Presto: A Match-Action TCP Stack for the Terabit Era

Rajath Shashidhara
University of Washington
Seattle, WA, USA
rajaths@cs.washington.edu

Antoine Kaufmann
MPI-SWS
Saarbrücken, Saarland, Germany
antoinek@mpi-sws.org

Simon Peter
University of Washington
Seattle, WA, USA
simpeter@cs.washington.edu

Abstract

We present Presto, the first TCP stack that delivers ASIC-class performance and energy efficiency on programmable Reconfigurable Match-Action Table (RMT) pipelines, providing flexibility while retaining standard TCP semantics and POSIX socket compatibility. The key challenge in designing Presto is reconciling TCP's complex, dependent state updates with RMT's unidirectional, lock-step execution model. To overcome this challenge, Presto introduces three novel techniques: optimistic concurrency (speculative updates validated downstream), pseudo-segment injection (circular dependency resolution without stalls), and bump-in-the-wire processing (single-pass segment handling). Together, these enable TCP retransmission, reassembly, flow, and congestion control, as a pipeline of simple match-action operations.

Our Intel Tofino 2 prototype demonstrates Presto's scalability to terabit speeds, flexibility, and robustness to network dynamics. Presto matches RDMA performance and efficiency for both RPC and streaming workloads (including NVMe-oF with SPDK), while maintaining TCP/POSIX compatibility. Presto saves up to 16 host CPU cores versus state-of-the-art kernel-bypass TCP, while achieving 5× lower 99.99p tail latency and 2× better throughput-per-watt for key-value stores. At scale, Presto drives nearly 1 Bpps at 20 μs RPC tail latency. Unlike fixed-function offloads, Presto supports transport evolution through in-data-path extensions (selective ACKs, congestion control variants, application co-design for shared logs). Finally, Presto generalizes to FPGA SmartNICs, outperforming Tonic's monolithic design by 3× under equal timing.

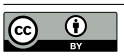
CCS Concepts

• **Networks** → **In-network processing**; **Programmable networks**; **Transport protocols**; **Data center networks**; • **Hardware** → *Networking hardware.*

Keywords

transport protocols, TCP, datacenter networks, programmable networks, RMT, network stack

ACM Reference Format:
Rajath Shashidhara, Antoine Kaufmann, and Simon Peter. 2026. Presto: A Match-Action TCP Stack for the Terabit Era. In *ACM SIGCOMM 2026 Conference (SIGCOMM '26), August 17–21, 2026, Denver, CO, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3789240.3829111



SIGCOMM '26, Denver, CO, USA

ACM ISBN 979-8-4007-2467-1/2026/08
https://doi.org/10.1145/3789240.3829111

1 Introduction

In modern datacenters, the "CPU tax" of networking has become unsustainable. Even with state-of-the-art kernel-bypass TCP stacks, applications spend up to 48% of per-packet CPU cycles in the stack [91]. Yet, software TCP stacks remain the default due to their broad compatibility, flexibility, and robust semantics. As networks scale to terabits and latency targets tighten, operators face an increasingly untenable trade-off among performance, efficiency, and flexibility. CPU-based stacks cannot keep pace, yielding poor bandwidth utilization, high energy consumption, and excessive overhead. At the other extreme, ASIC transports, such as TCP offload engines (TOEs) and remote direct memory access (RDMA), are highly efficient, but inflexible and operationally brittle in the cloud [93–95], and are thus largely confined to high-performance computing and storage deployments [7, 32]. Programmable network processing units (NPUs) [91] and FPGAs [5, 58] only partially bridge this gap and still face limitations in scalability, complexity, and energy efficiency.

We break this tension by leveraging the Reconfigurable Match-Action Table (RMT) architecture [12], widely deployed in programmable network switches [20] and SmartNICs [73]. RMT provides deterministic, energy-efficient line-rate packet processing through a sequence of match-action stages, while retaining a programmable execution model that has enabled a wide range of networking tasks [35, 49, 50, 54, 56, 87, 88, 96, 102]. Realizing TCP in RMT, however, is challenging. TCP requires complex, interdependent state updates, consistent state for bidirectional flow coordination, and buffering for segment reassembly and retransmission, all ill-suited to RMT's strictly unidirectional pipeline with limited local state and tight timing constraints. Bridging this mismatch would unlock ASIC performance and efficiency with software flexibility.

We present Presto, the first TCP stack tailored to the RMT architecture. Presto reimagines TCP as a pipeline of lightweight match-action operations, enabling line-rate processing without buffering and delivering practical TCP at terabit speeds while preserving high throughput, low latency, energy efficiency, flexibility, and interoperability. To reconcile TCP's complex state dependencies and RMT's unidirectional flow, Presto introduces: (1) *optimistic concurrency with deferred validation* to resolve *forward read dependencies*, where earlier stages depend on state available only in later stages, by performing speculative updates validated downstream; (2) *pseudo segment injection*, which eliminates *circular write dependencies* without side-effects or pipeline stalls; and (3) a *bump-in-the-wire* design that avoids recirculation and performs stateful TCP segment processing in a single pipeline pass. Together, these principles enable terabit-scale performance, while maintaining TCP semantics and POSIX socket compatibility.

We make the following contributions:

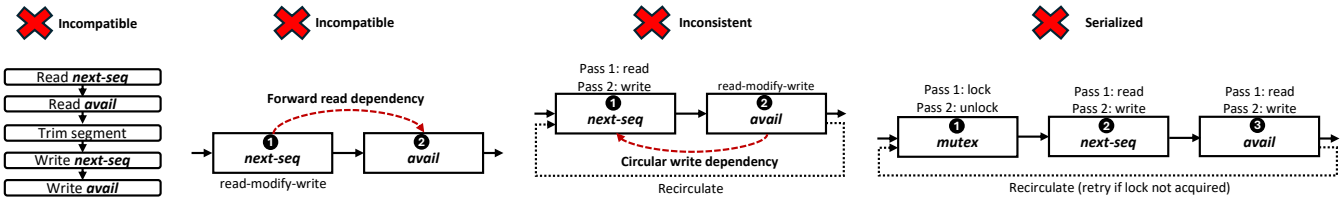


Figure 1: Challenges in mapping TCP reassembly onto an RMT pipeline. (a) CPU: the stack freely reads and updates state multiple times. (b) RMT: placing state in different stages introduces a *forward read dependency*. (c) RMT: Deferring updates to resolve this dependency creates a *circular write dependency*. (d) RMT: Serializing updates stalls the pipeline.

- **A match-action TCP design.** We present Presto's design principles, showing how to map TCP's dependency-heavy data-path onto a unidirectional, match-action pipeline with tight timing and resource constraints. While rooted in TCP, these principles apply broadly across stateful protocols and programmable network platforms, including SmartNICs, FPGAs, and hybrid systems that combine programmable and fixed-function components.
- **Prototype on multiple hardware targets.** We demonstrate that these principles are realizable on real hardware with a Presto design and implementation on an Intel Tofino 2 switch [20, 72], showing for the first time that RMT can support a TCP stack with ASIC-like performance and energy efficiency while retaining programmability. We establish generalizability with our Presto port to an FPGA SmartNIC; under identical timing constraints it supports 3× higher packet rates than Tonic [5] and orders of magnitude higher than Beehive [58], while consuming significantly fewer hardware resources. Our code is available at https://github.com/tcp-acceleration-service/Presto.
- **Performance, robustness, flexibility, and interoperability.** We compare Presto to Linux, TAS [48], and RDMA (§5). Presto surpasses TAS's peak throughput using 16 fewer CPU cores and matches RDMA performance and efficiency for both RPCs and streaming workloads (up to 25Mpps per core, enough to saturate 1.6Tbps with 8K MTU). At scale, Presto drives nearly 1Bpps with RPC tail latency near 20μs and remains resilient to packet loss and congestion (up to 2× higher throughput than TAS under loss), while interoperating with other TCP stacks. Presto benefits real applications: it improves a key-value store's throughput-per-watt by 2× while maintaining 99.99p tail latency below TAS's best case, and enables NVMe-over-TCP to achieve RDMA-level performance and CPU efficiency. We demonstrate flexibility with transport extensions (Timely [67] congestion control, delayed and selective ACKs) and application optimizations, including an in-network sequencer for a shared log, similar to NoPaxos [55].

2 Background

Can we map TCP's stateful, dependency-heavy data path onto a lock-step match-action pipeline at line-rate? To approach this question, we first summarize the RMT programming model that underlies high-speed programmable switches and SmartNICs (§2.1). We then explain why TCP poses significant challenges for this model (§2.2), as TCP's per-segment processing creates forward read dependencies (Figure 1b), circular write dependencies (Figure 1c), and, if we avoid both by serializing, pipeline stalls (Figure 1d).

2.1 Match-Action Architecture

The reconfigurable match-action table (RMT) architecture targets line-rate packet processing. It exploits packet-level parallelism and a fixed-latency pipeline of stages that apply match-action logic to a per-packet header vector (PHV). Modern implementations sustain billions of packets per second with sub-microsecond latency and low ASIC power consumption [12, 72].

Pipeline execution. Packets enter through a MAC/DMA channel and are parsed into a PHV. The PHV then traverses a sequence of stages that execute in lock-step. Within each stage, simple ALU operations update PHV fields, accessing *stage-local* state via tightly-timed memory operations. Because stages are synchronized, packet processing latency is fixed by the number of occupied pipeline stages. Implementations typically split processing into ingress and egress pipelines, with a traffic manager (TM) in between. The TM buffers bursts and schedules packets to match output bandwidth; it can also participate in lossless operation via backpressure (e.g., PFC).

RMT provides limited ways to feed information back into the pipeline: (i) **recirculation** (send a packet for another full pass), (ii) **mirroring** (duplicate PHV and reinject), and (iii) **packet generator** (emit packets on triggers). These mechanisms are useful but must be used sparingly: recirculation doubles latency and consumes pipeline bandwidth; mirroring is cheaper but still consumes scheduling and pipeline resources. Finally, uncommon processing and configuration are handled by a CPU control plane that can directly read and write pipeline state, and exchange packets via a dedicated DMA channel.

2.2 Challenges for TCP on RMT

TCP's reliable in-order semantics couple each segment's processing to shared per-connection state (e.g., window boundaries and out-of-order bookkeeping). On a CPU, the stack can read and update this state multiple times in a single run-to-completion handler (Figure 1a). On an RMT pipeline, however, state is stage-local and the PHV flows only forward, so even simple TCP logic quickly runs into architectural constraints.

