TAS: TCP Acceleration as an OS Service

Antoine Kaufmann
MPI-SWS

Tim Stamler
The University of Texas at Austin

Simon Peter
The University of Texas at Austin

Naveen Kr. Sharma
University of Washington

Arvind Krishnamurthy
University of Washington

Thomas Anderson
University of Washington

ACM Reference Format:
Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP Acceleration as an OS Service. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3302424.3303985

Abstract

As datacenter network speeds rise, an increasing fraction of server CPU cycles is consumed by TCP packet processing, in particular for remote procedure calls (RPCs). To free server CPUs from this burden, various existing approaches have attempted to mitigate these overheads, by bypassing the OS kernel, customizing the TCP stack for an application, or by offloading packet processing to dedicated hardware. In doing so, these approaches trade security, agility, or generality for efficiency. Neither trade-off is fully desirable in the fast-evolving commodity cloud.

We present TAS, TCP acceleration as a service. TAS splits the common case of TCP processing for RPCs in the data-center from the OS kernel and executes it as a fast-path OS service on dedicated CPUs. Doing so allows us to streamline the common case, while still supporting all of the features of a stock TCP stack, including security, agility, and generality. In particular, we remove code and data of less common cases from the fast-path, improving performance on the wide, deeply pipelined CPU architecture common in today's servers. To be workload proportional, TAS dynamically allocates the appropriate amount of CPUs to accommodate the fast-path, depending on the traffic load. TAS provides up to 90% higher throughput and 57% lower tail latency than the IX kernel bypass OS for common cloud applications, such as a key-value store and a real-time analytics framework. TAS also scales to more TCP connections, providing 2.2× higher throughput than IX with 64K connections.

1 Introduction

As network speeds rise, while CPU speeds stay stagnant, TCP packet processing efficiency is becoming ever more important. Many data center applications require low-latency and high-throughput network access to deliver remote procedure calls (RPCs). At the same time, they rely on the lossless, in-order delivery properties provided by TCP. To provide this convenience, software TCP stacks consume an increasing fraction of CPU resources to process network packets.

TCP processing overheads have been known for decades. In 1993, Van Jacobson presented an implementation of TCP common-case receive processing within 30 processor instructions [21]. Common network stacks, such as Linux's, still use Van's performance improvements [1]. Despite these optimizations, a lot of CPU time goes into packet processing and TCP stack processing latencies are high. For a key-value store, Linux spends 7.5μs per request in TCP packet processing. While kernel-bypass TCP stacks bring direct overhead down, they still introduce overhead in other ways. As network speeds continue to rise, these overheads increasingly consume the available CPU time.

We investigate TCP packet processing overhead in the context of modern processor architecture. We find that existing TCP stacks introduce overhead in various ways (and to varying degree): 1. By running in privileged mode on the same processor as the application, they induce system call overhead and pollute the application-shared L1, L2, and translation caches. 2. They spread per-connection state over several cache lines, causing false sharing and reducing cache efficiency; 3. They share state over all processor cores in the machine, resulting in cache coherence and locking overheads; 4. They execute the entire TCP state machine to completion for each packet, resulting in code with many branches that do not make efficient use of batching and prefetching opportunities.

We harken back to TCP's origin as a computationally efficient transport protocol, e.g., TCP congestion control was designed to avoid the use of integer multiplication and division [22]. Although TCP as a whole has become quite complex with many moving parts, the common case data path remains relatively simple. For example, packets sent within the data center are never fragmented at the IP layer, packets are almost always delivered reliably and in order, and timeouts almost never fire. Can we use this insight to eliminate the existing overheads?

We present TCP acceleration as a service (TAS), a light-weight software TCP network *fast-path* optimized for common-case client-server RPCs and offered as a separate OS service to applications. TAS interoperates with legacy Linux TCP endpoints and can support a variety of congestion control protocols including TIMELY [28] and DCTCP [6].

Separating the TCP fast-path from a monolithic OS kernel and offering it as a separate OS service enables a number of streamlining opportunities. Like Van Jacobson, we realize that TCP packet processing can be separated into a common and an uncommon case. TAS implements the fast-path that handles common-case TCP packet processing and resource enforcement. A heavier stack (the *slow path*), in concert with the rest of the OS, processes less common duties, such as connection setup/teardown, congestion control, and timeouts. The TAS fast-path executes on a set of dedicated CPUs, holding the minimum state necessary for common-case packet processing in processor caches. While congestion control policy is implemented in the slow path, it is enforced by the fast path, allowing precise control over the allocation of network resources among competing flows by a trusted control plane. The fast path takes packets directly from (and directly delivers packets to) user-level packet queues. Unprivileged application library code implements the POSIX socket abstraction on top of these fast path queues, allowing TAS to operate transparent to applications.

Beyond streamlining, another benefit of separating TAS from the rest of the system is the opportunity to scale TAS independent of the applications using it. Current TCP stacks run in the context of the application threads using them, sharing the same CPUs. Network-intensive applications often spend more CPU cycles in the TCP stack than in the application. When sharing CPUs, non-scalable applications limit TCP processing scalability, even if the TCP stack is perfectly scalable. Separation not only isolates TAS from cache and TLB pollution of the applications using it, but also allows TAS to scale independently of these applications.

We implement TAS as a user-level OS service intended to accelerate the Linux OS kernel TCP stack. TAS is workload proportional—it acquires CPU cores dynamically depending on network load and can share CPU cores with application threads when less than one CPU is required. We evaluate TAS on a small cluster of servers using microbenchmarks and common cloud application workloads. In particular, we compare TAS' per-packet CPU overheads, latency, throughput, connection and CPU scalability, workload proportionality, and resiliency to packet loss to those of Linux, IX [9], and mTCP [24]. Finally, we evaluate TAS' congestion control performance with TCP-NewReno and DCTCP at scale using simulations.

Within a virtualized cloud context, NetKernel [31] also proposes to separate the network stack from guest OS kernels and to offer it as a cloud service in a separate virtual machine. NetKernel's goal is to accelerate provider-driven network stack evolution by enabling new network protocol enhancements to be made available to tenant VMs transparently and simultaneously. TAS can provide the same benefit, but our focus is on leveraging the separation of fast and slow path to streamline packet processing.

We make the following contributions:

- We present the design and implementation of TAS, a low-latency, low-overhead TCP network fast-path. TAS is fully compatible with existing TCP peers.
- We analyze the overheads of TAS and other state-of-the-art TCP stacks in Linux and IX, showing how they use modern processor architecture.
- We present an overhead breakdown of TAS, showing that we eliminate the performance and scalability problems with existing TCP stacks.
- We evaluate TAS on a set of microbenchmarks and common data center server applications, such as a key-value store and a real-time analytics framework. TAS provides up to 57% lower tail latency and 90% better throughput compared to the state-of-the-art IX kernel bypass OS. IX does not provide sockets, which are heavy-weight [47], but TAS does. TAS still provides 30% higher throughput than IX when TAS provides POSIX sockets. TAS also scales to more TCP connections, providing 2.2× higher throughput than IX with 64K connections.

2 Background

Common case TCP packet processing can be accelerated when split from its uncommon code paths and offered as a separate service, executing on isolated processor cores. To motivate this rationale, we first discuss the tradeoffs made by existing software network stack architectures and TCP hardware offload designs. We then quantify these tradeoffs for the TCP stack used inside the Linux kernel, the IX OS, and TAS.

2.1 Network Stack Architecture

Network stack architecture has a well-established history and various points in the design space have been investigated. We cover the most relevant designs here. As we will see, all designs split TCP packet processing into different components to achieve a different tradeoff among performance, security, and functionality. TAS builds on this history to arrive at its own, unique point in the design space.

Monolithic, in-kernel. The most popular TCP stack design is monolithic and resides completely in the OS kernel. A monolithic TCP stack fulfills all of its functionality in software, as a single block of code. Built for extensibility, it follows a deeply modular design approach with complex inter-module dependencies. Each module implements a different part of the stack's feature set, interconnected via queues, function call APIs, and software interrupts. The stack itself is trusted and to protect it from untrusted application code, a split is made between application-level and stack-level packet processing at the system call interface, involving a processor privilege mode switch and associated data copies for security. This is the design of the Linux, BSD, and Windows TCP network stacks. The complex nature of these stacks leads them to execute a large number of instructions per packet, with a high code and data footprint (§2.2).

Kernel bypass. To alleviate the protection overheads of in-kernel stacks, such as kernel-crossings, software mutliplexing, and copying, kernel bypass network stacks split the responsibilities of TCP packet processing into a trusted control plane and an untrusted data plane. The control plane deals with connection and protection setup and executes in the kernel, while the data plane deals with common-case packet processing on existing connections and is linked directly into the application. To enforce control plane policy on the data plane, these approaches leverage hardware IO virtualization support [24, 34]. In addition, this approach allows us to tailor the stack to the needs of the application, excluding unneeded features for higher efficiency [27]. The downside of this approach is that, beyond coarse-grained rate limiting and firewalling, there is no control over low-level transport protocol behavior, such as congestion response. Applications are free to send packets in any fashion they see fit, within their limit. This can interact badly with the data center's congestion control policy, in particular with many connections.

