Matt Reid, Steven Rhodes, Anish Shankar, Alden King, Yaogong Wang, and the large supporting cast at Google who helped realize this project.

References

- [1] Data plane development kit. <http://www.dpdk.org>.
- [2] Fast memcopy with SPDK and intel IOAT DMA engine. <https://software.intel.com/en-us/articles/fast-memcpy-using-spdk-and-ioat-dma-engine>.
- [3] Github repository: Neper linux networking performance tool. <https://github.com/google/neper>.
- [4] grpc benchmarking. <https://grpc.io/docs/guides/benchmarking.html>.
- [5] Linux CFS scheduler. <https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt>.
- [6] memfd manpage. http://man7.org/linux/man-pages/man2/memfd_create.2.html.
- [7] Nice levels in the linux scheduler. <https://www.kernel.org/doc/Documentation/scheduler/sched-nice-design.txt>.
- [8] Scaling in the linux networking stack. <https://www.kernel.org/doc/Documentation/networking/scaling.txt>.
- [9] Short waits with umwait. <https://lwn.net/Articles/790920/>.
- [10] M. J. Accetta, R. V. Baron, W. J. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference, Atlanta, GA, USA, June 1986*, pages 93–113, 1986.
- [11] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 435–446, New York, NY, USA, 2013. ACM.
- [12] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 45–58, Berkeley, CA, USA, 1999. USENIX Association.
- [13] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association.
- [14] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 49–65, Berkeley, CA, USA, 2014. USENIX Association.
- [15] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8(1):37–55, Feb. 1990.
- [16] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 120–133, New York, NY, USA, 1993. ACM.
- [17] L. Chen, K. Chen, W. Bai, and M. Alizadeh. Scheduling mix-flows in commodity datacenters with karuna. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 174–187, New York, NY, USA, 2016. ACM.
- [18] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.
- [19] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zerneno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCaboote, M. de Kruijf, N. Hua, N. Lewis, N. Kasi-nadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 373–387, Renton, WA, 2018. USENIX Association.
- [20] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 122–136, New York, NY, USA, 1991. ACM.
- [21] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 261–275, New York, NY, USA, 1996. ACM.
- [22] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilengiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 523–535, Berkeley, CA, USA, 2016. USENIX Association.
- [23] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.
- [24] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. pHost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 1:1–1:12, New York, NY, USA, 2015. ACM.
- [25] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. KASLR is dead: Long live KASLR. In *Engineering Secure Software and Systems - 9th International Symposium, ESSoS 2017, Proceedings, volume 10379 LNCS of Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 161–176, Italy, 2017. Springer-Verlag Italia.
- [26] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A software NIC to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.
- [27] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 29–42, New York, NY, USA, 2017. ACM.
- [28] P. B. Hansen. The nucleus of a multiprogramming system. *Commun. ACM*, 13(4):238–241, Apr. 1970.
- [29] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ -kernel-based systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 66–77, New York, NY, USA, 1997. ACM.
- [30] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, 2014. USENIX Association.
- [31] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for usecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, 2019. USENIX Association.
- [32] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 1–16, 2019.

- [33] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM.
- [34] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 185–201, Berkeley, CA, USA, 2016. USENIX Association.
- [35] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. TAS: TCP acceleration as an OS service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 24:1–24:16, New York, NY, USA, 2019. ACM.
- [36] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella. Iron: Isolating network-based CPU in container environments. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 313–328, Renton, WA, 2018. USENIX Association.
- [37] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.
- [38] C. Kulkarni, S. Moore, M. Naqvi, T. Zhang, R. Ricci, and R. Stutsman. Splitter: Bare-metal extensions for multi-tenant low-latency storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 627–643, Carlsbad, CA, Oct. 2018. USENIX Association.
- [39] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zemenko, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 1–14, New York, NY, USA, 2015. ACM.
- [40] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasnic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi. The QUIC transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 183–196, New York, NY, USA, 2017. ACM.
- [41] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 175–188, New York, NY, USA, 1993. ACM.
- [42] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 429–444, Berkeley, CA, USA, 2014. USENIX Association.
- [43] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Melt-down: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.
- [44] Z. Mi, D. Li, Z. Yang, X. Wang, and H. Chen. SkyBridge: Fast and secure inter-process communication for microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 9:1–9:15, New York, NY, USA, 2019. ACM.
- [45] R. Mittal, T. Lam, N. Dukkkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. Timely: RTT-based congestion control for the datacenter. In *Sigcomm '15*, 2015.
- [46] B. Montazeri, Y. Li, M. Alizadeh, and J. K. Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. *CoRR*, abs/1803.09615, 2018.
- [47] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 217–231, New York, NY, USA, 1999. ACM.
- [48] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive data-center workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, 2019. USENIX Association.
- [49] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fast-pass: A centralized "zero-queue" datacenter network. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 307–318, New York, NY, USA, 2014. ACM.
- [50] 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. pages 1–16, 2014.
- [51] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 325–341, New York, NY, USA, 2017. ACM.
- [52] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015, pages 342–355, 2015.
- [53] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout. Arachne: Core-aware thread management. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 145–160, Berkeley, CA, USA, 2018. USENIX Association.
- [54] G. J. Regnier, S. Makineni, R. Illikkal, R. R. Iyer, D. B. Minturn, R. Huggahalli, D. Newell, L. S. Cline, and A. P. Foong. TCP onloading for data center servers. *IEEE Computer*, 37(11):48–58, 2004.
- [55] L. Rizzo. Netmap: A novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.
- [56] L. Shalev, V. Makhervaks, Z. Machulsky, G. Biran, J. Satran, M. Ben-Yehuda, and I. Shimony. Loosely coupled TCP acceleration architecture. In *Hot Interconnects*, pages 3–8. IEEE Computer Society, 2006.
- [57] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. IsoStack: Highly efficient network processing on dedicated cores. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.
- [58] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 33–46, Berkeley, CA, USA, 2010. USENIX Association.
- [59] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Trans. Netw.*, 1(5):554–565, Oct. 1993.
- [60] S.-Y. Tsai and Y. Zhang. LITE kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 306–324, New York, NY, USA, 2017. ACM.
- [61] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Commun. ACM*, 17(6):337–345, June 1974.
- [62] K. Yap, M. Motiwala, J. Rahe, S. Padgett, M. Holliman, G. Baldus, M. Hines, T. Kim, A. Narayanan, A. Jain, V. Lin, C. Rice, B. Rogan, A. Singh, B. Tanaka, M. Verma, P. Sood, M. Tariq, M. Tierney, D. Truemic, V. Valancius, C. Ying, M. Kallahalla, B. Koley, and A. Vahdat. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. 2017.