Prior work, such as TAS [48] and FlexTOE [91], splits TCP into a fast data-path and a slower control path, but assumes a programmable execution model with run-to-completion processing and flexible memory access per packet (e.g., a CPU or NPU). In contrast, RMT executes a fixed-depth, lock-step match-action pipeline: packets carry a bounded amount of metadata and cannot revisit

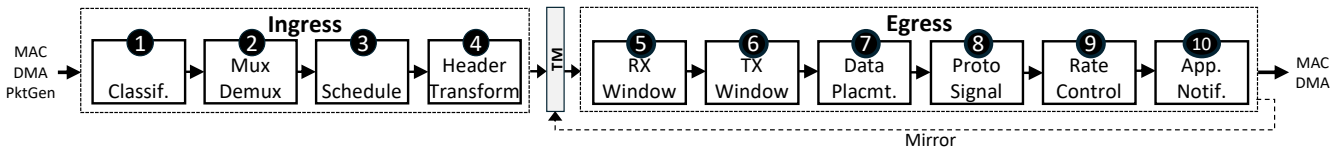


Figure 2: Presto RMT data-path functional blocks.

earlier stages, while per-flow state is accessed only through stage-local stateful operations. As a result, the primary challenge on RMT is not the fast/slow decomposition, but managing TCP's state dependencies without feedback, buffering, or stalls.

In particular, placing related TCP state—or, more generally, any reliable transport (§7)—in different stages forces one of three problematic mappings: (i) if an early stage must update state based on a value stored later, the design introduces a *forward read dependency*, which cannot be resolved (Figure 1b); (ii) if we defer the update until the later value is known, the update must effectively flow backward to earlier-stage state. This creates a *circular write dependency* that requires loopback via recirculation or mirroring (Figure 1c), which consumes bandwidth, adds latency, and can transiently expose stale state; and (iii) serializing updates to avoid both (e.g., freezing state and committing later) stalls the pipeline and squanders parallelism (Figure 1d).

3 Presto Overview

To overcome the challenges of §2.2, Presto rearchitects TCP's per-connection transport logic to maximize RMT pipeline parallelism. We adopt these design principles, each addressing a specific RMT constraint:

1. **Bump-in-the-wire processing (avoid stalls):** Presto processes each TCP segment at most once in the pipeline (no recirculation on the common path) and never buffers segments in the pipeline, preserving both bandwidth and low latency under tight memory constraints.
2. **Optimistic segment processing (handle forward reads):** To deal with forward read dependencies, Presto executes the common case speculatively (in-window segment receipt and delivery) and defers validation to later stages; uncommon cases fall back to corrective actions.
3. **Pseudo segment cyclic updates (resolve circular writes):** To resolve circular write dependencies without stalling the pipeline, Presto injects *pseudo segments* via mirroring. These segments are processed like regular TCP segments, but are crafted to trigger the required updates to earlier-stage state without application-visible side effects.

Like many other proposals [48, 91], Presto accelerates TCP processing for established connections—the common case for high-performance applications—and divides the TCP stack into three key components: (i) A *control plane* orchestrates connection lifecycle and data-path resources, and executes congestion control policies. It may run either on the host or on a control CPU alongside the RMT pipeline. (ii) An *application interface* enables direct interaction with the RMT data-path, exposing both a POSIX-compatible sockets API (libPresto) and a zero-copy API (libPresto-ZC) to applications. libPresto-ZC allows data-intensive applications to access libPresto receive and transmit buffers directly, avoiding unnecessary data copies, while libPresto provides a conventional socket API that copies user buffers into libPresto receive and transmit buffers. (iii) The *RMT data-path* executes core TCP transport logic to ensure reliable delivery of ordered byte streams directly to/from libPresto.

Data-path architecture. For established connections, Presto executes TCP transport logic as a single RMT pipeline triggered by *events* from three sources: the network MAC (incoming TCP segments), the host DMA interface (application payload and notifications), and periodic/explicit triggers (window/credit refresh). These events induce three logical workflows: RX (network → host delivery), TX (host payload → network transmission), and SYNC (connection state update). Although we describe them separately, they all traverse the same match-action pipeline and touch shared per-connection state, reflecting TCP's bidirectional semantics.

To achieve extensibility, Presto structures this pipeline as a sequence of decoupled, independently programmable functional blocks (Figure 2). Each block spans multiple match-action stages and is subject to the state and dependency constraints of the RMT model. The *ingress* portion performs stateless classification and preparation: ❶ Classification identifies whether the event corresponds to RX, TX, or SYNC and selects the corresponding fast path; ❷ Mux/Demux retrieves connection and context identifiers; ❸ Scheduler triggers segment transmissions subject to per-flow rate limits; and ❹ Header Transformation constructs the required protocol and DMA headers before handing the packet/metadata to the traffic manager (TM) for egress processing.

The *egress* portion executes the stateful transport logic for all three workflows. For RX, ❺ Receive Window validates sequence state, detects gaps/loss signals, and records out-of-order segments, while ❻ Transmit Window handles acknowledgment processing and loss detection; ❼ Data Placement translates sequence numbers to host buffer addresses for DMA; and ❿ Application Notification informs the host of newly arrived data. For TX, ❻ Transmit Window enforces flow control, while ❾ Rate Control enforces credit-based transmission rates. For SYNC, the same window and rate control blocks apply credit updates. Finally, ❽ Protocol & Congestion Signaling collects congestion/ACK state and prepares acknowledgments that are emitted without involving the host.

Crucially, Presto restricts *mutable* protocol state updates to the lossless egress pipeline, treating ingress as strictly read-only. This makes Presto *drop resistant*: TM drops are equivalent to network loss and naturally recovered by TCP retransmission, preserving correctness.

We now describe a baseline instantiation of these blocks that implements a complete, standards-compliant TCP stack that integrates directly with real-world datacenter applications running on the

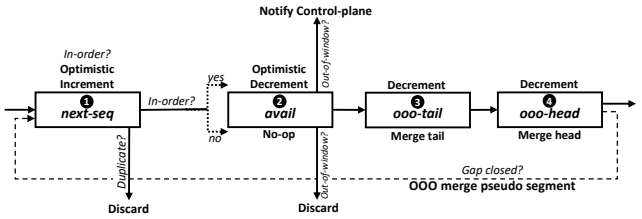


Figure 3: RMT TCP segment reassembly paths in ❺.

host. We focus on segment reassembly (§3.1) and (re-)transmission (§3.2), as they concentrate the hardest RMT-mapping challenges and largely determine correctness and throughput. In §5.3, we evaluate how the same blocks can be reused and extended to support additional transport extensions and application-specific optimizations.

3.1 Segment Reassembly

Segment reassembly involves incorporating any relevant part of an RX segment within the receive window, delivering in-order data to the application, buffering out-of-order (OOO) segments for future delivery, collecting congestion metrics, generating acknowledgments, and notifying libPresto of newly available data.

3.1.1 Receive Window Tracking (Figure 3). Dynamic data structures are infeasible in RMT, so Presto implements reassembly using a *fixed* amount of per-connection state. We define the design's reassembly fidelity by the number of out-of-order (OOO) intervals—contiguous sequence ranges received ahead of the cumulative acknowledgment point—that it can track simultaneously. This yields a spectrum from OOO-0 (drop all OOO segments, no tracking state) to OOO-n (tracking n OOO intervals). Section 5.4 quantifies this fidelity-performance trade-off, while Table 2 shows the resource costs.

Baseline reassembly (OOO-0). OOO-0 maintains only two state variables: `next-seq` (the next expected sequence number) and `avail` (remaining receive-window space). For an in-order segment, the receiver must (i) validate that the segment fits in the advertised window and (ii) advance `next-seq` and decrement `avail` by the accepted payload length. Mapping this directly to an RMT pipeline creates a *forward read dependency*: the stage that advances `next-seq` needs the current `avail` value to validate the window.

Presto resolves this dependency using *optimistic execution*. Well-behaved senders respect the advertised flow-control window, so Presto speculatively assumes the segment is in-window and validates that assumption later. Concretely, in ❶, Presto trims/drops duplicates and *optimistically* advances `next-seq` for in-order segments, forwarding a snapshot of the updated `next-seq` to later stages. In ❷, it then decrements `avail` by the (trimmed) accepted length and performs the definitive out-of-window (OOW) check using the now-updated window state.

If the segment (partially) exceeds the advertised receive window (a sender protocol violation), the speculative update in ❶ has advanced the receive state incorrectly. Presto detects this in ❷, drops the segment, and raises an exception to the control plane, which restores the per-connection state to its previous values. During this brief recovery window, `avail` may become negative due to the speculative decrement. While `avail`<0, subsequent segments

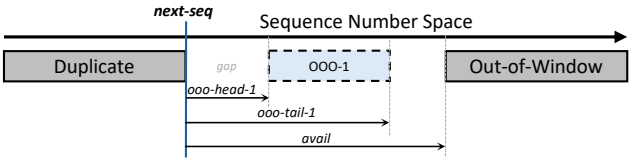


Figure 4: TCP reassembly state (OOO-1) in sequence space.

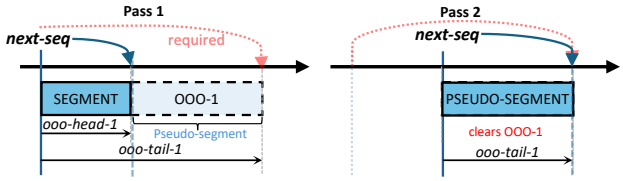


Figure 5: Resolving *circular write dependency* using pseudo-segments (OOO-1).

fail boundary checks and are dropped, and Presto advertises a zero receive window to pause the sender until the restoration completes. Because the receiver accepts payload only after final validation, externally visible behavior stays correct.

Robust reassembly (OOO-1). OOO-1 improves robustness by tracking a single OOO "island" non-contiguous with `next-seq`. It adds two per-connection state variables, `ooo-tail` and `ooo-head`, stored as *offsets relative to* `next-seq` (Figure 4). This representation keeps the OOO interval aligned as the receive window advances. An `ooo-tail`=0 indicates that no OOO interval is currently tracked.

For in-order segments, as `next-seq` advances, Presto decrements `ooo-tail` and `ooo-head` by the same accepted length, preserving the island's absolute position. For out-of-order segments, if the segment overlaps with (or is adjacent to) the tracked interval, Presto merges it by updating the interval boundaries. If no interval is active (`ooo-tail`=0 in ❸), the arriving OOO segment initializes `ooo-tail` and `ooo-head`. Segments that do not overlap the single tracked range are dropped.

A difficult case arises when an in-order segment closes the gap to the tracked OOO interval, making the byte stream contiguous through the end of that interval (Figure 5). At this point the receiver should advance the window to the end of the OOO island and free the corresponding space, i.e., update `next-seq` and `avail`. However, this condition is only known once `ooo-head` is inspected (❹), while `next-seq` and `avail` reside in earlier stages—a *circular write dependency*.

Presto resolves this without stalling the pipeline using a two-step merge. When ❹ detects that the gap is closed (i.e., `ooo-head` reaches 0), it injects a *pseudo-segment*. The pseudo-segment carries no payload but encodes a sequence range spanning the newly contiguous bytes (from the current `next-seq` through `ooo-tail`). When processed, it appears as a standard in-order segment, so the OOO-0 logic advances `next-seq` and decrements `avail` and the OOO interval is cleared.