Protected kernel bypass. To alleviate this particular problem of kernel bypass network stacks, IX [9] leverages hardware CPU virtualization to insert an intermediate layer of protection, running the network stack in guest kernel mode, while the OS kernel executes in host kernel mode. This allows us to deploy trusted network stacks, while allowing them to be tailored and streamlined for each application. However, this approach re-introduces some of the overheads of the kernel-based approach.

NIC offload. Various TCP offload engines have been proposed in the past [12]. These engines leverage various splits of TCP packet processing responsibilities and distribute them among software executing on a CPU and a dedicated hardware engine executing on the NIC. The most popular is TCP chimney offload [2], which retains connection control within the OS kernel and executes data exchange on the NIC. By offloading work from CPUs to NICs, these designs achieve high energy-efficiency and free CPUs from packet processing work. Their downside is that they are difficult to evolve and to customize. Their market penetration has been low for this reason.

Dedicated CPUs. These approaches dedicate entire CPUs to executing the TCP stack [40, 44]. These stacks interact with applications via message queues instead of system calls, allowing them to alleviate the indirect overheads of these calls, such as cache pollution and pipeline stalls, and to batch calls for better efficiency. Barrelfish [8] subdivides the stack further, executing the NIC device driver, stack, and application, all on their own dedicated cores. These approaches attain high and stable throughput via pipeline parallelism and performance isolation among stack and application. However, even when dedicating a number of processors to the TCP stack, executing the entire stack can be inefficient, causing pipeline stalls and cache misses due to complexity.

	Linux		**IX**		**TAS**	
Module	kc	%	kc	%	kc	%
Driver	0.73	4%	0.05	2%	0.09	4%
IP	1.53	9%	0.12	4%	0	0%
TCP	3.92	23%	1.05	39%	0.81	32%
Sockets/IX	8.00	48%	0.76	28%	0.62	24%
Other	1.50	9%	0.00	0%	0.00	0%
App	1.07	6%	0.76	28%	0.68	26%
Total	16.75	100%	2.73	100%	2.57	100%

Table 1. CPU cycles per request by network stack module.

Dedicated fast path. TAS builds on the approaches dedicating CPUs, but leverages a unique split. By subdividing the TCP stack data plane into common and uncommon code paths, dedicating separate threads to each, and revisiting efficient stack implementation on modern processors, TAS can attain higher CPU efficieny. In addition to efficiency, this approach does not require new hardware (unlike NIC offload), protects the TCP stack from untrusted applications (unlike kernel bypass), retains the flexibility and agility of a software implementation (unlike NIC offload), while minimizing protection costs (unlike protected kernel bypass). The number of CPU cores consumed by TAS for this service is workload proportional. TAS threads can also share CPUs with application threads under low load.

2.2 TCP Stack Overheads

To demonstrate the inefficiencies of kernel and protected kernel-bypass TCP stacks, we quantify the overheads of the Linux and IX OS TCP stack architectures and compare them to TAS. To do so, we instrument all stacks using hardware performance counters, running a simple key-value store server benchmark on 8 server cores. Our benchmark server serves 32K concurrent connections from several client machines that saturate the server network bandwidth with small requests (64B keys, 32B values) for a small working set, half the size of the server's L3 cache (details of our experimental setup in §5). We execute the experiment for two minutes and measure for one minute after warming up for 30 seconds.

Linux overheads. Table 1 shows a breakdown of the result. We find that Linux executes 16.75 kilocycles (kc) for an average request, of which only 6% are spent within the application, while 85% of total cycles are spent in the network stack. For each request, Linux executes 12.7 kilo-instructions (ki), resulting in 1.32 cycles per instruction (CPI), 5.3× above the ideal 0.25 CPI for the server's 4-way issue processor architecture. This results in high average per-request processing latency of 8μs. The reason for these inefficiencies is the computational complexity and high memory footprint of a monolithic, in-kernel stack. Per-request privilege mode switches and software interrupts stall processor pipelines; software multiplexing, cross-module procedure calls, and security checks require additional instructions; large, scattered per-connection state increases memory footprint and causes stalls on cache and TLB misses; shared, scattered global state causes stalls on locks and cache coherence, inflated by coarse lock granularity and false sharing; covering all TCP packet processing cases in one monolithic block causes the code to have many branches, increasing instruction cache footprint and branch mispredictions.

We measure these inefficiencies with CPU performance counters [46]. The results are shown in Table 2 and indicate cycles spent retiring instructions, and blocked fetching instructions (frontend bound), fetching data (backend bound), and on bad speculation. We can see that Linux spends an order of magnitude more of these cycles than the application. In particular data fetches weigh heavily. Due to the high memory footprint we encounter many cache and TLB misses.

IX overheads. IX can tailor the network stack to the application, simplifying it substantially. IX executes only 2.73 kc for an average request. IX spends 28% of cycles doing work in the application, while the rest (72%) are spent in the IX network stack. We note that the comparison is not entirely fair, as IX vastly simplifies the socket interface. IX does not support POSIX sockets, instead relying on a custom libevent-based API. With sockets, IX would spend a smaller proportion of cycles in the application. For each request, IX executes 3.3 ki, resulting in a CPI 3.3× above the ideal for the server. This results in an average per-request processing latency of 1.3μs. Privilege mode switches remain for IX and while IX simplifies the TCP stack, it still covers all TCP packet processing cases in one monolithic block, causing its code to access sizeable per-connection state and execute many branches, increasing cache footprint. Table 2 shows that while IX reduces overheads dramatically across the board versus Linux, it still spends many backend bound cycles.

TAS overheads. TAS (with sockets) executes 2.57 kc for an average request, 26% of these in the application, while the rest (74%) are spent in TAS. TAS executes 3.9 ki per request, resulting in a CPI only 2.6× above the ideal. While TAS is not perfect and executes more instructions, these instructions are executed on a separate fast-path, resulting in an average per-request processing latency of 1.2μs, due to less per-connection state, pipeline parallelism, and isolation. Table 2 shows that TAS reduces application frontend overhead by 15%, while reducing TCP stack backend overhead by 32% versus IX. TAS frontend overhead comes primarily from the sockets emulation and is reduced to 168 cycles (4% lower than IX) with a low-level interface. Speculation performance did not improve. TAS spends these cycles on message queues.

Counter	Linux	IX	TAS
CPU cycles	1.1k/15.7k	0.8k/1.9k	0.7k/1.9k
Instructions	12.7k	3.3k	3.9k
CPI	1.32	0.82	0.66
Retiring (cycles)	175/3591	190/753	167/848
Frontend Bound	173/2600	121/175	102/248
Backend Bound	388/9046	402/1005	353/684
Bad Speculation	141/515	48/52	63/129

Table 2. Per request app/stack overheads.

3 Design

In this section we describe the design of TAS, with the following design goals:

- **Efficiency:** Data center network bandwidth growth continues to outpace processor performance. TAS must deliver CPU efficient packet processing, especially for latency-sensitive small packet communication that is the common case behavior for data center applications and services.
- **Connection scalability:** As applications and services scale up to larger numbers of servers inside the data center, *incast*, where a single server handles a large number of connections, continues to grow. TAS must support this increasing number of connections.
- **Performance predictability:** Another consequence of this scale is that predictable performance is becoming as important as high common case performance for many applications. In large-scale systems, individual user requests can access thousands of backend servers [23, 30] causing one-in-a-thousand request performance to determine common case performance.
- **Policy compliance:** Applications from different tenants must be prevented from intercepting and interfering with network communication from other tenants. Thus, TAS must be able to enforce policies such as bandwidth limits, memory isolation, firewalls, and congestion control.
- **Workload proportionality:** TAS should not use more CPUs than necessary to provide the required throughput for the application workloads running on the server. This requires TAS to scale its CPU usage up and down, depending on demand.

TAS has three components: Fast path, slow path, and untrusted per-application user-space stack. All components are connected via a series of shared memory queues, optimized for cache-efficient message passing [8]. The fast path is responsible for handling common case packet exchanges. It deposits valid received packet payload directly in user-space memory. On the send path, it fetches and encapsulates payload from user memory according to per-connection *rate or*

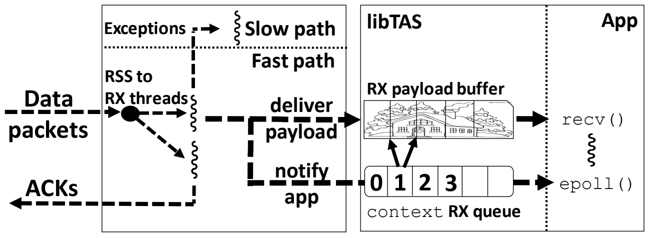


Figure 1. TAS receive flow.

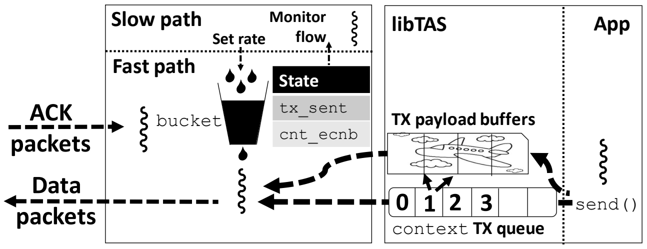


Figure 2. TAS transmit flow.

window limits that are dynamically configured by the slow path. User-level TCP stacks provide the standard POSIX API to applications. Applications do not need to be modified, only (dynamically) relinked. For connection setup and teardown, each user-level stack interacts with the slow path. We describe each component in detail in this section.