Intuitively, pseudo-segments allow Presto to express backward state updates using the same forward execution path as real packets, preserving pipeline invariants without introducing special cases or stalls. Pseudo-segments preserve correctness during the transient period before the merge commits. If intervening in-order segments

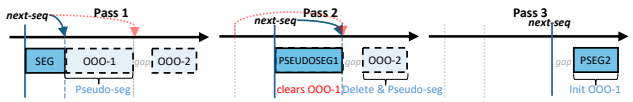


Figure 6: Cascading merge of multiple out-of-order intervals.

advance `next-seq` first, the pseudo-segment's overlapping prefix is naturally trimmed in ❶, preventing double-accounting. A pseudo-segment merge also cannot create an OOW condition: the OOO interval was only formed from segments that were previously validated against the receive window in ❷. Pseudo-segments are best-effort; if one is dropped under congestion, the OOO interval remains tracked and the next triggering segment re-injects it, yielding eventual consistency.

OOO-n. The same approach extends to tracking multiple OOO intervals by adding `ooo-tail-2`, `ooo-head-2`, etc. across subsequent stages. To keep processing simple (and avoid sorting primitives), Presto maintains two invariants: (i) active intervals are stored in increasing sequence order (i.e., `ooo-tail-`$i \geq$ `ooo-tail-`$(i-1)$). (ii) There are no "holes". If `ooo-tail-`i=0 then `ooo-tail-`$(i+1)$=0. With these invariants, merging is a single pass: an arriving OOO segment is tested against each active interval and merged on overlaps. Otherwise, it cascades to the first inactive slot and initializes a new interval.

When an in-order segment closes the gap from `next-seq` to one or more intervals, Presto again uses pseudo-segments to commit the resulting window advance. The exceptional case is when closing OOO-1 would leave a hole before an active OOO-2, violating the second invariant. Presto handles this with a cascading merge: in the triggering pass, it snapshots and clears the subsequent active intervals (Figure 6). To do so, it treats the in-order segment as having an effective length greater than the maximum window size for the purposes of processing subsequent OOO intervals. Since in-order OOO maintenance simply decrements `ooo-tail-i` and `ooo-head-i` by the segment length (as in OOO-1), this deterministically clears all downstream active intervals with no new match-action logic. Presto then injects one pseudo-segment per cleared interval that includes the snapshots. These pseudo-segments are replayed in order, advancing the window or re-initializing intervals without leaving holes.

Replenishment. As the receiver delivers data, it must replenish window space. libPresto triggers the SYNC workflow after the application consumes more than a configurable fraction of its receive buffer (default: $\frac{1}{4}$). On SYNC, the Receive Window block increments `avail` by the amount supplied by libPresto. Because SYNC events traverse the same lossless egress pipeline as RX, replenishment serializes naturally with segment processing, ensuring consistent updates.

3.1.2 Data Placement (❼ in Figure 2). Leveraging our *bump-in-the-wire* design principle, Presto directly places received payload into host memory via DMA, even for OOO segments. Presto maintains a per-connection host receive buffer in libPresto, and the Data Placement block translates TCP sequence numbers into offsets within this buffer to resolve host memory addresses for DMA transfers.

To handle buffer wraparound, Presto double-buffers: the receive buffer is mapped twice consecutively in libPresto's virtual address space, sharing the same physical memory. This lets a single contiguous DMA span the logical end of the buffer rather than splitting into multiple transfers.

3.1.3 Acknowledgment Generation (❽ in Figure 2). Acknowledgment generation in Presto occurs after segment reassembly. The Protocol & Congestion Signaling block governs acknowledgment generation based on post-reassembly window snapshots. When an ACK is required, Presto generates a pseudo-segment encapsulating the reassembly state; this segment is mirrored and transformed into a valid TCP acknowledgment. If the pipeline drops the pseudo segment, the effect is identical to network packet loss of a TCP acknowledgment. Mirroring acknowledgments does not increase pipeline workload, as it replaces host-generated ACK packets rather than introducing additional traffic. As the TX workflow also traverses the reassembly path, it can generate piggybacked acknowledgments on outgoing segments based on receive state. Presto conserves pipeline bandwidth by coalescing pseudo-segments; for instance, the OOO-1 merge operation (§3.1.1) piggybacks on the corresponding ACK pseudo-segment.

3.1.4 Application Notification (❿ in Figure 2). The Application Notification block informs libPresto when newly received data becomes available for application consumption, by including a notification inline with the DMA data transfer. For each *in-order* segment that advances the receive window, Presto notifies libPresto of the highest contiguous buffer offset that is ready to be consumed by the application.

When a received segment closes the gap to an existing out-of-order interval (§3.1.1), Presto issues the notification immediately, ahead of the subsequent pseudo-segment merge, to minimize notification latency.

3.2 (Re-)transmission

(Re-)transmission in Presto regulates the sending rate, selects data for transmission, and recovers from packet loss, driven by data-path transport state.

Presto adopts a *push-based* transmission model optimized for low latency on uncongested flows, which dominate datacenter workloads [46]. The data-path grants transmission credits to libPresto via SYNC messages; upon receiving credits, libPresto proactively pushes eligible payloads into the data-path using DMA writes, allowing immediate transmission aligning with the *bump-in-the-wire* design principle. By contrast, *pull-based* scheduling adds a roundtrip interaction between libPresto and the data-path, raising latency. The same design supports a pull-based variant by transforming SYNCs into DMA reads; we leave this to future work.

3.2.1 Congestion Control. To remain robust to packet loss and congestion, Presto supports DCTCP [2] by default. Like other TCP stacks [48, 71, 91, 94, 106], Presto locates the congestion control policy in the control plane for extensibility. The data-path collects per-connection metrics (acknowledged bytes, ECN-marked bytes, duplicate ACKs) in the Protocol & Congestion Signaling block; the control plane periodically collects these metrics and updates data-path tables with computed transmission rates.

Scheduling via SYNC. Based on the configured rate, the data-path issues periodic SYNC messages that convey transmission credits to libPresto. SYNCs are generated directly in the RMT pipeline using the packet generator, without additional control-plane involvement. The Scheduler block (❸ in Figure 2) assigns SYNCs to flows according to the chosen scheduling policy and rate, controlling both SYNC frequency and the credits granted per SYNC. SYNCs also signal retransmission timeouts; in this case, libPresto resets its transmission head and initiates retransmissions.

To be work conserving, Presto pauses SYNC workflows for inactive connections (no data transmission for N RTTs). For uncongested flows with short messages that underutilize credits, the control plane reduces SYNC frequency. libPresto may also self-pace SYNCs for such flows, triggering them only as the transmit window nears exhaustion.

3.2.2 Transmission. libPresto pushes new data from the host transmit buffer into the data-path via DMA when sufficient credits are available. TCP sequence numbers are derived from DMA offset within the transmit buffer. The Transmit Window block tracks transmit state using logic analogous to the receive side (§3.1.1): segments are validated against the current transmit window. Segments below the cumulative acknowledgment point are dropped and segments extending beyond the window advance it accordingly (❸). Transmission is credit-limited and flow-controlled in the Rate Control block (❾), ensuring the sending rate respects both allocated credits and the peer's advertised receive window.

3.2.3 Acknowledgment Processing. For received ACKs, the Transmit Window block (❻ in Figure 2) updates transmit window state, including the cumulative acknowledgment point, merging any SACK blocks to track lost segments, mirroring the logic of the receive side (§3.1.1). When sufficient transmit window space becomes available, the Application Notification block (❿) transforms the ACK into a notification to libPresto to reclaim buffer space.

Fast Retransmission. Upon three duplicate ACKs, Presto triggers fast retransmission and notifies libPresto of the updated transmit window state (including SACK blocks) via DMA. The Rate Control block simultaneously reduces transmission credits, similar to TCP Reno, and conveys the updated credits to libPresto in the same DMA notification.

4 Implementation

Presto's data-path is implemented on an Intel Tofino 2-based switch, a commercially available RMT pipeline architecture. The system comprises 20,775 lines of code (LoC) across the data-path, control-plane, and libPresto. The P4 code for the offloaded data-path has 2,597 LoC. The control plane is split between the host and a switch-resident CPU (`SwitchConf`), which configures the match-action tables, executes congestion control policy, and processes control notifications. `SwitchConf` interacts with the data-path via Intel's proprietary bfrt C++ API. They are implemented in 3,352 and 6,917 C++ LoC respectively. libPresto is implemented in 7,909 C++ LoC. For DMA, libPresto utilizes RDMA (`libibverbs`) without kernel intervention. The host control-plane and POSIX emulation in libPresto are adapted from TAS. Our prototype leverages some RDMA and Tofino 2 features for optimizations, described below.

DMA. libPresto interacts with the data-path via RDMA unreliable connection (UC) queue pairs, one per application context (thread), akin to per-core NIC queues, with each context managing multiple connections. Data transfers use RDMA Writes (with immediate for notifications) directly to/from per-connection payload buffers. libPresto allocates per-connection payload buffers and registers them with the RDMA NIC, storing the buffer's virtual address and remote memory key in the data-path. Based on available transmission credits, libPresto can issue RDMA writes larger than the MTU, relying on the NIC's segmentation offload to dynamically packetize the data, and paces transmissions over the SYNC interval to smooth bursts. Note that we use RDMA solely for DMA—reliability, ordering, and congestion control are all handled by the data-path. For flow control, PFC is enabled between the host and the switch. External-facing switch ports connected to hosts outside the subnet using TCP do not require PFC. For Presto implementations on RMT-based SmartNICs with PCIe-based DMA interfaces, PFC is unnecessary for lossless operation.

Interrupt-driven operation. By default, libPresto polls the completion queue for events. After 10ms of inactivity, it switches to interrupt-mode by via RDMA completion event channels, letting application sleep while waiting for IO. Periodic SYNC notifications from the data-path are also paused for inactive connections. This reduces the excessive CPU overhead and energy consumption due to polling.

4.1 Testbed Constraints

Below are some constraints of our testbed, which are specific to the Tofino 2 architecture and can be elided in non-switch platforms.

Presto-Presto connections. Same-switch Presto-Presto connections require recirculation because both server and client stacks reside on one switch; non-switch platforms avoid this limitation. All TCP traffic for such connections, segments and ACKs alike, must recirculate to the data-path. For example, an RPC request leaves libPresto as an RDMA write, the switch data-path converts it to a TCP segment, recirculates it for peer reassembly, and converts it back to an RDMA write. The RPC response and ACKs also undergo recirculation. Recirculation adds latency ($\approx 750ns$) and is limited by 3×100G recirculation port bandwidth, with connections sharded across the three ports.

Single-pipeline operation. Although the Tofino 2 features multiple pipelines, our prototype confines data-path processing to a single pipeline. Because we maintain all mutable state within the egress pipeline to ensure drop resistance (§3), the sender and receiver must share the same pipeline, precluding cross-pipeline connections. The remaining pipelines can operate independently as isolated subnets.

Checksums. Our prototype omits payload checksums (including TCP and RDMA iCRC), as the Tofino 2 checksum engine lacks support; a production version would include hardware checksum units. Because the RDMA iCRC is appended after the payload, our prototype requires the sender to add Ethernet padding for the iCRC in TCP segments.

5 Evaluation

We evaluate Presto by addressing the following questions:

- **Performance:** Can Presto deliver predictable μs latency and high message rate for RPCs (§5.1), sustain terabit throughput with near-zero CPU utilization (§5.2), scale efficiently to thousands of applications and connections (§5.1), and benefit real-world storage (§5.2.2) and key-value stores (§5.1.3)?
- **Efficiency:** Does Presto improve performance-per-watt over host-based stacks (§5.1.3)?
- **Flexibility:** Can Presto customize transport logic for critical datacenter applications (§5.3)?
- **Robustness:** Is Presto resilient to loss and congestion, and does it ensure performance isolation between latency-sensitive and bandwidth-intensive workloads (§5.4)?
- **Generalizability:** Do Presto's pipelined transport logic principles extend to FPGA-based SmartNICs (§5.5)?

Testbed. Our evaluation cluster comprises eight Intel Xeon Gold 6430 machines, each with 32-core (64 hyperthreads) running at 2GHz, 256GB RAM, and 126.5MB aggregate cache. Each machine is equipped with a RDMA-capable 200G Mellanox ConnectX-6 NIC, all connected to an 32×400G Intel Tofino-2 based Netberg Aurora 810 switch. We configure the NICs in 100G mode because the switch fails to negotiate with the NICs at 200G speeds. Two machines are designated as servers, with Priority Flow Control (PFC) enabled for lossless RDMA, while the remaining machines function as clients. To ensure high performance and low latency, we disable turbo boost and set frequency scaling to performance mode. Power measurements are obtained via the IPMI interface of both the machines and the switch. `SwitchConf` runs on a 4-core (8 hyperthreads) onboard Intel Xeon D-2123IT CPU, and its power consumption is included in the switch power measurements.

Baselines. We compare Presto performance against the Linux TCP stack, the TAS kernel-bypass TCP stack, and RDMA. Presto interoperates with both TCP baselines (cf. §4.1). Unless otherwise specified, clients use the TAS network stack to generate network traffic. TAS struggles with large flows due to high payload copy overheads, failing to saturate the 100G line rate with a single connection. To mitigate this bottleneck in such client workloads, we elide payload copying in TAS and transmit junk data instead (TAS-nocopy). Across all TCP-based baselines, we use identical application binaries built on a sockets-based `epoll()` interface; for the RDMA baseline, we adapt the benchmarks to use the `libibverbs` interface (a separate verbs-based code path), whereas Presto uses the same sockets-based applications via its POSIX interposition layer. We confirm real-world application compatibility through FlexKVS (§5.1.3) and SPDK NVMe-oF (§5.2.2); more broadly, our POSIX emulation layer is `epoll()` and `libevent` compatible, and runs unmodified datacenter applications, including *memcached* [66], *nginx* [101], and NVMe-oF clients (cf. [91, 99], §5.2.2). Unless otherwise noted, our evaluation configures Presto with OOO-1 reassembly and a simple OOO-0 (go-back-N) sender recovery as a baseline. We also implemented OOO-1 SACK, which enables selective recovery using the same OOO interval abstraction, but compilation issues in the Intel Tofino 2 toolchain prevent hardware evaluation. We therefore evaluate OOO-1 SACK in simulation (Section 5.4). Importantly, go-back-N is not fundamental to Presto's pipeline architecture: it is a baseline choice that offers the best performance-complexity trade-off, and Presto supports higher-fidelity reassembly and selective recovery mechanisms (cf. §5.4, Table 2). We use DCTCP as the default congestion control policy.

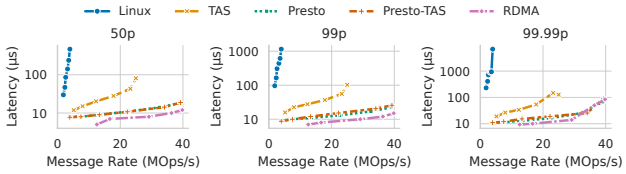


Figure 7: Load-latency across network stacks.

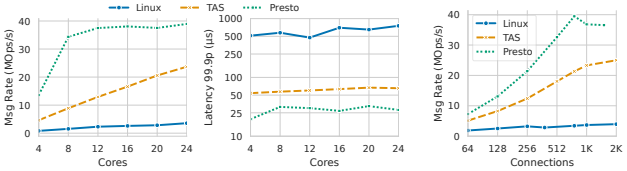


Figure 8: Core and connection scalability.

5.1 Remote Procedure Calls (RPCs)

RPC workloads demand low latency, high packet rates, and scalability across many applications and connections, requirements that heavily tax host TCP stacks. Presto's bump-in-the-wire design efficiently delivers constant, low TCP processing latency even at multi-million segment rates, fully eliminating all CPU overhead. We demonstrate the benefits with an echo server benchmark.

5.1.1 Load-latency. Figure 7 presents the round-trip latency profile of a 32-core echo server across stacks, as load increases. The benchmark scales connections across multiple clients, each leaving a single 64B RPC in-flight. Presto's performance closely matches RDMA, sustaining nearly 40 MOps/sec (1.67× TAS) with an 18μs/24μs median/99.99p tail latency. Recirculation in Presto-Presto connections adds ~ 2μs relative to RDMA (§4). Our setup bypasses egress processing for non-Presto packets, reducing baseline RTTs; a conventional switch would require both ingress and egress. Both Presto and RDMA experience 99.99p tail latency spikes as they approach the NIC's processing limits. A hybrid setup (Presto server with TAS clients) achieves a similar peak throughput but with slightly higher latency due to client-side TAS overheads. Linux is especially inefficient, delivering 10× lower throughput and 78× higher latency than Presto. Presto's sockets interface adds a copy between libPresto and user buffers; libPresto-ZC eliminates it, reaching RDMA parity.

5.1.2 Scalability. We evaluate scalability along cores, connections, and pipeline capacity. We repeat the previous experiment sweeping across host, core, and connection counts, finding the throughput-latency knee for each configuration.

Cores. Figure 8 shows that Presto scales linearly with server cores. Data-path performance is independent of core count, as Presto multiplexes connections onto per-core DMA channels (*contexts*), steering payload directly to application cores (like aRFS [23]) and avoiding RDMA scalability limits. Performance peaks at 8 cores,

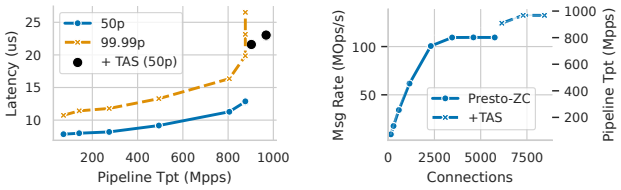


Figure 9: Pipeline packet processing scalability.

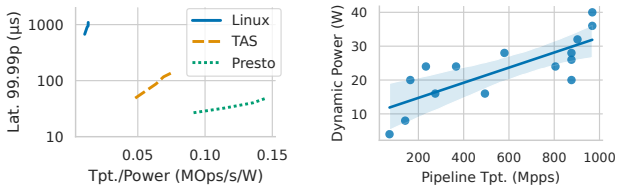


Figure 10: (a) Throughput-per-watt and tail latency for FlexKVS key-value store. (b) Pipeline dynamic power consumption with utilization.

where Presto hits the NIC line rate, surpassing TAS's peak throughput while keeping tail latency 4.3× lower. Linux remains far behind.

Connections. Figure 8 showcases Presto's superior connection scalability for a single host. Presto saturates NIC bandwidth at 768 connections, achieving 1.8× TAS and 10× Linux throughput. Beyond NIC line rate, additional connections slightly reduce throughput. Accessing connection state via stage-local memory at constant latency, Presto sustains 32K connections without performance loss. Sharding TCP state tables across stages or multiple pipelines supports higher connection counts (§5.3). Other high-performance stacks degrade with many connections due to cache pressure [46] or limited on-NIC memory [91].

Pipeline. Each RMT pipeline in our testbed is capable of processing 1.2 Bpps across 8 × 400G links. Using 8 hosts running Presto, exchanging 8B echo RPCs, stresses the pipeline. Each RPC generates 8 packets through the egress pipeline—4 for the request (TX, RX, ACK generation, ACK processing) and 4 for the response. Figure 9 highlights pipeline scalability. Latency stays flat and narrow up to 850 Mpps (71% utilization), after which recirculation bandwidth saturates and tail latency rises. Two more TAS clients (black dots) push throughput near 1 Bpps (80% utilization), with aggregate pipeline throughput nearing 500 Gbps (130 MOps/sec goodput) even at 8B payloads. Latency is higher in this configuration due to client-side TAS overheads. We have tested scalability up to 8K concurrent connections and 256 contexts in this benchmark. Notably, Presto does not experience ACK pseudo-segment drops, even under such extreme loads; the mirroring engine reliably sustains the required packet rate (≈ 250 Mpps). Because OOO pseudo-segments are piggybacked on ACKs (§3.1.3), Presto ensures robust OOO reassembly and recovery without imposing additional load on the pipeline.

5.1.3 Energy Efficiency. We measure the power consumption of the TCP stacks using FlexKVS [47], a scalable, high-performance key-value store inspired by memcached. Short RPCs dominate these workloads, with host TCP stacks consuming up to 48% of per-request CPU cycles [91]. Our benchmark uses 1M uniformly distributed key-value pairs (8B keys, 64B values, 0.99 GET:SET ratio). Baseline power is first recorded for the server (including NIC) and switch. The workload then runs for 5 minutes while measuring total power and application performance. We vary server threads and connections, with each connection limited to a single in-flight request.

Figure 10a shows throughput-per-watt and tail latency across server configurations (# threads, # connections). Presto consistently beats host stacks: even its worst case exceeds TAS's best by 1.2× in throughput-per-watt at 5× lower 99.99p latency; its best configuration doubles TAS's throughput-per-watt and beats TAS's best tail latency. Linux fares far worse, with at least an order of magnitude lower efficiency and two orders higher latency. Switch power varies by no more than 4W between baselines and Presto, with efficiency driven primarily by reduced host CPU usage. RMT-based switches consume power comparable to non-programmable commodity switches of similar bandwidth [1, 12], in contrast to power-hungry FPGAs [3].

Finally, Figure 10b breaks down dynamic power (increase from idle, measured over 10 minutes) as a function of pipeline throughput, showing Presto processes TCP at ≈ 25 Mpps/W for 8B payload packets. These switch measurements include the on-board CPU, RMT pipeline, transceivers, and other fixed-function components (cf. Table 1).