3.1 Fast Path

To streamline RPC processing and improve performance, the fast path handles the minimum functionality required for the common-case exchange of RPC packets between untrusted user applications and the network in a data center. To do so, it processes protocol headers, sends TCP acknowledgements, enforces network congestion policy, and performs payload segmentation. To be fully functional, it must also detect exceptions, such as out-of-order arrivals, connection control events, and unknown or unusual packets (such as unusual IP and TCP header options). Data center applications typically pre-establish all required connections. Hence, the TAS fast path handles connection establishment and teardown as exceptions. Exceptions are generally forwarded to the slow path (but see *exceptions*, below).

Common-case receive (Figure 1). The TAS fast path receives TCP packets from the NIC on a dynamic number (cf. Section 3.4) of *receive threads* via the NIC's receive-side scaling (RSS) functionality. TAS assumes that packets are commonly delivered in order by the network. This is true for data center networks today due to connection-stable multi-path routing [17, 43]. With in order packets, the fast path can discard all network headers and directly insert the payload into a user-level, per-flow, circular *receive payload buffer* (identified by `rx_start|size` fields in per-flow state, with `rx_head|tail` identifying write/read positions), notifying an appropriate *receive context queue* (`context` field in per-flow state) by identifying the connection and number of bytes received. The application receives notifications when polling for them (e.g., via `epoll()`) and copies received data out of the payload buffer via socket calls (e.g., `recv()`).

When a payload buffer is full, the fast path simply drops the packet—an uncommon situation, as TCP controls the flow of packets into payload buffers via per-connection window size. If a context queue is full, the fast path will inform the user-level stack upon future packet arrivals when the queue is available again—context queues only fill when payload is queued at an application, providing that the application has work to do and will check the context queue soon. User-level stacks are free to define and configure contexts. We describe them in more detail in Section 3.3. Per-flow payload receive buffers simplify packet handling, flow control, and improve isolation. Calculating an accurate flow control window with shared buffers requires iteration over all connections sharing the buffer, imposing non-constant per-packet overhead. To avoid this overhead, we opt for per-flow payload buffers.

After depositing the payload of an in-order packet, the fast path automatically generates an acknowledgement packet and transmits it to the sender to update its TCP window. Handling TCP acknowledgements in the fast path is important for security. If user-space was given control over acknowledgements, as in many kernel-bypass solutions, it can use it to defeat TCP congestion control [42]. Fast path acknowledgements also provide correct explicit congestion notification (ECN) feedback, and accurate TCP timestamps for round-trip time (RTT) estimation. Finally, the fast path updates local per-connection state (e.g., `seq`, `ack`, and `window` fields).

Common-case send (Figure 2). User-level stacks send data on a flow by appending it to a flow's circular transmit buffer (e.g., when invoked by `send()`). Per-flow send buffers are required to alleviate head-of-line blocking under congestion and flow control. To inform the fast path, the stack issues a TX command on a context queue and wakes a waiting fast path thread. The fast path fills a per-flow bucket (`bucket` field in per-flow state) with the amount of new data to send. Asynchronously, the fast path drains these buckets, depending on a slow path configured per-connection rate-limit or send window size and the receiver's TCP window, to enforce congestion and flow control. When data can be sent, the fast path fetches the appropriate amount from the transmit buffer, produces TCP segments and packet headers for the connection, and transmits. The fast path also uses TCP timestamps to provide the slow path with an accurate RTT estimate for congestion control and timeouts (cf. Section 3.2), among other relevant flow statistics (e.g., `tx_sent` and `cnt_ecnb` fields).

Transmit payload buffer space reclamation. Any payload that has been sent remains in the transmit buffer until acknowledged by the receiver. The fast path parses incoming

Field	Bits	Description
`opaque`	64	Application-defined flow identifier
`context`	16	RX/TX context queue number
`bucket`	24	Rate bucket number
`rx|tx_start`	128	RX/TX buffer start
`rx|tx_size`	64	RX/TX buffer size
`rx|tx_head|tail`	128	RX/TX buffer head/tail position
`tx_sent`	32	Sent bytes from `tx_head`
`seq`	32	Local TCP sequence number
`ack`	32	Peer TCP sequence number
`window`	16	Remote TCP receive window
`dupack_cnt`	4	Duplicate ACK count
`local_port`	16	Local port number
`peer_ip|port|mac`	96	Peer 3-tuple (for segmentation)
`ooo_start|len`	64	Out-of-order interval
`cnt_ackb|ecnb`	64	ACK'd and ECN marked bytes
`cnt_frexmits`	8	Fast re-transmits triggered count
`rtt_est`	32	RTT estimate

Table 3. Required per-flow fast path state (102 bytes).

acknowledgements, updates per-flow sequence and window state, frees acknowledged transmit payload buffer space, and informs user-space of reliably delivered packets by issuing a notification with the number of transmitted bytes for the corresponding flow on an RX context queue (not shown in figures). This requires constant time.

Per-flow state. To carry out its tasks, the fast path requires the per-flow state shown in Table 3. The opaque field is specified by and relayed to user-space to help it identify the corresponding connection. RX/TX buffer state is used for management of per-flow buffers in user-space. The slow path has access to all fast path state via shared memory. In all, we require 102 bytes of per-flow state. Current commodity server CPUs supply about 2MB of L2/3 data cache per core (§4). This allows us to keep the state of more than 20,000 active flows per core in the fast path. Integrating ideas to reduce fast path state (e.g., SENIC [37]) is future work.

Exceptions (Figure 1). Unidentified connections, corrupted packets, out-of-order arrivals, and packets with unhandled flags, are exceptions. Exception packets are filtered and sent to the slow path for processing. The fast path detects out-of-order arrivals by matching arrivals against expected sequence numbers in the per-flow `seq` field. As an optimization, we handle two exceptions on the fast path:

1. When processing incoming acknowledgements the fast path counts duplicates and triggers fast recovery after three duplicates, by resetting the sender state as if those segments had not been sent. The fast path also increments a per-flow retransmit counter to inform the slow path to reduce the flow's rate limit.
2. The fast path tracks one interval of out-of-order data in the receive buffer (starting at `ooo_start` and of length `ooo_len`). The fast path accepts out-of-order segments of the same interval if they fit in the receive buffer. In that case, the fast path writes the payload to the corresponding position in the receive buffer. When an in-order segment fills the gap between existing stream and interval, the fast path notifies the user-level stack as if one big segment arrived, and resets its out-of-order state. Other out-of-order arrivals are dropped and the fast path generates an acknowledgement specifying the next expected sequence number to trigger fast retransmission.

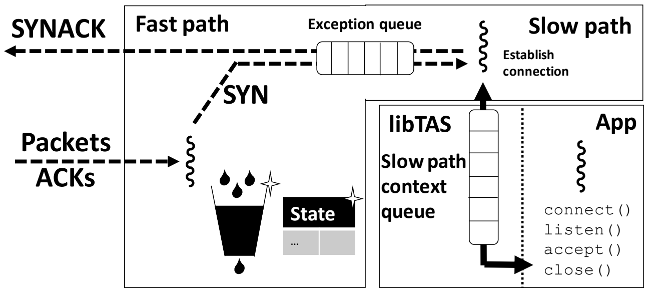


Figure 3. TAS slow path connection control.

3.2 Slow Path

The slow path implements all policy decisions and management mechanisms that have non-constant per packet overhead or are too expensive or stateful to process in the fast path. This includes congestion control policy, connection control, a user-space TCP stack registry, timeouts and other exceptional situations.

Congestion control (Figure 2). TAS supports both rate and window-based congestion control. The fast path runs a control loop iteration for each flow every control interval (by default every 2 RTTs). The slow path retrieves per-connection congestion feedback from the fast path (`cnt_ackb|ecnb`, `cnt_frexmits`, and `rtt_est` fields), then runs a congestion control algorithm to calculate a new flow send rate or window size, and finally updates this information in the fast path via shared memory.

This provides a generic framework to implement different congestion control algorithms. We implement DCTCP [6] and TIMELY [28] (adapted for TCP by adding slow-start). We adapt DCTCP to operate on rates instead of windows by applying its control law (rate-decrease proportional to fraction of ECN marked bytes) to flow rates: during slow start we double the rate every control interval until there is an indication of congestion, and during additive increase we add a configurable step size (10mbps by default) to the current rate. To prevent rates from growing arbitrarily in the absence of congestion, we ensure at the beginning of the control loop that the rate is no more than 20% higher than the flow's send rate.

The choice for rate-based DCTCP for our prototype is deliberate. Rate-based congestion control is more stable with many flows. It smoothes bursts that otherwise occur due to abrupt changes in congestion window size and thus provides a fairer allocation of bandwidth among flows. Our rate-based DCTCP implementation is compatible with Linux peers.

Connection control (Figure 3). Connection control is complex. It includes port allocation, negotiation of TCP options, maintaining ARP tables, and IP routing. We thus handle it in the slow path. User-level TCP stacks issue a `new_flow` command on the slow path context queue to locally request new connections (triggered by a `connect()` call). If granted, the slow path establishes the connection by executing the TCP handshake and, if successful, installs the established flow's state in the fast path and allocates a rate/window bucket. Remote connection control requests are detected by the fast path and forwarded to the slow path, which then completes the handshake.

Servers can listen on a port by issuing a `listen` command to the slow path (triggered by `listen()` socket call). Incoming packets with a SYN flag are forwarded as exceptions to the slow path. The slow path informs user-space of incoming connections on registered ports by posting a notification in the slow path context queue. If the application decides to accept the connection (via `accept()`), its TCP stack may issue the `accept` command to the slow path (via the slow path context queue), upon which the slow path establishes the flow by allocating flow state and bucket, and sending a SYNACK packet. To tear down a connection (e.g., upon `close()`), user-space issues `close`, upon which the slow path executes the appropriate handshake and removes the flow state from the fast path. Similarly, for remote teardowns, the slow path informs user-space via a `close` command.

Retransmission timeouts. We handle retransmission timeouts in the slow path. When collecting congestion statistics for a flow from the fast path, the kernel also checks for unacknowledged data (i.e. `tx_sent > 0`). If a flow has unacknowledged data with a constant sequence number for multiple control intervals (2 by default) the slow path instructs the fast path to start retransmitting by adding a command to the slow path context queue. In response to this command the fast path will reset the flow and start transmitting exactly as described above for fast retransmits.

TCP stack management. To associate new user-space TCP stacks with TAS, the slow path has to be informed via a special system call. If the request is granted, the slow path creates an initial pair of context queues that the user-space stack uses to create connection buffers, etc.

3.3 User-space TCP Stack

The user-space TCP stack presents the programming interface to the application. The default interface is POSIX sockets so applications can remain unmodified, but per-application modifications and extensions are possible, as the interface is at user-level [9, 27, 34]. For example, TAS also offers a low-level API that is similar to the IX networking API. The low-level API directly passes events from the context RX queue to the application and offers functions to add entries to the context TX queue. The TCP stack is responsible for managing connections and contexts. To fulfill our performance goal, common-case overhead of the TCP stack has to be minimal.

Context management. User-space stacks are responsible for defining and allocating contexts. Contexts are useful in various ways, but typically stacks allocate one context per application thread for scalability, as it allows cores to poll (e.g., `epoll()`) only a private context queue, rather than a number of shared payload buffers. Stacks allocate contexts via management commands to the slow path.

3.4 Workload Proportionality

TAS executes protocol processing on dedicated processor cores, and the number of cores needed depends on the workload. As a result, TAS has to dynamically adapt the number of processor cores used for processing to be proportional with the current system load. We implement this with three separate mechanisms. On the fast path, we use hardware and software packet steering to direct packets to the correct cores, while the slow path monitors the CPU utilization and, as needed, adjusts steering tables to add or remove cores. Finally, the fast path blocks threads when no packets are received for a period of time (10 ms in our implementation). These cores can be woken up via kernel notifications (eventfd). This requires us to carefully handle packet re-assignments among queues during scale up/down events.

Fast path. When initializing, TAS creates threads for the configured maximum number of cores and assigns NIC queues and application queues for all cores. Because of the adaptive polling with notifications, cores that do not receive any packets automatically block and are de-scheduled.

We design the data path to handle packets arriving on the wrong TAS core, either from the NIC or the application, with a per-connection spinlock protecting the connection state. This avoids the need for expensive coordination and draining queues when adding or removing cores. Instead we simply asynchronously update NIC and application packet steering to route packets to or away from a specific core. We eagerly update the NIC RSS redirection table to steer incoming packets, and lazily update the routing for outgoing packets from applications. This allows us to be robust during scale up/down events.

Slow path. The slow path is responsible to decide when to add or remove cores by monitoring fast path CPU utilization. If it detects that in aggregate more than 1.25 cores are idle, it initiates the removal of a core. If, on the other hand, less than 0.2 cores are idle in aggregate, it adds a core. The specific thresholds are configuration parameters.

4 Implementation

We have implemented TAS from scratch in 10,127 lines of C code (LoC). TAS' fast path comprises 2,931 LoC. The slow path comprises 3,744 LoC. The user-level library providing the POSIX sockets API is dynamically linked to unmodified application binaries and comprises 3,452 LoC. TAS runs in a user-level process, separate from the applications. Both fast and slow path run as separate threads within this process.

Fast path. The fast path uses DPDK [3] to directly access the machine's NIC, bypassing the Linux kernel. Unlike systems that rely on batching to reduce kernel-user switches, TAS uses a configurable number of dedicated host cores, which we can vary based on the offered network load. Each core replicates a linear packet processing pipeline and exposes a queue pair to the slow path and to each application context to avoid synchronization. The NIC's RSS mechanism ensures that packets within flows are assigned to the same pipeline and not reordered.

Slow path. The slow path runs as a separate thread within the TAS process. To bootstrap context queues, we require applications to first connect to the slow path via a named UNIX domain socket on which the slow path thread listens. Applications use the socket to set up a shared memory region for the context queues. The slow path also uses the socket for automatic cleanup, to detect when application processes exit by receiving a hangup signal via the corresponding socket.

4.1 Limitations

Fixed connection buffer sizes. TAS requires connection send and receive buffers to be fixed upon connection creation. We do not currently implement any buffer resizing depending on load. For workloads with large numbers of inactive connections, buffer resizing (via additional management commands) is desirable.

TCP slow start. Our prototype does not fully implement the TCP slow start algorithm. Instead, we currently double the sending rate every RTT until we reach steady-state. With the exception of short-lived connections, our measurements are concerned with steady-state performance and this limitation does not impact the reported results. For short-lived connections, TAS might be slightly negatively impacted, as RTT estimates are based on and thus may lag behind received TCP acknowledgements.

No IP fragments. Our current prototype does not support fragmented IP packets. We believe this is sufficient, as IP fragmentation does not normally occur in the data center.

5 Evaluation

Our evaluation seeks to answer the following questions:

- How does TAS' throughput, latency, and connection scalability for remote procedure call operation compare to state-of-the-art software solutions in the common case? How in the case of short-lived connections? (§5.1)
- Does our simplified fast-path TCP operation negatively affect performance under packet loss or congestion? (§5.2)
- Do these improvements result in better end-to-end throughput and latency for data center applications? How do these workloads scale with the number of CPU cores? (§5.3, 5.4)
- How does TAS perform at scale? Does the split of labor into slow and fast path affect congestion control fidelity with many connections and various round-trip times to remote machines? (§5.5)
- Is TAS consuming CPU resources proportional to its workload? What is the impact on network throughput and latency when TAS changes its CPU resource use? (§5.6)

To answer these questions we first evaluate RPC performance on a number of systems using microbenchmarks. We then evaluate two data center application workloads: a typical, read-heavy, key-value store application and a real-time analytics framework. Finally, we validate our results at scale with an ns-3 simulation.

Testbed cluster. Our evaluation cluster contains a 24-core (48 hyperthreads) Intel Xeon Platinum 8160 (Skylake) system at 2.1 GHz with 196 GB RAM, 33 MB aggregate cache, and an Intel XL710 40Gb Ethernet adapter. We use this system as *the server*. There are also six 6-core Intel Xeon E5-2430 (Sandy Bridge) systems at 2.2 GHz with 18MB aggregate cache, which we use as clients. These systems have Intel X520 (82599-based) dual-port 10Gb Ethernet adapters with both ports connected to the switch. We run Ubuntu Linux 16.04 (kernel version 4.15) with DCTCP congestion control on all machines. We use an Arista 7050S-64 Ethernet switch, set up for DCTCP-style ECN marking at a threshold of 65 packets. The switch has 10G ports (connected to the clients) and 40G ports (connected to the server).

Baseline. We compare TAS performance to the Linux monolithic, in-kernel TCP stack (using `epoll`), to the mTCP kernel-bypass TCP stack [24], and to the IX protected kernel-bypass TCP stack [9]. Unless stated, our benchmarks do not mix peer systems. mTCP and IX do not provide the standard sockets API, requiring significant application modification [9, 24]. Unless stated, we use the same application binary for TAS and Linux.

Peer compatibility. We confirm that TAS interoperates with existing Linux TCP peers by comparing the aggregate throughput of 100 flows between two hosts among all combinations of Linux and TAS senders and receivers. Table 4 shows the result. Line rate was achieved in all cases.

5.1 Remote Procedure Call (RPC)

RPC is a demanding, but necessary mechanism for many server applications. RPCs are both latency and throughput

Sender		Linux	TAS
Receiver	Linux	9.4Gbps	9.4Gbps
	TAS	9.4Gbps	9.4Gbps

Table 4. Compatibility between Linux and TAS: 100 bulk transfer flows from 1 sending machine to 1 receiving machine running the specified combination of network stacks.

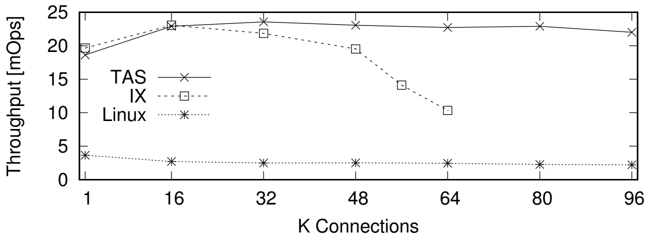


Figure 4. Connection scalability for RPC echo benchmark on 20 core server.

sensitive. Scaling reliable RPCs to many connections has been a long-standing challenge due to the high overhead of software TCP packet processing [30, 39, 41]. To demonstrate the per-core efficiency benefits of TAS, we evaluate a simple event-based RPC echo server.