5.2 Byte Streaming

Streaming bandwidth is critical for modern storage and AI workloads. CPU-based stacks like Linux and TAS fail to saturate line rate on a single core, even with hardware-assisted payload copy optimizations [14, 91]. Specialized DMA copy engines [95] do not resolve this, as single-core protocol processing is the bottleneck (§6). Consequently, hardware transports like RDMA and Chelsio TOE, despite their inflexibility, are increasingly preferred for streaming [7, 17, 33, 78]. Can Presto match their performance?

5.2.1 Single-core Throughput. In this benchmark, a client running TAS-nocopy transmits packets in open-loop 1MB streaming bursts (similar to iperf [26]) to a single server core that discards incoming payloads. To provide different intensities of protocol to payload processing, the benchmark varies the segment size by adjusting the maximum transmission unit (MTU). To avoid payload copying in the sockets layer, we use the libPresto-ZC interface, comparing against the equivalent low-level TAS interface (TAS-ll), though TAS still requires copying the payload between network buffers and application memory.

Figure 11 shows that Presto matches RDMA's single-core packet rate of ~25 Mpps, independent of packet size, reaching line rate for higher MTUs, sufficient to exceed 1.6 Tbps (= $25M \times 8 \times 8K = 1600G$) with 8K MTU. The bottleneck is NIC and CPU processing [44], not the RMT pipeline, whose capacity is far greater (cf. Figure 9). Even a single connection can exploit the pipeline's full parallelism and bandwidth. TAS is limited to ~5 Mpps per core—a bottleneck shared by NPU-based stacks [91], dropping further above 512 B due to copy overhead. Linux, again, performs poorly.

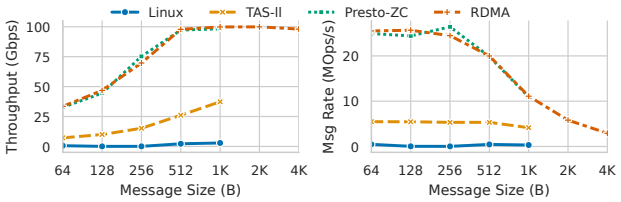


Figure 11: Streaming performance across segment sizes.

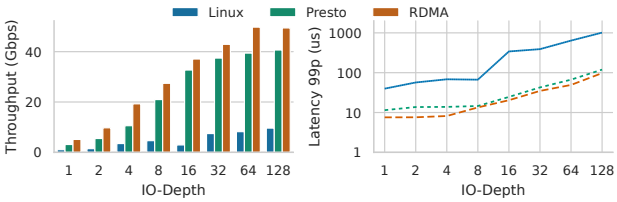


Figure 12: SPDK NVMe-oF performance: 4K Random Writes.

5.2.2 NVMe-over-TCP. Modern datacenters are rapidly adopting large-scale disaggregated storage. We integrate SPDK [31], an open-source user-space storage stack, with Presto to show that NVMe-oTCP can rival RDMA in both performance and efficiency, while retaining a TCP/POSIX programming model.

We extend SPDK with a new transport that interfaces with Presto via libPresto-ZC, delivering NVMe commands and payloads into per-connection buffers directly accessible by SPDK, avoiding intermediate copies and enabling high throughput. A key challenge in supporting NVMe-oTCP is that NVMe devices exploit parallelism by allowing commands to complete out-of-order (OOO). To support this, we provide a modified version of libPresto-ZC that exposes explicit mechanisms to free connection buffer space out-of-order while tracking the buffer head. NVMe-oRDMA provides similar functionality by requiring SPDK to post multiple receive buffers to the RDMA work queue, which are freed independently upon command completion.

We evaluate our NVMe-oTCP storage target using 4KB random writes. In our setup with a single NVMe SSD, Presto can easily exceed the 1M IOPS disk bandwidth, even when only partially utilizing a server core. To shift the bottleneck from the device to the CPU, we configure a RAM block device [97], which emulates Non-Volatile Memory (NVM) or Compute eXpress Link (CXL)-attached memory.

Figure 12 compares the performance of a single-core SPDK NVMe-oF target using the Linux kernel stack, the Presto transport, and RDMA, as IO-depth increases. We use the SPDK perf [98] tool on a remote client node[1] that drives the load through Presto's POSIX sockets interposition layer. We use an MTU of 4,400 with Linux, 4,096 with RDMA and Presto.

Presto closely matches RDMA in throughput and tail latency until the server core saturates at high IO depths from storage-stack processing and RAM block device copies. Presto's peak throughput is slightly lower from periodic SYNC grant overhead, and its latency slightly higher due to client-side sockets layer copies. Linux, by contrast, incurs extra copies and higher stack overhead, leaving fewer application cycles and $> 4\times$ lower throughput.

Takeaway. Matching RDMA in this end-to-end storage setting shows that TCP semantics need not imply CPU-bound transport processing: Presto achieves RDMA-class efficiency while preserving TCP/POSIX compatibility and leaving room for transport evolution in the data-path.

[1]SPDK perf can generate more load compared to fio [98].

5.3 Flexibility

Unlike fixed-function hardware transports, Presto provides programmable transport functionality within the constraints of RMT pipelines, enabling customization both within and beyond standard TCP features, stack design, and application integration. We evaluate flexibility systematically: we first quantify baseline resource usage and then measure the incremental cost of extensions along key design dimensions.

Resource footprint. We begin by quantifying the resource footprint of our baseline design. As shown in Table 1, it consumes a fraction of available pipeline resources while supporting 32K connections and 1K contexts. Presto's data-path sustains 3.2 Tbps (8×400G) line-rate or 1.2 Bpps at a total processing latency (excluding queuing delays) of 395ns and a power consumption of 2.81W for the RMT pipeline (excluding SerDes, MAC, TM). With basic L2/L3 forwarding implemented, the low footprint leaves ample headroom for new functions. For manageability, Presto exports telemetry and debugging metrics for internal transport state via P4 counters—OOO segments, retransmissions, congestion events, and pseudo-segments.

Coverage of the design surface. Transport evolution is broad, but most changes fall into a small number of axes: (i) transport semantics (e.g., ACK and reassembly strategy, loss recovery hints), (ii) congestion control hooks (which metrics are measured, and how they drive pacing/scheduling), and (iii) application integration (what events are exposed and how data is placed/copied). Presto's modular blocks expose these axes directly, so extensions typically require localized changes rather than redesigning the pipeline. Conversely, Presto does not aim to support unbounded per-flow data structures or complex control-flow in the data-path; such functionality is delegated to the control plane or endpoints.

5.3.1 Case Studies. Presto's architecture is modular (Figure 2), with each block exposing a distinct dimension of programmability. These dimensions include: transport semantics (reassembly and ACK strategies), congestion control (policy, metrics, and scheduling), and application integration (notifications, APIs, and data placement). We next illustrate how these flexibility dimensions can be exercised in practice through multiple extensions and reassembly strategies, each time incurring a modest resource cost (Table 2).

Reassembly fidelity. Our reassembly fidelity spectrum (§3.1.1) enables operators to trade-off resource usage for robustness by tuning the Receive Window block. As shown in Table 2, OOO-n reassembly variants fit within Tofino 2 constraints; each increment in fidelity requires two additional stages and results in a modest power increase.

Resource	Usage
Stages	14/20
Processing Latency	395 ns
MAU Power (*worst case* per-pipe)	2.8 W
Memory	
SRAM	19.8 %
Map RAM	14.5 %
TCAM	0.4 %
Compute	
VLIW Instruction	9.5 %
Stateful ALU	26.3 %
Match Crossbar	10.8 %
Gateway	26.6 %

Table 1: Tofino 2 RMT pipeline resource usage for Presto.

Extension	Blocks (Fig. 2)	P4 LoC	Power (W)
Reassembly Fidelity			
OOO-0	❺	176	0.19
OOO-1	❺	257	0.35
OOO-2	❺	343	0.41
Transport Ext.			
Delayed ACKs	❽, Control Plane	42	0.05
1-OOO SACK	❽, ❹	152	0.12
Congestion Control			
Timely	❽, Control Plane	64	0.37
Application Co-design			
Shared log	❼, ❿, libPresto	156	0.29

Table 2: Marginal cost of Presto extensions.

Transport extensions. We modify the Protocol & Congestion Signaling block to support alternative acknowledgment strategies, including delayed and selective ACK variants.

Delayed or cumulative ACKs [13] acknowledge every M consecutively received segments, reducing sender and network load [2]. For instance, for RPCs, piggybacking the ACK onto the response can halve sender overhead. In Presto, a counter tracks unacknowledged segments, issuing an ACK pseudo-segment at threshold M (or immediately for OOO segments); transmitted segments reset it. The control plane manages the delayed ACK timer. Benchmarking with a Presto echo-server and TAS client (single fast-path core) shows delayed ACKs improve peak throughput by 24% by lowering client CPU load.

Selective ACKs [30] enable targeted retransmissions of lost segments, accelerating recovery in high-loss scenarios. We implement 1-OOO selective ACKs by extending the ACK generation logic to encode the 1-OOO interval, as discussed in §3.1.1, and verify that the design fits within Tofino 2 constraints (cf. §5.4).

Congestion control. Congestion control remains an evolving research area with numerous proposals addressing diverse network conditions and performance objectives [93]. Timely [67], which infers congestion from round-trip time (RTT) variations, is a notable example. Recent RMT hardware enhancements [20, 104], including support for lightweight arithmetic and look-up table approximations, enable more sophisticated computations within the data-path. As a concrete example, we implement Timely by leveraging Tofino 2's hardware timestamping to add TCP timestamps [11] and compute per-flow RTT exponentially weighted moving averages (EWMAs) in the data-path using its low-pass filter unit [36]. The control plane periodically adjusts transmission rates based on these RTTs, while the data-path enforces the rate limits.

Application co-design. Databases [8], storage clusters [9], and replicated state machines [21] rely on shared logs to linearize state updates [9, 62]. Inspired by NoPaxos [55], we accelerate a shared log microbenchmark with a tailored Presto API: the application registers its log buffer and client connections, and Presto sequences incoming records across connections, appending them directly to the log via a buffer tail pointer within the data-path [2]. Our design guarantees that records from the same connection are appended

[2]Clients ensure log records are not fragmented across TCP segments.

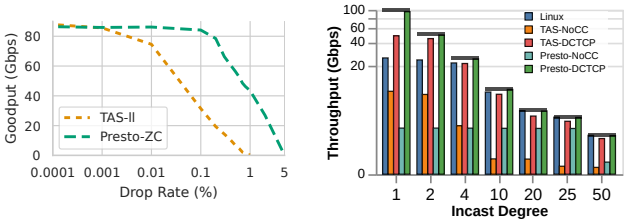


Figure 13: Streaming goodput: (a) random drops, (b) incast.

to the shared log in order, resulting in a linearized shared log. If OOO segments arrive, Presto reserves space up to the highest sequence number, creating gaps that the application later fills from the corresponding segments Presto redirects to it.

We modify the Data Placement, Application Notification, and the libPresto API layers to accelerate a shared log microbenchmark in this way. With up to 32 clients (64–1024B records, 256 in-flight per client), Presto saturates line-rate with 1 core, versus 5 for TAS (3 TAS + 2 app). The gain comes from removing sequencing, protocol handling, and payload copying from the CPU in the common case (no loss). Similar RDMA acceleration is hard, especially for variable-sized records, since it needs multiple round-trips to sequence and write each record [44].