Connection scalability. For each benchmark run, we establish an increasing number of client connections to the server and measure RPC throughput over 1 minute. To do so, we use multi-threaded clients running on as many client machines as necessary to offer the required load. Each client thread leaves a single 64-byte RPC per connection in flight and waits for a response in a closed loop.

Figure 4 shows throughput as we vary the number of client connections. With 1k connections TAS shows a throughput of 5.1× Linux, and 0.95× IX. The improvement relative to Linux is because TAS streamlines processing and thus gains efficiency. After reaching saturation, throughput for both IX and Linux degrades as the number of connections increases, by 40% for Linux and up to 60% for IX. TAS on the other hand only degrades by up to 7% relative to peak throughput. This is because of TAS's minimal fast-path connection state and streamlined packet processing code allowing the CPU to prefetch state efficiently.

Short-lived connections. Separating packet processing into a common case fast path and a separate slow path reduces packet processing overheads in the common case. However, operations that involve slow path processing do incur additional overheads because of handoff overheads between the slow path and the fast path. The most heavy-weight such operations in TAS are connection setup and teardown, involving not just the slow path but also the application several times during each handshake. To quantify these overheads we measure throughput of 1,024 concurrent, short-lived connections in our RPC echo benchmark. We use one application core, and for TAS two fast-path cores and one partially used

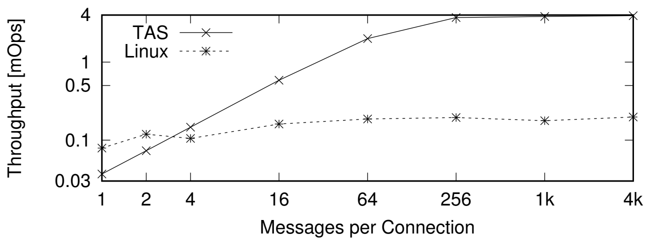


Figure 5. Throughput with short-lived connections.

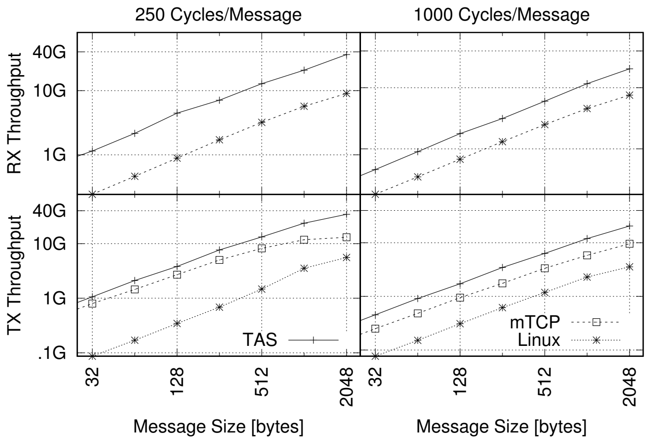


Figure 6. Pipelined RPC throughput, varying per-RPC delay and size, for a single-threaded server.

core for the slow-path. Figure 5 shows the results for varying numbers of RPCs before connections are torn down and re-established with Linux and TAS. With 4 or more RPCs per connection TAS outperforms Linux, and reaches 95% bandwidth utilization with 256 RPCs per connection.

Pipelined RPC. In cases without dependencies, RPCs can be pipelined on a single connection. These transfers can still be limited by TCP stack overheads, depending on RPC size. We compare pipelined RPC throughput for different sizes by running a single-threaded event-based server processing RPCs on 100 connections, partitioned equally over 4 client machines using 4 threads each. After each RPC the server waits for an artificial delay of 250 or 1000 cycles to simulate application processing. To break out improvements in receive and transmission overhead, we run separate benchmarks, one where the server only receives RPCs and one where it only sends.

Figure 6 shows the results. When receiving small ($\leq$ 64B) RPCs, TAS provides up to 4.5× better throughput than Linux. TAS's improvement reduces to 4× as RPCs become larger. TAS reaches 40G line-rate with 2KB RPCs for 250 cycles of processing while Linux barely reaches 10G. For 1000 cycles of processing, no stack achieves line-rate and TAS provides a steady throughput improvement around 2.5× regardless of RPC size. mTCP locks up in this experiment.

When sending small and moderate ($\leq$ 256B) RPCs at 250 cycles processing time, TAS provides up to 12.4× Linux and

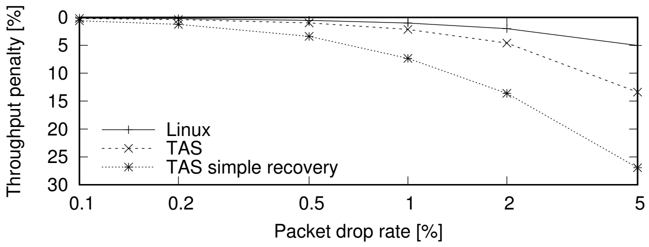


Figure 7. Throughput penalty with varying packet loss rate.

1.5× mTCP efficiency. For large (2KB) RPCs, TAS's advantage declines to 6.1× Linux, but improves to 2.6× mTCP. mTCP reaches scalability limitations beyond 512B RPCs, while Linux catches up as memory copying costs start to dominate. TAS again achieves 40G line-rate at 2KB RPC size, while Linux and mTCP do not reach beyond 10G. This shows that simplifications in common-case send processing, such as removing intermediate send queueing, can make a big difference.

This difference again diminishes as application-level processing grows to 1000 cycles. In this case, TAS provides a steady improvement of up to 5–6× Linux, regardless of RPC size. Compared to mTCP, TAS provides up to 2× improvement. TAS performs comparably to mTCP in both transmit cases, but does provide protection.

We conclude that TAS indeed provides better RPC latency and throughput when compared to both state-of-the-art in-kernel and kernel-bypass TCP stack solutions. Further, TAS provides throughput on par with and better latency than kernel-bypass stacks while retaining traditional OS safety guarantees. Thus we improve performance and efficiency of all networked data center applications relying on RPCs over TCP.

5.2 Packet Loss

Even in a data center environment, minimal (≤1%) packet loss can occur due to congestion and transmission errors. TAS uses a simplified recovery mechanism and we are interested in how packet loss affects TAS throughput in comparison to Linux. We quantify this effect in an experiment measuring throughput of 100 flows over a single link between two machines under different rates of induced packet loss between 0.1% and 5%. We compare TAS with receiver out-of-order processing (cf. Exceptions in Section 3.1) and without it (simple go-back-N).

Figure 7 shows the penalty relative to the throughput achieved without loss. We can see that TAS throughput is minimally affected (up to 1.5%) for loss rates up to 1%. For a loss rate of 5%, TAS incurs a throughput penalty of 13%. Overall, TAS's penalty is about 2× that of Linux. Linux keeps all received out-of-order segments and also issues selective acknowledgements, allowing it to recover more quickly. TAS only keeps one continuous interval of out-of-order data, requiring the sender to resend more in some cases. Without receiver out-of-order processing, the penalty increases by a factor of 3. We conclude that limited out-of-order processing has a benefit, but full out-of-order processing has minimal impact for the loss rates common in data centers.

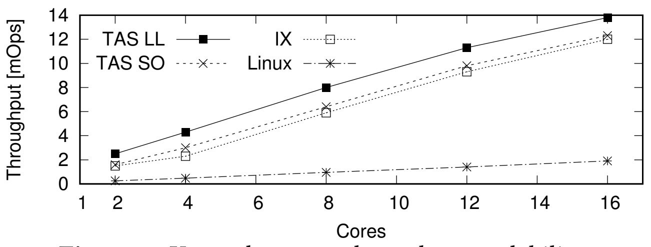


Figure 8. Key-value store throughput scalability.

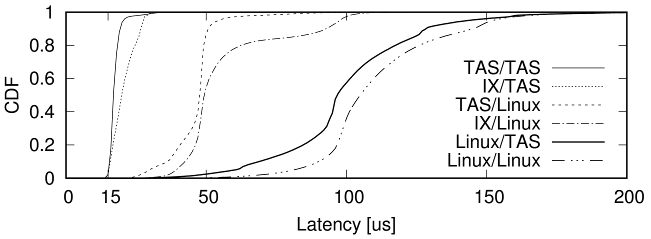


Figure 9. Key-value store latency CDF with different configurations (server stack / client stack).

Latency [μs]	Median	90th	99th	Max
Linux	97	129	177	1319
IX	20	27	30	280
TAS	17	20	30	122

Table 5. Key-value store request latency in microseconds with TAS clients.

5.3 Key-Value Store

Key-value stores strongly rely on RPCs. Due to the high TCP processing overhead, some cloud operators use UDP for reads and use TCP only for writing. In this section, we demonstrate that TAS is fast enough to be used for both reading and writing, simplifying application design. To do so, we evaluate a scalable key-value store, modeled after memcached [4]. We send it requests at a constant rate using a tool similar to the popular memslap benchmark. The workload consists of 100,000 key-value pairs of 32 byte keys and 64 byte values, with a skewed access distribution (zipf, $s = 0.9$). The workload contains 90% GET requests and 10% SET requests. Throughput was measured over 2 minutes after 1 minute of warm-up.

Throughput scalability. To conduct throughput benchmarks we run 5 client machines, each using 12 cores in total to generate requests directed at the server. For all cases we run the clients on TAS to maximize the throughput they can generate. We establish 32k connections with at most one request in flight per connection, while the clients adjust the offered load to maximize throughput without excessive

Total Cores		**2**	**4**	**8**	**12**	**16**
Sockets	App	1	2	5	7	9
	TAS	1	2	3	5	7
Lowlevel	App	1	2	4	6	8
	TAS	1	2	4	6	8

Table 6. Core split for TAS in the key-value store throughput experiment from Figure 8.

queuing in the application receive buffers. We run the benchmark, varying the number of server cores available. Figure 8 shows the result, counting all server cores in use for the application and TCP stack. We also measure throughput for a version of the key-value store that uses the TAS low-level API (TAS LL), skipping the sockets compatibility layer (TAS SO). We can see that TAS LL outperforms Linux and IX in total throughput by up to 9.6× and 1.9×, respectively, and by up to 7.0× and 1.3× with TAS SO. Table 6 shows the split of cores between the key-value store and TAS. TAS SO requires up to 2 fewer cores for TCP processing, which we allocate to the application instead.

Latency. We also conduct end-to-end latency experiments under 15% bandwidth utilization, so that queues do not build excessively. This experiment uses a single application core (and one TAS fast-path core). To quantify both server-side and client-side effects, we repeat the experiment with TAS and Linux on the client side (IX does not support our client). Figure 9 shows the resulting latency distributions and Table 5 summarizes the results. When using TAS clients, we can see that TAS outperforms Linux and IX by a median 5.6× and 15%, respectively. Both IX and TAS demonstrate much better tail behavior than Linux, improving 99th percentile tail latency versus Linux by 5.9×. While 99th percentile latency of TAS and IX is identical, IX has a longer tail than TAS, with a maximum latency 2.3× that of TAS. TAS also maintains lower latencies than IX in the median to 99th percentile range, with a 90th percentile improvement of 26%. We attain similar improvements when using a Linux client.

Non-scalable workloads. We evaluate TAS's performance for workloads with scalability bottlenecks by increasing the access skew to maximize contention on a single 4-byte key and value. Our key-value store uses locks to serialize key updates, causing it to scale badly in this case. This experiment uses the same client setup with 256 connections.

Table 7 shows throughput with varying numbers of aggregate cores used for TAS LL and SO, IX, and Linux. The TAS core numbers use 1 application core with 1-3 fast path cores. TAS scales to 4 cores and IX to 3 cores. In the limit TAS improves throughput by 1.6× relative to IX, and by 5.7× relative to Linux (1.1× and 3.9× with sockets). We conclude that TAS's ability to scale the network stack independently from the application can signficantly improve performance for applications with scalability bottlenecks (e.g., Memcached) or not designed to scale (e.g. Redis [5]).

Throughput [mOps]	1 Core	2 C	3 C	4 C
TAS LL		2.4	3.8	4.6
TAS SO		2.4	3.1	3.1
IX	1.5	2.5	2.8	2.8
Linux	0.3	0.4	0.6	0.8

Table 7. Throughput for non-scalable key-value store workload, with a single 4-byte key and 4-byte value pair.

We conclude that TAS can improve the performance of RPC-based client-server applications, such as key-value stores, even in cases where these applications have scalability bottlenecks. It exceeds state-of-the-art network stacks in both latency and throughput, both in median and the tail. TAS can simplify the design of RPC-based applications by allowing them to rely on the familiar TCP sockets interface.

5.4 Real-time Analytics

Real-time analytics platforms are useful tools to gain instantaneous, dynamic insight into vast datasets that change frequently. These systems must be able to produce answers within a short timespan and process millions of dataset changes per second. To do so, analytics platforms utilize data stream processing techniques: A set of *workers* run continuously on a cluster of machines; data *tuples* containing updates stream through them according to a dataflow processing graph, known as a *topology*. The system scales by replicating workers over multiple cores and spreading incoming tuples over the replicas. To minimize loss, many implementations transmit tuples via the TCP protocol.

To direct tuples to worker cores on each machine, a demultiplexer thread is introduced that receives all incoming tuples and forwards them to the correct executor for processing. Similarly, outgoing tuples are first relayed to a multiplexer thread that batches tuples before sending them onto their destination connections for better performance.

Testbed setup. We evaluate the performance of the FlexStorm real-time analytics platform, obtained from the authors of [25], by running the same benchmark presented in [25]. Figure 10 and Table 8 show average achievable throughput and latency at peak load on this workload. Throughput is measured over a runtime of 20 seconds, shown raw and per core over the entire deployment. Per-tuple latency is broken down into time spent in processing, and in input and output queues, as measured at user-level, within FlexStorm. We deploy FlexStorm on 3 machines of our client cluster. We evenly distribute workers over the machines to balance the load.

Linux performance. Overhead introduced by the Linux kernel network stack limits FlexStorm performance. Even

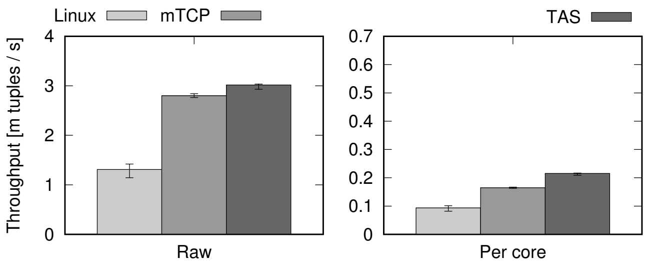


Figure 10. Average throughput on various FlexStorm configurations. Error bars show min/max over 20 runs.

	Input	Processing	Output	**Total**
Linux	6.96 μs	0.37 μs	20 ms	**20 ms**
mTCP	4 ms	0.33 μs	14 ms	**18 ms**
TAS	7.47 μs	0.36 μs	8 ms	**8 ms**

Table 8. Average FlexStorm tuple processing time.

though per-tuple processing time is short, tuples spend several milliseconds in queues after reception and before emission. Queueing before emission is due to batching in the multiplexing thread, which batches up to 10 milliseconds of tuples before emission. Input queueing is minimal in FlexStorm as it is past the bottleneck of the Linux kernel and thus packets are queued at a lower rate than they are removed.

mTCP performance. Running all FlexStorm nodes on mTCP yields a 2.1× raw throughput improvement versus Linux, while utilizing an additional core per node to execute the mTCP user-level network stack. The per-core throughput improvement is thus lower, 1.8×. We could not run mTCP threads on application cores, as mTCP relies on the NIC's symmetric RSS hash to distribute packets to isolated per-thread stacks for scalability. This does not work for asymmetric applications, like FlexStorm, where the sets of receiving and sending threads are disjoint. The bottleneck is now the FlexStorm multiplexer thread. Input queuing delay has increased dramatically, while output queueing delay decreased only slightly. This is primarily because mTCP collects packets into large batches to minimize context switches among threads. Overall, tuple processing latency has decreased only 10% versus Linux due to the much higher amount of batching in mTCP.

TAS performance. Running all FlexStorm nodes on TAS yields an 8% raw throughput improvement versus mTCP and the per-core throughput improvement is 26%. The improvement is only small as the bottleneck remains the multiplexer thread. Overall, tuple processing latency has decreased 56% versus mTCP. This is because TAS does not require any batching to achieve its performance.

While there are limited throughput improvements to using TAS due to application-level bottlenecks, we conclude that tuple processing latency can be improved tremendously compared to approaches that use batching, as fewer tuples are held in queues. This provides the opportunity for tighter real-time processing guarantees under higher workloads using the same equipment.

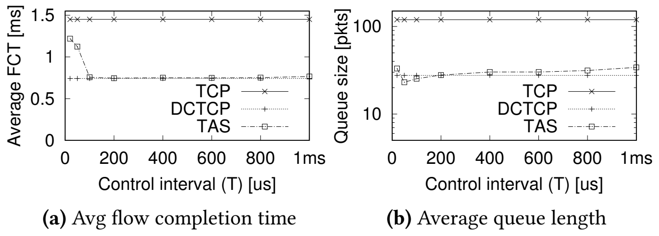


Figure 11. Simulation of a single 10Gbps link.

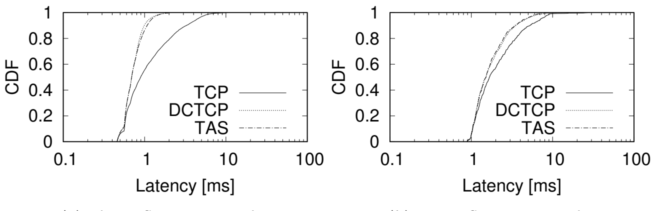
5.5 Congestion Control

We implemented DCTCP congestion control in TAS with the key difference that transmission is rate based, with rates updated periodically for all flows by the kernel at a fixed pre-defined control interval τ. We investigate the impact of τ on congestion behavior via ns-3 simulations, comparing to vanilla DCTCP. First, we simulate a single 10Gbps link with an RTT of 100µs at 75% utilization with Pareto-distributed flow sizes and varying τ. Next, we simulate a large cluster of 2560 servers and a total of 112 switches that are configured in a 3-level FatTree topology with an oversubscription ratio of 1:4. All servers follow and on-off traffic pattern, sending flows to a random server in the data center at a rate such that the core link utilization is approximately 30%. Finally, we investigate congestion fairness experimentally with $\tau = 2 \times \mathrm{RTT}$ (as measured for each flow) under incast.