5.4 Robustness

Packet drops. We measure the streaming throughput while randomly discarding packets with varying probability. This workload maintains many in-flight packets, ensuring losses trigger OOO processing. Due to recirculation bottlenecks (§4), we always use a TAS sender. This isolates the evaluation to receiver-side reassembly.

Figure 13a shows that Presto sustains full throughput up to a 0.1% drop rate, whereas TAS declines earlier—2× lower throughput at 0.1% loss. Both provide 1-OOO reassembly on the receiver, but Presto's faster acknowledgments accelerate recovery. Linux[3] withstands higher loss rates via selective recovery (e.g., SACK), but cannot reach terabit throughput even without loss [14], while Presto prioritizes fast, bounded-state hardware recovery. The ASIC-based Chelsio TOE degrades steeply even under moderate loss, and traditional RDMA drops all out-of-order segments, performing poorly under loss [68, 94].

[3]Results for Linux and the Chelsio TOE can be found in [91].

OOO	Flow Completion Time (FCT) [ms]							
	Foreground				Background			
	90p		99.9p		90p		99.9p	
NewReno								
max	4.97	—	17.05	—	68	—	283	—
0	5.80	×1.17	18.49	×1.08	91	×1.34	317	×1.12
1	5.03	×1.01	17.05	×1.00	84	×1.23	300	×1.06
2	4.97	×1.00	17.05	×1.00	78	×1.15	295	×1.04
DCTCP								
max	4.69	—	12.95	—	62	—	278	—
0	5.13	×1.09	28.52	×2.20	71	×1.14	283	×1.02
1	4.70	×1.00	12.91	×1.00	62	×1.00	279	×1.00
2	4.70	×1.00	12.98	×1.00	63	×1.01	275	×0.99

Table 3: Simulated FCTs across OOO tracking fidelity levels.

Congestion control. To evaluate congestion control, we simulate incast by adding a traffic shaper frontend to a TAS receiver. The shaper limits ingress bandwidth (corresponding to the incast degree), uses tail-drop, and marks ECN above a queue occupancy threshold (determined experimentally as 130). Figure 13b presents the results for a single-connection streaming workload under varying incast degrees. Black ticks mark the optimal bandwidth achievable at each degree.

With its control-plane-driven DCTCP and credit-based rate enforcement, Presto effectively regulates transmission, minimizing loss and matching the shaped line rate. In contrast, Presto without congestion control (Presto-NoCC) transmits aggressively and causes excessive tail drops. Even when traffic shaping is disabled (incast degree=1), the Presto-NoCC sender significantly outpaces the TAS receiver, resulting in up to 50× lower throughput. TAS and Linux also regulate their rates effectively but fail to achieve maximum possible throughput at low incast degrees (≤ 2) due to transport stack inefficiencies.

Large-scale simulation. We next study large-scale behavior via simulation. Using TCT's ns-3 [57] (TCT disabled), we test whether bounded OOO reassembly and selective recovery suffice for real datacenter loss patterns. We model a 96-node leaf-spine datacenter topology with latency-sensitive partition-aggregate incasts of 8KB queries competing with bandwidth-heavy background flows, drawn from real-world traces [57]. We extend ns-3's TCP to support varying reassembly fidelity, tracking up to N OOO intervals. OOO-0 drops all OOO segments, while OOO-max tracks all segments with unlimited state.

Table 3 compares the impact of different OOO fidelities. As expected, the OOO-0/go-back-N baseline performs poorly, greatly inflating tail FCTs. In contrast, Presto's OOO-1 design with SACK substantially improves tail FCT, especially under DCTCP, which reacts proactively to congestion. Beyond 1–2 intervals, fidelity yields diminishing returns: congestion losses are typically bursty, so a few intervals suffice for efficient SACK-based recovery. Takeaway: selective recovery is compatible with Presto's RMT pipeline, and a small, bounded OOO state (1–2 intervals) captures most of the benefit while remaining implementable on today's hardware.

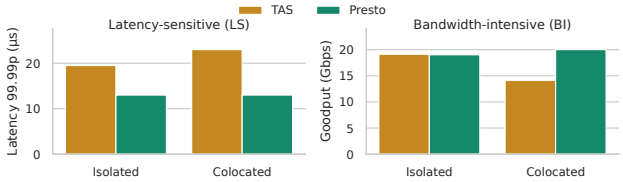


Figure 14: Performance isolation for connections.

Performance isolation. Run-to-completion TCP stacks struggle to isolate latency-sensitive (LS) and bandwidth-intensive (BI) connections. Specifically, TAS hashes connections to cores, mixing LS and BI connections, which causes interference, inflating LS tail latency and reducing BI bandwidth [14].

To evaluate performance isolation, we co-locate an echo server RPC workload with 32 latency-sensitive (LS) connections (single in-flight request) and a streaming workload with 4 bandwidth-intensive (BI) connections (5 Gbps each). We first measure both in isolation, then compare with concurrent execution. We provision TAS with two fast-path cores to avoid bottlenecks.

As Figure 14 shows, Presto delivers optimal performance for both LS and BI flows: consistent per-packet overhead and independent DMA channels steer payload directly to application cores, preventing cache pollution. Under co-location, TAS shows an 18% higher LS tail latency and 30% lower BI bandwidth.

5.5 Generalizability

Prior FPGA work implements TCP as a monolithic circuit, demanding ever-shorter timing and more power to scale. Presto eases this with fine-grained per-state pipelining, even within a single connection. To investigate whether Presto generalizes to FPGA-based SmartNICs, we port Presto's stateful egress pipeline to the AMD Alveo U250 [3] using the VitisNet P4 compiler [4]. Our design meets 10ns timing for 300 Mpps while using only 2.5% of LUTs and 0.78% of BRAM to support 32K connections, consuming 4.93W worst-case power. The low resource footprint leaves headroom to replicate the pipeline and shard connections for higher rates and connections.

By comparison, on a similar FPGA with identical timing constraint, Tonic [5] peaks at 100 Mpps and consumes over 20% of LUTs and BRAM, supporting only 1K connections. Beehive [58] peaks at 2.6 Mpps with comparable resource use. Direct comparison is limited since each design implements a different subset of TCP features—Tonic drops TCP's byte-stream abstraction in favor of a fixed segment size, while Beehive omits OOO handling and congestion control but implements connection handshake.

6 Related Work

Software stack improvements. Software TCP stacks, including Linux and kernel-bypass designs [10, 42, 48, 75, 105], scale by distributing connections across threads with sequential per-connection processing [14, 48]. Despite optimized interfaces [22, 37, 42, 59–61, 74, 103], drivers [27, 29, 34, 76, 80, 85], hardware assistance [23, 24, 39–41, 92], and disaggregation [15, 91], these remain CPU-limited, failing to exploit network processing's inherent packet-level parallelism. Presto instead pipelines TCP state updates across RMT stages even for a single connection, achieving line-rate processing with ASIC-class efficiency.

Co-designed transports. Tightly coupled application-transport designs [25, 38, 45, 46, 53, 69, 93], including RDMA [79], achieve high performance but burden developers with transport complexities, buffer management, and restrictive execution models, sacrificing generality and software reuse. Presto maintains broad TCP and POSIX compatibility, with flexibility for diverse application needs.

FPGA stacks. FPGAs enable flexible offload [28, 77], including for TCP. Tonic [5] provides a sender-only TCP stack (evaluated in simulation), while other efforts [58, 81, 107] do not integrate with host applications [58, 81, 107]. Partial offloads like ZeroNIC [95] and Enso [82] accelerate payload copies but leave TCP on the CPU.

These designs implement TCP state processing as monolithic *circuits* with tight per-packet timing (e.g., 10 ns in Tonic [5]), making them hard to extend and power-intensive at high frequencies. Presto maps TCP to a pipelined match-action architecture that achieves (§5.5) higher throughput and lower resource usage than single-circuit designs, while retaining P4 agility over Verilog [77, 83].

SmartNIC stacks. SmartNICs are widely used for network virtualization [65, 77, 86, 100]. AccelTCP [70] and IO-TCP [52] partially offload TCP; FlexTOE [91] fully offloads the TCP data-path, but faces single-core NPU limits [18, 91]; Falcon [94] and SCR [106] pair fixed transports with NPU congestion control. Data-parallel decompositions across NPU cores [15, 64, 91] are bounded by inter-core queueing, per-core variability, and serialized per-connection logic (FlexTOE's *Protocol* module bottlenecks at $\approx$ 8 Mpps/core [91]). This serialization is fundamental: even on a CPU, ZeroNIC [95] reports a 560 Gbps for single-core ceiling at 9K MTU. By pipelining TCP state updates across RMT, Presto extracts instruction-level parallelism, sustaining terabits within a single connection.

RMT transport acceleration. RMT is emerging as a platform for flexible packet processing [50, 54, 56, 102]. NanoTransport [6] uses P4 but offloads reassembly and packetization to FPGAs; Presto realizes both fully within RMT. R2P2 [53] uses RMT for UDP-based RPC load balancing. GEM implements lightweight RDMA for external memory access [49, 51, 84]. Conweave [96] applies RMT to buffer out-of-order RDMA packets for network load balancing, and LinkGuardian [43] adds link-local retransmissions. Other proposals leverage RMT for scheduling and congestion control [35, 87–89]. None provide full transport offload or obviate the need for an end-to-end reliable protocol like TCP, as Presto does.

7 Discussion

We now turn to Presto's limitations and its generalizability [90].

Datacenter (DC) focus. Presto targets DC environments—Bpps rates, Tbps bandwidth, μs latency, ASIC-efficiency, and software flexibility—exploiting their defining traits: persistent connections, congestion-driven (not corruption-driven) loss, short RPCs, and benign traffic patterns. Wide-area networks (WANs) and High-performance computing (HPC) settings may warrant different trade-offs, yet Presto's principles still apply: BBR [16] is implementable, and although WAN loss rates may demand more OOO intervals, Presto's OOO design may track in-flight segments efficiently for large bandwidth-delay-product (BDP) flows [63].

Advanced DC techniques. Several techniques remain future work. Multi-pathing for AI cluster congestion [94, 106] maps onto Presto's OOO intervals, which tracks cross-path reordering, though we do not evaluate it. RACK [19], which tracks per-packet transmission times and scans O(window) unacknowledged packets, is harder given RMT's strict state limits; but such settings run few large flows per host at modest packet rates, permitting slower RMT pipelines with stronger per-stage processing. Network stack disaggregation, enabled by Presto's switch-based prototype, is also future work.

Message-boundary parsing. Flexible offloads for higher-level protocols (e.g., RPC load balancing or access control) that inspect payloads need application-level message boundaries, which TCP's byte-stream semantics obscure when delimiters straddle segments (§5.3.1). Pushing message orientation into the transport solves this.