Single link. Figure 11 shows average flow completion time (FCT) and average queue size with varying τ for the single 10Gbps link. The average FCT for TAS is very similar to that of DCTCP when τ is greater than the RTT. However, if τ is set too low, frequent fluctuations in congestion window cause slow convergence and long completion times. The average queue length is very similar to that of DCTCP and grows, but slowly, as τ increases beyond the RTT, due to delayed congestion window updates.

Large cluster. Figure 12 shows the average flow completion times for short and long flow sizes in the large cluster simulation with the control interval τ set to 100µs. The performance of TAS is similar to that of DCTCP in both cases. 100µs is a reasonable amount of time for the kernel to update congestion windows for thousands of flows. Even with larger values of τ, queue size is only minimally affected and FCTs stay approximately identical. We thus conclude that our out-of-band approach works to provide DCTCP-compatible congestion behavior.

Tail-latency under incast. To evaluate performance under congestion, we measure tail latency under incast with 4 machines sending to a single receiver (operating at line



(a) Short flows ≤ 50 pkts **(b)** Long flows > 50 pkts

Figure 12. Flow completion times for large cluster simulation.

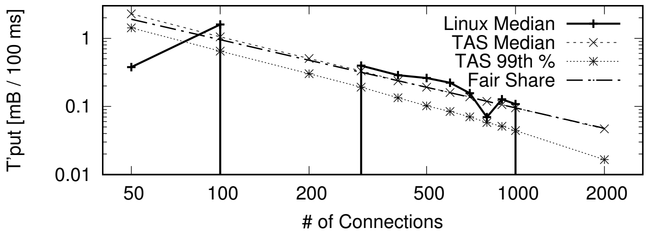


Figure 13. Distribution of connection rates under incast.

rate) with different numbers of connections. We record the number of bytes received on each connection every 100ms on the receiver over the period of a minute, discarding a warmup 20 seconds. Figure 13 shows the median (and 99th percentile) throughput over the measured intervals and connections on Linux (using DCTCP) and TAS. For TAS, the tail falls within 1.6× and 2.8× of the median, while the median is close to each connection's fair share. Linux median (and tail—not shown) behavior fluctuates widely, showing significant starvation of flows in some cases.

Linux fairness is hurt in three interacting ways: (1) Linux window based congestion control creates bursts when windows abruptly widen and contract under congestion. (2) Window-based congestion control limits the control granularity for low-rtt links. (3) The Linux TCP stack architecture requires many shared queues that can overflow when flows are bursty, resulting in dropped packets without regard to fairness. Rate-based packet scheduling and per-flow queueing in TAS smoothes bursts and eliminates unfair packet drops at end hosts.

5.6 Workload Proportionality

Finally, we analyze a dynamic workload to evaluate how TAS adapts to workload changes and how this affects end-to-end performance. For this experiment, we re-use and instrument the key-value store server and vary the number of clients over time. At time 0 we start with one client machine, and add four additional client machines, one every 10 seconds, after an additional 10 seconds we remove the client machines again one by one. Figure 14 shows both the number of fast-path cores for TAS as well as the total server throughput. TAS starts out with just 1 core, and ramps up to 3 cores for the first client, and continues to add additional cores until reaching 9 cores, before incrementally removing cores again as the

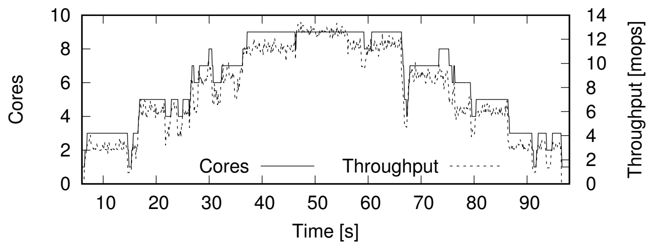


Figure 14. Number of TAS processor cores and end-to-end throughput as key-value store server load first increases and then decreases again.

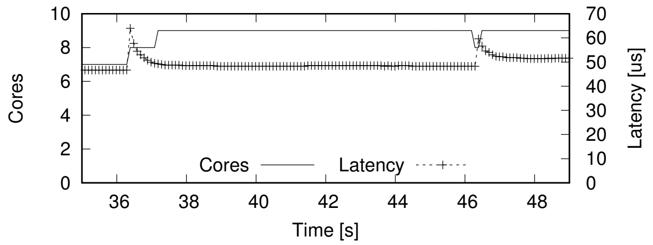


Figure 15. End-to-end request latency as TAS acquires additional processor cores in response to increasing load.

load reduces. Figure 15 shows request latency as measured by clients as for the transition from 3 to 4 clients and with it 7 to 9 cores. During the adjustment the latency temporarily spikes by about 15μs or 30% before quckly returning back to the previous level. We conclude that TAS is able to adapt to workload changes, acquiring and releasing processors as needed, without significantly impacting end-to-end latency or throughput.

6 Discussion

Our TAS prototype streamlines data center software TCP processing. In light of recent interest in using different protocols for data center networking and hardware offload of packet processing, we discuss how TAS fits in the picture. We also discuss how TAS might be used to accelerate TCP processing across the Internet.

Beyond TCP. A number of alternatives to TCP for data center and Internet packet processing have been proposed [18, 20, 38]. While TCP still dominates both domains, it is worthwhile to ask if TAS can support these proposals. Beyond differences in protocol details, such as header format and framing, most high level ideas including the fast path/slow path split for congestion control and timers generalize to other protocols. While adding datagram framing to TAS is simple, it is interesting to note that TCP's byte stream abstraction requires less per-connection state. The fast path only needs to track the stream position and length in the circular buffer (constant size), instead of tracking a variable number of message boundaries.

NIC offload. NIC offload of network packet processing also received renewed recent interest [15, 25]. We believe that offload is a promising long-term solution to accelerating reliable network packet exchange—as long as it is flexible. Data center network infrastructure, protocols, and in particular congestion control are constantly evolving [6, 11, 20, 28, 48]. NIC offload must adapt quickly to these new protocols. We believe TAS' division of labor can inform NIC offload designs. The minimal but resource intensive fast path can be offloaded to the NIC. The complex, but less intensive slow path can remain on host CPUs. This includes the congestion control policy, which can be changed quickly using familiar software programming abstractions.

Internet TCP acceleration. Our focus in this paper is on data center use of TCP. Another benefit of TCP support in TAS is TCP's strong use on the Internet, allowing TAS to be potentially useful to edge applications. Supporting Internet TCP is possible in principle, but requires revisiting some common-case assumptions: Are packet loss rates comparable to the data center scenario? Will IP packet fragmentation need to be handled in the fast path? Will longer round-trip times impact slow path mechanisms, such as congestion control and timeouts? Are connection control events, such as setup and teardown more common and need to be handled on the fast path? Finally, Internet clients will want to send RPCs over secure connections. This entails supporting transport layer security (TLS) features. While doing so via an application-level library is always possible, another potential for fast-path acceleration of TLS within TAS presents itself.

7 Related Work

Software TCP stack improvements. A closely related line of work aims to reduce TCP CPU overhead, often with some level of NIC assistance. Many of these systems also use batching to reduce overhead at some cost in latency; our focus is on reducing overhead for latency-sensitive RPCs where batching is less appropriate. Affinity-accept [33] and Fastsocket [26] use flow steering on the NIC to keep connections local to cores. Arrakis [34] and mTCP [24] use NIC virtualization to move the TCP stack into each application, eliminating kernel calls in the common case, at the cost of trusting the application to implement congestion control. StackMap [47] takes a hybrid approach, using the featureful Linux in-kernel TCP stack, but keeping packet buffers in user-space, eliminating copies between user and kernel space; this provides moderate speedups, but requires the application to be trusted and modified to StackMap's interface. Sandstorm [27] co-designs the TCP stack using application-specific knowledge about packet payloads; this is an interesting avenue for future work. Megapipe [19] re-designs the kernel-application interface around communication channels; we use a similar idea in our design. IX [9] pushes this farther by also changing the socket interface; we aim to keep compatibility with existing applications. To improve load balancing, ZygOS [36] introduces an object steering layer that is similar to ours. Finally, CCP proposes separating congestion control policy from its enforcement [29]; our work can be seen as an implementation of that idea.

NIC-Software co-design. Earlier work on improving packet processing performance used new HW/SW interfaces to reduce the number of required PCIe transitions [10, 16], to scale rate limiting [37], and to enable kernel-bypass [14, 35, 45]. TCP Offload Engines [12, 13] and remote direct memory access (RDMA) [38] go a step further, entirely bypassing the remote CPU for their specific use-case. Scale-out NUMA [32] extends the RDMA approach by integrating a remote memory access controller with the processor cache hierarchy that automatically translates certain CPU memory accesses into remote memory operations. Portals [7] is similar to RDMA, but adds a set of offloadable memory and packet send operations triggered upon matching packet arrival. Out of these approaches, only kernel bypass has found broad market acceptance. One hindrance to widespread adoption of network stack offload is that hardware stack deployment is slower than software stack deployment, while application demands and datacenter network deployments change rapidly. Hardware approaches are thus often not able to keep pace fast enough with the changing world around them. By providing an efficient software network stack, TAS side-steps this issue, while providing performance close to that of hardware solutions.

8 Conclusion

The continuing increase in data center link bandwidth, coupled with a much slower improvement in CPU performance, is threatening the viability of kernel software TCP processing, pushing researchers to investigate alternative solutions. Any alternative has to ensure that it is safe, efficient, scalable, and flexible.

We present TAS, TCP acceleration as a software service. TAS executes common-case TCP operation in an isolated fast path, while handling corner cases in a slow path. TAS achieves throughput up to 7× that of Linux and 1.3× that of IX for common, unmodified cloud applications. Unlike kernel bypass, TAS enforces congestion control on untrusted applications, and achieves much higher levels of per-flow fairness than Linux. TAS scales to many cores and connections, providing up to 2.2× higher throughput than IX on 64K connections, while facilitating TCP protocol innovation.

Acknowledgements

This work is supported in part by NSF grants NSF 1751231 and CNS-1518702, the Texas Systems Research Consortium, Huawei Innovation Research Lab YBN2017120001, as well as gifts from Google, Facebook, Huawei, VMware, Citadel Securities, and ARM. We would like to thank the anonymous reviewers and our shepherd, Robert Soulé, for their comments and feedback.

References

[1] [n. d.]. https://github.com/torvalds/linux/blob/master/net/ipv4/tcp_input.c#L5302.

[2] [n. d.]. https://support.microsoft.com/en-us/help/951037/information-about-the-tcp-chimney-offload-receive-side-scaling-and-net.

[3] [n. d.]. Intel Data Plane Development Kit. http://www.dpdk.org/.

[4] [n. d.]. http://memcached.org/.

[5] [n. d.]. http://redis.io/.

[6] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *2010 ACM Conference on SIGCOMM (SIGCOMM)*. 12. https://doi.org/10.1145/1851182.1851192

[7] Brian W. Barrett, Ron Brightwell, Scott Hemmert, Kevin Pedretti, Kyle Wheeler, Keith Underwood, Rolf Riesen, Arthur B. Maccabee, and Trammell Hudson. 2013. *The Portals 4.0.1 Network Programming Interface* (sand2013-3181 ed.). Sandia National Laboratories.

[8] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *16th ACM Symposium on Operating Systems Principles (SOSP)*. 16. https://doi.org/10.1145/1629575.1629579

[9] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 17. http://dl.acm.org/citation.cfm?id=2685048.2685053

[10] Nathan L. Binkert, Ali G. Saidi, and Steven K. Reinhardt. 2006. Integrated Network Interfaces for High-bandwidth TCP/IP. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/1168857.1168897

[11] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* 14, 5, Article 50 (Oct. 2016), 34 pages. https://doi.org/10.1145/3012426.3022184

[12] Chelsio Communications. 2013. TCP Offload at 40Gbps. http://www.chelsio.com/wp-content/uploads/2013/09/TOE-Technical-Brief.pdf.

[13] Andy Currid. 2004. TCP Offload to the Rescue. *ACM Queue* 2, 3 (June 2004).

[14] Peter Druschel, Larry Peterson, and Bruce Davie. 1994. Experiences with a High-Speed Network Adaptor: A Software Perspective. In *1994 ACM Conference on SIGCOMM (SIGCOMM)*.

[15] 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 *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone

[16] Mario Flajslik and Mendel Rosenblum. 2013. Network Interface Design for Low Latency Request-response Protocols. In *2013 USENIX Annual Technical Conference (ATC)*. 14. http://dl.acm.org/citation.cfm?id=2535461.2535502

[17] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: a scalable and flexible data center network. In *2009 ACM Conference on SIGCOMM (SIGCOMM)*.

[18] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk. 2016. QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2. https://tools.ietf.org/html/draft-tsvwg-quic-protocol-02.

[19] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 14. http://dl.acm.org/citation.cfm?id=2387880.2387894

[20] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. 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)*. ACM, New York, NY, USA, 29–42. https://doi.org/10.1145/3098822.3098825

[21] Van Jacobson. [n. d.]. TCP in 30 instructions. http://www.pdl.cmu.edu/mailinglists/ips/mail/msg00133.html.

[22] V. Jacobson. 1988. Congestion Avoidance and Control. *SIGCOMM Computer Communication Review* 18, 4 (Aug. 1988), 314–329. https://doi.org/10.1145/52325.52356

[23] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. 2013. Speeding Up Distributed Request-response Workflows. In *2013 ACM Conference on SIGCOMM (SIGCOMM)*.

[24] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 14. http://dl.acm.org/citation.cfm?id=2616448.2616493

[25] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 15. https://doi.org/10.1145/2872362.2872367

[26] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. 2016. Scalable Kernel TCP Design and Implementation for Short-Lived Connections. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 14. https://doi.org/10.1145/2872362.2872391

[27] Ilias Marinos, Robert N.M. Watson, and Mark Handley. 2014. Network Stack Specialization for Performance. In *2014 ACM Conference on SIGCOMM (SIGCOMM)*. 12. https://doi.org/10.1145/2619239.2626311

[28] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *2015 ACM Conference on SIGCOMM (SIGCOMM)*. 14. https://doi.org/10.1145/2785956.2787510

[29] Akshay Narayan, Frank J. Cangialosi, Prateesh Goyal, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. 2017. The Case for Moving Congestion Control Out of the Datapath. In *Sixteenth ACM Workshop on Hot Topics in Networks (HotNets)*. Palo Alto, CA.

[30] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 14. http://dl.acm.org/citation.cfm?id=2482626.2482663

[31] Zhixiong Niu, Hong Xu, Dongsu Han, Peng Cheng, Yongqiang Xiong, Guo Chen, and Keith Winstein. 2017. Network Stack As a Service in the Cloud. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets-XVI)*. ACM, New York, NY, USA, 65–71. https://doi.org/10.1145/3152434.3152442

[32] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/2541940.2541965

[33] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. 2012. Improving Network Connection Locality on Multicore

Systems. In *7th ACM European Conference on Computer Systems (EuroSys)*. 14. https://doi.org/10.1145/2168836.2168870

[34] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2015. Arrakis: The Operating System Is the Control Plane. *ACM Transactions on Computer Systems* 33, 4, Article 11 (Nov. 2015), 30 pages. https://doi.org/10.1145/2812806

[35] Ian Pratt and Keir Fraser. 2001. Arsenic: A User-Accessible Gigabit Ethernet Interface. In *20th IEEE International Conference on Computer Communications (INFOCOM)*.

[36] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. 17. https://doi.org/10.1145/3132747.3132780

[37] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. 2014. SENIC: Scalable NIC for End-Host Rate Limiting. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/radhakrishnan

[38] RDMA Consortium. [n. d.]. Architectural specifications for RDMA over TCP/IP. http://www.rdmaconsortium.org/.

[39] Rick Reed. 2012. Scaling to Millions of Simultaneous Connections. http://www.erlang-factory.com/upload/presentations/558/efsf2012-whatsapp-scaling.pdf.

[40] Greg Regnier, Srihari Makineni, Ramesh Illikkal, Ravi Iyer, Dave Minturn, Ram Huggahalli, Don Newell, Linda Cline, and Annie Foong. 2004. TCP Onloading for Data Center Servers. *Computer* 37, 11 (Nov. 2004), 48–58. https://doi.org/10.1109/MC.2004.223

[41] Mihai Rotaru. 2013. Scaling to 12 Million Concurrent Connections: How MigratoryData Did It. https://mrotaru.wordpress.com/2013/10/10/scaling-to-12-million-concurrent-connections-how-migratorydata-did-it/.

[42] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. 1999. TCP Congestion Control with a Misbehaving Receiver. *SIGCOMM Computer Communication Review* 29, 5 (Oct. 1999), 71–78. https://doi.org/10.1145/505696.505704

[43] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *2015 ACM Conference on SIGCOMM (SIGCOMM)*.

[44] Livio Soares and Michael Stumm. 2010. 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)*. USENIX Association, Berkeley, CA, USA, 33–46. http://dl.acm.org/citation.cfm?id=1924943.1924946

[45] T. von Eicken, A. Basu, V. Buch, and W. Vogels. 1995. U-Net: a user-level network interface for parallel and distributed computing. In *15th ACM Symposium on Operating Systems Principles (SOSP)*.

[46] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *ISPASS*. IEEE Computer Society, 35–44.

[47] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-latency Networking with the OS Stack and Dedicated NICs. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16)*. USENIX Association, Berkeley, CA, USA, 43–56. http://dl.acm.org/citation.cfm?id=3026959.3026964

[48] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. In *2015 ACM Conference on SIGCOMM (SIGCOMM)*. 14. https://doi.org/10.1145/2785956.2787484