Head-of-Line (HoL) blocking with libPresto-ZC. Presto's zero-copy API transmits segments in allocation order, occasionally causing HoL delays. This rarely matters in practice: small RPCs can use copy-based APIs cheaply, and bulk flows can spread across connections. Presto can also expose queue-based interfaces (e.g., `ibverbs`-style), via table-lookup address translation.

Generalizability beyond TCP. Presto's principles extend beyond TCP to many transports [90]. As Tonic [5] observes, reliable transports share common structural patterns: tracking in-flight segments for reassembly, recovering losses via retransmissions, and regulating in-flight segments to avoid congestion, differing only in mechanism. Presto realizes these efficiently on RMT architecture by overcoming the dependencies of Figure 1.

Message-oriented transports (e.g., RDMA) are a case in point, replacing byte-stream reassembly with explicit message boundaries—no fundamental changes to Presto's design. Messages are tracked by packet sequence number (PSN) rather than byte offsets, and OOO intervals map directly to PSN ranges. By decoupling data placement from application notification, Presto eliminates on-NIC buffering: each message either carries a host buffer address or draws from a queue of pre-allocated buffers (cf. RDMA WRITE vs. SEND), which Presto indexes to place directly into host memory, advancing the base index as the window moves. We leave implementation and evaluation to future work.

8 Conclusion

Presto redefines the design-space of high-performance TCP stacks by showing that full transport functionality fits within RMT constraints. Embracing pipeline-parallelism, optimistic concurrency, and pseudo-segment updates, Presto delivers terabit throughput, low latency, and hardware efficiency while preserving software flexibility. Our Intel Tofino 2 prototype proves that RMT data planes can support stateful, loss-resilient, and scalable transport, laying a foundation for next-generation SmartNICs and datacenter networks.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their helpful feedback. This work is supported in part by NSF grant 2212193 and the University of Washington Center for the Future of Cloud Infrastructure (FOCI). This work raises no ethical concerns.

References

[1] Anurag Agrawal and Changhoon Kim. 2020. Intel Tofino 2—A 12.9 Tbps P4-programmable Ethernet Switch. In *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society, 1–32. https://www.hc32.hotchips.org/assets/program/conference/day2/HotChips2020_Networking_Tofino.pdf

[2] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *Proceedings of the 2010 Conference of the ACM Special Interest Group on Data Communication* (New Delhi, India) *(SIGCOMM '10)*. Association for Computing Machinery, New York, NY, USA, 63–74. doi:10.1145/1851182.1851192

[3] AMD. 2026. *Alveo U200 and U250 Data Center Accelerator Cards Data Sheet*. AMD. Retrieved June 29, 2026 from https://docs.amd.com/r/en-US/ds962-u200-u250/Alveo-Product-Details

[4] AMD. 2026. *Vitis Networking P4*. Vivado Software. Retrieved June 29, 2026 from https://www.amd.com/en/products/adaptive-socs-and-fpgas/intellectual-property/ef-di-vitisnetp4.html

[5] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI '20)*. USENIX Association, USA, 93–110. doi:10.5555/3388242.3388250

[6] Serhat Arslan, Stephen Ibanez, Alex Mallery, Changhoon Kim, and Nick McKeown. 2021. NanoTransport: A Low-Latency, Programmable Transport Layer for NICs. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)* (Virtual Event, USA) *(SOSR '21)*. Association for Computing Machinery, New York, NY, USA, 13–26. doi:10.1145/3482898.3483365

[7] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. 2023. Empowering Azure Storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 49–67. https://www.usenix.org/conference/nsdi23/presentation/bai

[8] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. 2020. Virtual Consensus in Delos. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 617–632. https://www.usenix.org/conference/osdi20/presentation/balakrishnan

[9] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. 2013. CORFU: A distributed shared log. *ACM Trans. Comput. Syst.* 31, 4, Article 10 (Dec. 2013), 24 pages. doi:10.1145/2535930

[10] 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 *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI '14)*. USENIX Association, USA, 49–65. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay

[11] David Borman, Robert T. Braden, Van Jacobson, and Richard Scheffenegger. 2014. *TCP Extensions for High Performance*. Internet Engineering Task Force (IETF). doi:10.17487/RFC7323

[12] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference* (Hong Kong, China) *(SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 99–110. doi:10.1145/2486001.2486011

[13] Robert T. Braden. 1989. *Requirements for Internet Hosts—Communication Layers*. Internet Engineering Task Force (IETF). doi:10.17487/RFC1122

[14] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding host network stack overheads. In *Proceedings of the ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 65–77. doi:10.1145/3452296.3472888

[15] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. 2022. Towards μs tail latency and terabit ethernet: disaggregating the host network stack. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 767–779. doi:10.1145/3544216.3544230

[16] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2017. BBR: congestion-based congestion control. *Commun. ACM* 60, 2 (Jan. 2017), 58–66. doi:10.1145/3009824

[17] Chelsio Communications. 2016. *T6 ASIC: High Performance, Dual Port Unified Wire 1/10/25/40/50/100Gb Ethernet Controller*. Retrieved June 29, 2026 from https://www.chelsio.com/wp-content/uploads/resources/Chelsio-Terminator-6-Brief.pdf

[18] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, and Zeke Wang. 2024. Demystifying Datapath Accelerator Enhanced Off-path SmartNIC. In *32nd IEEE International Conference on Network Protocols, ICNP 2024, Charleroi, Belgium, October 28-31, 2024*. IEEE, 1–12. doi:10.1109/ICNP61940.2024.10858560

[19] Yuchung Cheng, Neal Cardwell, Nandita Dukkipati, and Priyaranjan Jha. 2021. *RFC 8985: The RACK-TLP Loss Detection Algorithm for TCP*. Google. doi:10.17487/RFC8985

[20] Intel Corporation. 2023. *Intel Tofino 2*. Retrieved June 29, 2026 from https://www.intel.com/content/www/us/en/products/sku/218648/intel-tofino-2-12-8-tbps-20-stage-4-pipelines/specifications.html

[21] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. 2020. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 325–338. https://www.usenix.org/conference/nsdi20/presentation/ding

[22] Linux Networking Documentation. 2025. *MSG_ZEROCOPY*. Retrieved June 29, 2026 from https://docs.kernel.org/networking/msg_zerocopy.html

[23] Linux Networking Documentation. 2025. *Scaling in the Linux Networking Stack*. Retrieved June 29, 2026 from https://www.kernel.org/doc/html/v6.14/networking/scaling.html

[24] Linux Networking Documentation. 2025. *Segmentation Offloads in the Linux Networking Stack*. Retrieved June 29, 2026 from https://www.kernel.org/doc/Documentation/networking/segmentation-offloads.txt

[25] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA) *(NSDI'14)*. USENIX Association, USA, 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi{ć}

[26] ESnet. 2023. *iPerf3*. Retrieved April 15, 2025 from https://software.es.net/iperf/

[27] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2021. PacketMill: toward per-Core 100-Gbps networking. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/3445814.3446724

[28] 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 *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, USA) *(NSDI '18)*. USENIX Association, USA, 51–64. https://www.usenix.org/conference/nsdi18/presentation/firestone

[29] Mario Flajslik and Mendel Rosenblum. 2013. Network interface design for low latency request-response protocols. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (San Jose, CA) *(USENIX ATC'13)*. USENIX Association, USA, 333–346. https://www.usenix.org/conference/atc13/technical-sessions/presentation/flajslik

[30] Sally Floyd, Jamshid Mahdavi, Matt Mathis, and Dr. Allyn Romanow. 1996. *TCP Selective Acknowledgment Options*. Internet Engineering Task Force (IETF). doi:10.17487/RFC2018

[31] Linux Foundation. 2024. *Storage Performance Development Kit (SPDK)*. Retrieved June 29, 2026 from http://www.spdk.io

[32] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. 2024. RDMA over Ethernet for Distributed Training at Meta Scale. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 57–70. doi:10.1145/3651890.3672233

[33] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan

Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. 2021. When Cloud Storage Meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 519–533. https://www.usenix.org/conference/nsdi21/presentation/gao

[34] Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Q. Maguire Jr., and Dejan Kostić. 2022. Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 807–827. https://www.usenix.org/conference/nsdi22/presentation/ghasemirahni

[35] Prateesh Goyal, Preey Shah, Naveen Kr. Sharma, Mohammad Alizadeh, and Thomas E. Anderson. 2020. Backpressure Flow Control. In *Proceedings of the 2019 Workshop on Buffer Sizing* (Palo Alto, CA, USA) *(BS '19)*. Association for Computing Machinery, New York, NY, USA, Article 4, 3 pages. doi:10.1145/3375235.3375239

[36] Vladimir Gurevich and Andy Fingerhut. 2021. *P416 Programming for Intel Tofino using Intel P4 Studio*. Retrieved June 29, 2026 from https://opennetworking.org/wp-content/uploads/2021/05/2021-P4-WS-Vladimir-Gurevich-Slides.pdf

[37] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI '12)*. USENIX Association, USA, 135–148. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/han

[38] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. 2020. TCP = RDMA: CPU-efficient Remote Storage Access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 127–140. https://www.usenix.org/conference/nsdi20/presentation/hwang

[39] Intel. 2006. *Accelerating High-Speed Networking with Intel I/O Acceleration Technology*. Retrieved June 29, 2026 from https://www.intel.com/content/dam/doc/white-paper/i-o-acceleration-technology-paper.pdf

[40] Intel. 2012. *Intel Data Direct I/O Technology (Intel DDIO): A Primer*. Retrieved June 29, 2026 from https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf

[41] Intel. 2019. *Jumbo Frames and Jumbo Packets Notes*. Retrieved June 29, 2026 from https://www.intel.com/content/www/us/en/support/articles/000006639/ethernet-products.html

[42] 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 *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA) *(NSDI '14)*. USENIX Association, USA, 489–502. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong

[43] Raj Joshi, Cha Hwan Song, Xin Zhe Khooi, Nishant Budhdev, Ayush Mishra, Mun Choon Chan, and Ben Leong. 2023. Masking Corruption Packet Losses in Datacenter Networks with Link-local Retransmission. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 288–304. doi:10.1145/3603269.3604853

[44] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 437–450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[45] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 185–201. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia

[46] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2019. Datacenter RPCs Can Be General and Fast. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI '19)*. USENIX Association, USA, 1–16. https://www.usenix.org/conference/nsdi19/presentation/kalia

[47] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 67–81. doi:10.1145/2872362.2872367

[48] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP Acceleration as an OS Service. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 24, 16 pages. doi:10.1145/3302424.3303985

[49] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. 2020. TEA: Enabling State-Intensive Network Functions on Programmable Switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 90–106. doi:10.1145/3387514.3405855

[50] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. 2021. RedPlane: enabling fault-tolerant stateful in-switch applications. In *Proceedings of the 2021 ACM SIGCOMM Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 223–244. doi:10.1145/3452296.3472905

[51] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan. 2018. Generic External Memory for Switch Data Planes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks* (Redmond, WA, USA) *(HotNets '18)*. Association for Computing Machinery, New York, NY, USA, 1–7. doi:10.1145/3286062.3286063

[52] Taehyun Kim, Deondre Martin Ng, Junzhi Gong, Youngjin Kwon, Minlan Yu, and KyoungSoo Park. 2023. Rearchitecting the TCP Stack for I/O-Offloaded Content Delivery. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 275–292. https://www.usenix.org/conference/nsdi23/presentation/kim-taehyun

[53] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. 2019. R2P2: Making RPCs First-Class Datacenter Citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 863–879. https://www.usenix.org/conference/atc19/presentation/kogias-r2p2

[54] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. MIND: In-Network Memory Management for Disaggregated Data Centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 488–504. doi:10.1145/3477132.3483561

[55] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 467–483. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/li

[56] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. 2020. Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 387–406. https://www.usenix.org/conference/osdi20/presentation/li-jialin

[57] Hwijoon Lim, Wei Bai, Yibo Zhu, Youngmok Jung, and Dongsu Han. 2021. Towards timeout-less transport in commodity datacenter networks. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 33–48. doi:10.1145/3447786.3456227

[58] Katie Lim, Matthew Giordano, Theano Stavrinos, Irene Zhang, Jacob Nelson, Baris Kasikci, and Thomas Anderson. 2024. Beehive: A Flexible Network Stack for Direct-Attached Accelerators. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 393–408. doi:10.1109/MICRO61859.2024.00037

[59] 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 *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 339–352. doi:10.1145/2872362.2872391

[60] Linux. 2019. *Efficient I/O with io_uring*. Retrieved June 29, 2026 from https://kernel.dk/io_uring.pdf

[61] Linux. 2020. *io_uring—Linux manual page*. Retrieved June 29, 2026 from https://man7.org/linux/man-pages/man7/io_uring.7.html

[62] Xuhao Luo, Shreesha G. Bhat, Jiyu Hu, Ramnatthan Alagappan, and Aishwarya Ganesan. 2024. LazyLog: A New Shared Log Abstraction for Low-Latency Applications. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 296–312. doi:10.1145/3694715.3695983

[63] Kai Lv, Jinyang Li, Pengyi Zhang, Heng Pan, Luyang Li, Shuihai Hu, Zhenyu Li, Gaogang Xie, Jingbin Zhou, and Kun Tan. 2025. OmniDMA: Scalable RDMA Transport over WAN. In *Proceedings of the 9th Asia-Pacific Workshop on Networking (APNET '25)*. Association for Computing Machinery, New York, NY, USA, 135–141. doi:10.1145/3735358.3735373

[64] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP*

'19). Association for Computing Machinery, New York, NY, USA, 399–413. doi:10.1145/3341301.3359657

[65] Nirav Mehta. 2022. *The next wave of Google Cloud infrastructure innovation: New C3 VM and Hyperdisk*. Google Cloud. Retrieved June 29, 2026 from https://cloud.google.com/blog/products/compute/introducing-c3-machines-with-googles-custom-intel-ipu

[66] memcached. 2020. *Memcached: Free & open source, high-performance, distributed memory object caching system*. https://memcached.org/

[67] 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 *Proceedings of the 2015 Conference of the ACM Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 537–550. doi:10.1145/2785956.2787510

[68] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting Network Support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 313–326. doi:10.1145/3230543.3230557

[69] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 221–235. doi:10.1145/3230543.3230564

[70] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI '20)*. USENIX Association, USA, 77–92. https://www.usenix.org/conference/nsdi20/presentation/moon

[71] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2018. Restructuring endpoint congestion control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 30–43. doi:10.1145/3230543.3230553

[72] Netberg. 2024. *Netberg Aurora 810*. Retrieved June 29, 2026 from https://netbergtw.com/products/aurora-810/

[73] AMD Pensando. 2024. *AMD Pensando 2nd Generation (Elba) Data Processing Unit*. Retrieved June 29, 2026 from https://www.amd.com/content/dam/amd/en/documents/pensando-technical-docs/product-briefs/pensando-elba-product-brief.pdf

[74] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. 2012. Improving Network Connection Locality on Multicore Systems. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 337–350. doi:10.1145/2168836.2168870

[75] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. USENIX Association, Broomfield, CO, 1–16. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter

[76] Solal Pirelli and George Candea. 2020. A Simpler and Faster NIC Driver Model for Network Functions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 225–241. https://www.usenix.org/conference/osdi20/presentation/pirelli

[77] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2016. A reconfigurable fabric for accelerating large-scale datacenter services. *Commun. ACM* 59, 11 (Oct. 2016), 114–122. doi:10.1145/2996868

[78] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. 2024. Alibaba HPN: A Data Center Network for Large Language Model Training. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 691–706. doi:10.1145/3651890.3672265

[79] RDMA Consortium. 2009. *Architectural specifications for RDMA over TCP/IP*. Retrieved June 29, 2026 from http://www.rdmaconsortium.org/

[80] Luigi Rizzo. 2012. Netmap: a novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 9. https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo

[81] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio López-Buedo. 2019. Limago: An FPGA-Based Open-Source 100 GbE TCP/IP Stack. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 286–292. doi:10.1109/FPL.2019.00053

[82] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S. Berger, James C. Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. 2023. Enso: A Streaming Interface for NIC-Application Communication. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 1005–1025. https://www.usenix.org/conference/osdi23/presentation/sadok

[83] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S. Berger, James C. Hoe, Aurojit Panda, and Justine Sherry. 2021. We Need Kernel Interposition over the Network Dataplane. In *Proceedings of the 2021 Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 152–158. doi:10.1145/3458336.3465281

[84] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostić, and Marco Chiesa. 2023. A High-Speed Stateful Packet Processing Approach for Tbps Programmable Switches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1237–1255. https://www.usenix.org/conference/nsdi23/presentation/scazzariello

[85] Henry N. Schuh, Arvind Krishnamurthy, David Culler, Henry M. Levy, Luigi Rizzo, Samira Khan, and Brent E. Stephens. 2024. CC-NIC: a Cache-Coherent Interface to the NIC. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 52–68. doi:10.1145/3617232.3624868

[86] Amazon Web Services. 2023. *AWS: The Nitro System Journey*. Retrieved June 29, 2026 from https://docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/the-nitro-system-journey.html

[87] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. 2017. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 67–82. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/sharma

[88] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating Fair Queueing on Reconfigurable Switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 1–16. https://www.usenix.org/conference/nsdi18/presentation/sharma

[89] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. 2020. Programmable Calendar Queues for High-speed Packet Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 685–699. https://www.usenix.org/conference/nsdi20/presentation/sharma

[90] Rajath Shashidhara. 2025. *Building Flexible Data Center Network Stacks for the Terabit Era*. Ph. D. Dissertation. University of Washington.

[91] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. 2022. FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 87–102. https://www.usenix.org/conference/nsdi22/presentation/shashidhara

[92] Pravin Shinde, Antoine Kaufmann, Timothy Roscoe, and Stefan Kaestle. 2013. We Need to Talk about NICs. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems* (Santa Ana Pueblo, New Mexcio) *(HotOS '13)*. USENIX Association, USA, 1. https://www.usenix.org/conference/hotos13/session/shinde

[93] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 2020. 1RMA: Re-envisioning Remote Memory Access for Multi-tenant Datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 708–721. doi:10.1145/3387514.3405897

[94] Arjun Singhvi, Nandita Dukkipati, Prashant Chandra, Hassan M. G. Wassel, Naveen Kr. Sharma, Anthony Rebello, Henry Schuh, Praveen Kumar, Behnam Montazeri, Neelesh Bansod, Sarin Thomas, Inho Cho, Hyojeong Lee Seibert, Baijun Wu, Rui Yang, Yuliang Li, Kai Huang, Qianwen Yin, Abhishek Agarwal, Srinivas Vaduvatha, Weihuang Wang, Masoud Moshref, Tao Ji, David Wetherall, and Amin Vahdat. 2025. Falcon: A Reliable, Low Latency Hardware Transport. In *Proceedings of the ACM SIGCOMM 2025 Conference* (São Francisco Convent, Coimbra, Portugal) *(SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 248–263. doi:10.1145/3718958.3754353

[95] Athinagoras Skiadopoulos, Zhiqiang Xie, Mark Zhao, Qizhe Cai, Saksham Agarwal, Jacob Adelmann, David Ahern, Carlo Contavalli, Michael Goldflam, Vitaly Mayatskikh, et al. 2024. High-throughput and flexible host networking for accelerated computing. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 405–423.

[96] Cha Hwan Song, Xin Zhe Khooi, Raj Joshi, Inho Choi, Jialin Li, and Mun Choon Chan. 2023. Network Load Balancing with In-network Reordering Support for RDMA. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 816–831. doi:10.1145/3603269.3604849

[97] SPDK. 2024. *Block Device User Guide*. Retrieved June 29, 2026 from https://spdk.io/doc/bdev.html

[98] SPDK. 2024. *Running benchmarks with Perf Tool*. Retrieved June 29, 2026 from https://spdk.io/doc/nvme.html

[99] Matheus Stolet, Liam Arzola, Simon Peter, and Antoine Kaufmann. 2023. Virtuoso: High Resource Utilization and μs-scale Performance Isolation in a Shared Virtual Machine TCP Network Stack. *arXiv preprint arXiv:2309.14016v4* (2023). https://arxiv.org/pdf/2309.14016v4

[100] Naru Sundar, Brad Burres, Yadong Li, Dave Minturn, Brian Johnson, and Nupur Jain. 2023. An In-depth Look at the Intel IPU E2000. In *2023 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 9.4. 162–164. doi:10.1109/ISSCC42615.2023.10067333

[101] Igor Sysoev. 2026. *NGINX: HTTP and reverse proxy server*. F5, Inc. Retrieved June 12, 2026 from https://nginx.org/

[102] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. 2021. Concordia: Distributed Shared Memory with In-Network Cache Coherence. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 277–292. https://www.usenix.org/conference/fast21/presentation/wang

[103] 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* (Denver, CO, USA) *(USENIX ATC '16)*. USENIX Association, USA, 43–56. https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata

[104] Yifan Yuan, Omar Alama, Jiawei Fei, Jacob Nelson, Dan R. K. Ports, Amedeo Sapio, Marco Canini, and Nam Sung Kim. 2022. Unlocking the Power of Inline Floating-Point Operations on Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 683–700. https://www.usenix.org/conference/nsdi22/presentation/yuan

[105] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 195–211. doi:10.1145/3477132.3483569

[106] Chenxingyu Zhao, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. 2025. White-Boxing RDMA with Packet-Granular Software Control. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 427–449. https://www.usenix.org/conference/nsdi25/presentation/zhao-chenxingyu

[107] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. 2020. Achieving 100Gbps Intrusion Prevention on a Single Server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1083–1100. https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng