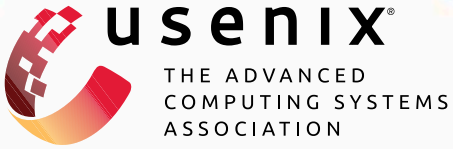

FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism

Rajath Shashidhara, *University of Washington;* Tim Stamler, *UT Austin;*
Antoine Kaufmann, *MPI-SWS;* Simon Peter, *University of Washington*

https://www.usenix.org/conference/nsdi22/presentation/shashidhara

This paper is included in the Proceedings of the
19th USENIX Symposium on Networked Systems
Design and Implementation.

April 4–6, 2022 • Renton, WA, USA

978-1-939133-27-4

Open access to the Proceedings of the
19th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism

Rajath Shashidhara[1] Tim Stamler[2] Antoine Kaufmann[3] Simon Peter[1]
[1]University of Washington [2]UT Austin [3]MPI-SWS

Abstract

FlexTOE is a flexible, yet high-performance TCP offload engine (TOE) to SmartNICs. FlexTOE eliminates almost all host data-path TCP processing and is fully customizable. FlexTOE interoperates well with other TCP stacks, is robust under adverse network conditions, and supports POSIX sockets.

FlexTOE focuses on data-path offload of established connections, avoiding complex control logic and packet buffering in the NIC. FlexTOE leverages fine-grained parallelization of the TCP data-path and segment reordering for high performance on wimpy SmartNIC architectures, while remaining flexible via a modular design. We compare FlexTOE on an Agilio-CX40 to host TCP stacks Linux and TAS, and to the Chelsio Terminator TOE. We find that Memcached scales up to 38% better on FlexTOE versus TAS, while saving up to 81% host CPU cycles versus Chelsio. FlexTOE provides competitive performance for RPCs, even with wimpy SmartNICs. FlexTOE cuts 99.99th-percentile RPC RTT by 3.2× and 50% versus Chelsio and TAS, respectively. FlexTOE's data-path parallelism generalizes across hardware architectures, improving single connection RPC throughput up to 2.4× on x86 and 4× on BlueField. FlexTOE supports C and XDP programs written in eBPF. It allows us to implement popular data center transport features, such as TCP tracing, packet filtering and capture, VLAN stripping, flow classification, firewalling, and connection splicing.

1 Introduction

TCP remains the default protocol in many networks, even as its CPU overhead is increasingly a burden to application performance [3, 17, 46]. A long line of improvements to software TCP stack architecture has reduced overheads: Careful packet steering improves cache-locality for multi-cores [17, 24, 45], kernel-bypass enables safe direct NIC access from user-space [3, 46], application libraries avoid system calls for common socket operations [17], and fast-paths drastically reduce TCP processing overheads [19]. Yet, even with these optimizations, communication-intensive applications spend up to 48% of per-CPU cycles in the TCP stack and NIC driver (§2.1).

Offload promises further reduction of CPU overhead. While moving parts of TCP processing, such as checksum and segmentation, into the NIC is commonplace [54], full TCP offload engines (TOEs) [6, 7, 33] have so far failed to find widespread adoption. A primary reason is that fixed offloads [56] limit protocol evolution after deployment [9, 29, 36]. Tonic [2] provides building blocks for flexible transport protocol offload to FPGA-SmartNICs, but FPGA development is still difficult and slow.

We present FlexTOE, a high-performance, yet flexible offload of the widely-used TCP protocol. FlexTOE focuses on scenarios that are common in data centers, where connections are long-lived and small transfers are common [29]. FlexTOE offloads the TCP data-path to a network processor (NPU) based SmartNIC, enabling full customization of transport logic and flexibility to implement data-path features whose requirements change frequently in data centers. Applications interface directly but transparently with the FlexTOE data-path through the *libTOE* library that implements POSIX sockets, while FlexTOE offloads all TCP data-path processing (§2.1).

TCP data-path offload to SmartNICs is challenging. SmartNICs support only restrictive programming models with stringent per-packet time budgets and are geared towards massive parallelism with wimpy cores [26]. They often lack timers, as well as floating-point and other computational support, such as division. Finally, offload has to mask high-latency operations that cross PCIe. On the other hand, TCP requires computationally intensive and stateful code paths to track in-flight segments, for reassembly and retransmission, and to perform congestion control [2]. For each connection, the TCP data-path needs to provide low processing tail latency and high throughput and is also extremely sensitive to reordering.

Resolving the gap between TCP's requirements and SmartNIC hardware capabilities requires careful offload design to efficiently utilize SmartNIC capabilities. Targeting FlexTOE at the TCP data-path of established connections avoids complex control logic in the NIC. FlexTOE's offloaded data-path is one-shot for each TCP segment—segments are never buffered in the NIC. Instead, per-socket buffers are kept in per-process host memory where libTOE interacts with them directly. Connection management, retransmission, and congestion control are part of a separate control-plane, which executes in its own protection domain, either on control cores of the SmartNIC or on the host. To provide scalability and flexibility, we decompose the TCP data-path into fine-grained modules that keep private state and communicate explicitly. Like microservices [29], FlexTOE modules leverage a data-parallel execution model that maximizes SmartNIC resource use **and** simplifies customization. We organize FlexTOE modules into a *data-parallel computation pipeline*. We also *reorder* segments on-the-fly to support parallel, out-of-order processing of pipeline stages, while enforcing in-order TCP segment delivery. To our knowledge, no prior work attempting full TCP data-path offload to NPU SmartNICs exists.

We make the following contributions:

- We characterize the CPU overhead of TCP data-path processing for common data center applications (§2.1). Our analysis shows that up to 48% of per-CPU cycles are spent in TCP data-path processing, even with optimized TCP stacks.

- We present FlexTOE, a flexible, high-performance TCP offload engine (§3). FlexTOE leverages data-path processing with fine-grained parallelism for performance, but remains flexible via a modular design. We show how to decompose TCP into a data-path and a control-plane, and the data-path into a data-parallel pipeline of processing modules to hide SmartNIC processing and data access latencies.
- We implement FlexTOE on the Netronome Agilio-CX40 NPU SmartNIC architecture, as well as x86 and Mellanox BlueField (§4). Using FlexTOE design principles, we are the first to demonstrate that NPU SmartNICs can support scalable, yet flexible TCP data-path offload. Our code is available at https://tcp-acceleration-service.github.io/FlexTOE.
- We evaluate FlexTOE on a range of workloads and compare to Linux, the high-performance TAS [19] network stack, and a Chelsio Terminator TOE [6] (§5). We find that the Memcached [32] key-value store scales throughput up to 38% better on FlexTOE than using TAS, while saving up to 81% host CPU cycles versus Chelsio. FlexTOE cuts 99.99th-percentile RPC RTT by 3.2× and 50% versus Chelsio and TAS respectively, 27% higher throughput than Chelsio for bidirectional long flows,and an order of magnitude higher throughput under 2% packet loss than Chelsio. We extend the FlexTOE data-path with debugging and auditing functionality to demonstrate flexibility. FlexTOE maintains high performance when interoperating with other network stacks. FlexTOE's data-path parallelism generalizes across platforms, improving single connection RPC throughput up to 2.4× on x86 and 4× on BlueField.

2 Background

We motivate FlexTOE by analyzing TCP host CPU processing overheads of related approaches (§2.1). We then place FlexTOE in context of this and further related work (§2.2). Finally, we survey the relevant *on-path* SmartNIC architecture (§2.3).

2.1 TCP Impact on Host CPU Performance

We quantify the impact of different TCP processing approaches on host CPU performance in terms of CPU overhead, execution efficiency, and cache footprint, when processing common RPC-based workloads. We do so by instrumenting a single-threaded Memcached [32] server application using hardware performance counters (cf. §5 for details of our testbed). We use the popular memtier_benchmark [51] to generate the client load, consisting of 32 B keys and values, using as many clients as necessary to saturate the server, executing closed-loop KV transactions on persistent connections. Table 1 shows a breakdown of our server-side results, for each Memcached request-response pair, into NIC driver, TCP/IP stack, POSIX sockets, Memcached application, and other factors.

In-kernel. Linux's TCP stack is versatile but bulky, leading to a large cache footprint, inefficient execution, and high CPU overhead. Stateless offloads [54], such as segmentation and generic receive offload [12], reduce overhead for large transfers, but they have minimal impact on RPC workloads dominated by short flows. We find that Linux executes 12.13 kc per Memcached request on average, with only 10% spent in the application. Not only does Linux have a high instruction and instruction cache (Icache) footprint, but privilege mode switches, scattered global state, and coarse-grained locking lead to 62% of all cycles spent in instruction fetch stalls (frontend bound), cache and TLB misses (backend bound), and branch mispredictions (cf. [19]). These inefficiences result in 1.33 instructions per cycle (IPC), leveraging only 33% of our 4-way issue CPU architecture. Linux is, in principle, easy to modify, but kernel code development is complex and security sensitive. Hence, introducing optimizations and new network functionality to the kernel is often slow [29, 42, 43].

Module	Linux		Chelsio		TAS		FlexTOE	
	kc	%	kc	%	kc	%	kc	%
NIC driver	0.71	6	1.28	14	0.18	5	0	0
TCP/IP stack	4.25	35	0.40	4	1.44	43	0	0
POSIX sockets	2.48	21	2.61	29	0.79	23	0.74	44
Application	1.26	10	1.31	16	0.85	26	0.89	53
Other	3.42	28	3.28	37	0.09	3	0.04	3
Total	12.13	100	8.89	100	3.34	100	1.67	100
Retiring	4.60	38	2.43	27	1.66	48	0.77	46
Frontend bound	3.53	29	1.52	17	0.46	13	0.34	21
Backend bound	3.40	28	4.68	53	1.24	36	0.46	27
Bad speculation	0.55	5	0.26	3	0.13	4	0.09	6
Instructions (k)	16.18		8.14		6.26		2.93	
IPC	1.33		0.92		1.85		1.75	
Icache (KB)	47.50		73.43		39.75		19.00	

Table 1. Per-request CPU impact of TCP processing.

Kernel-bypass. Kernel-bypass, such as in mTCP [17] and Arrakis [46], eliminates kernel overheads by entrusting the TCP stack to the application, but it has security implications [52]. TAS [19] and Snap [29] instead execute a protected user-mode TCP stack on dedicated cores, retaining security and performance. By eliminating kernel calls, TAS spends only 800 cycles in the socket API—31% of Linux's API overhead. TAS also reduces TCP stack overhead to 34% of Linux. TAS reduces Icache footprint, front and back-end CPU stalls, improving IPC by 40% versus Linux, and reducing the total per-request CPU impact to 27% of Linux. However, kernel-bypass still has significant overhead. Only 26% of per-request cycles are spent in Memcached—the remainder is spent in TAS (breakdown in §C).

Inflexible TCP offload. TCP offload can eliminate host CPU overhead for TCP processing. Indeed, TOEs [7] that offload the TCP data-path to the NIC have existed for a long time. Existing approaches, such as the Chelsio Terminator [6], hard-wire the TCP offload. The resulting inflexibility prevents data center operators from adapting the TOE to their needs

and leads to a slow upgrade path due to long hardware development cycles. For example, the Chelsio Terminator line has been slow to adapt to RPC-based data center workloads.

Chelsio's inflexbility shows in our analysis. Despite drastically reducing the host TCP processing cycles to 10% of Linux and 28% of TAS, Chelsio's TOE only modestly reduces the total per-request CPU cycles of Memcached by 27% versus Linux and inflates them by 2.6× versus TAS. Chelsio's design requires interaction through the Linux kernel, leading to a similar execution profile despite executing 50% fewer host instructions per request. In addition, Chelsio requires a sophisticated TOE NIC driver, with complex buffer management and synchronization. Chelsio's design is inefficient for RPC processing and leaves only 16% of the total per-request cycles to Memcached—6% more than Linux and 10% fewer than TAS.

FlexTOE. FlexTOE eliminates all host TCP stack overheads. FlexTOE's instruction (and Icache) footprint is at least 2× lower than the other stacks, leading to an execution profile similar to TAS, where 46% of all cycles are spent retiring instructions. In addition, 53% of all cycles can be spent in Memcached—an improvement of 2× versus TAS, the next best solution. The remaining cycles are spent in the POSIX sockets API, which cannot be eliminated with TCP offload.

FlexTOE is also flexible, allowing operators to modify the TOE at will. For example, we have modified the TCP data-path many times, implementing many features that require TOE modification, including scalable socket API implementations [24, 45], congestion control protocols [1, 34], scalable flow scheduling [53], scalable PCIe communication protocols [44], TCP tracing [13], packet filtering and capture (tcpdump and PCAP), VLAN stripping, programmable flow classification (eBPF [30]), firewalling, and connection splicing similar to AccelTCP [37]. All of these features are desirable in data centers and are adapted frequently.

2.2 Related Work

Beyond the TCP implementations covered in §2.1, we cover here further related work in SmartNIC offload, parallel packet processing, and API and network protocol specialization.

SmartNIC offload. On-path SmartNICs (§2.3), based on network processor units (NPUs) and FPGAs, provide a suitable substrate for flexible offload. Arsenic [47] is an early example of flexible packet multiplexing on a SmartNIC. Microsoft's Catapult [48] offloads network management, while Dagger [22] offloads RPC processing to FPGA-SmartNICs. Neither offloads a transport protocol, like TCP. AccelTCP [37] offloads TCP connection management and splicing [28] to NPU-SmartNICs, but keeps the TCP data-path on the host using mTCP [17]. Tonic [2] demonstrates in simulation that high-performance, flexible TCP transmission offload might be possible, but it stops short of implementing full TCP data-path offload (including receiver processing) in a non-simulated environment. LineFS [20] offloads a distributed file system to an off-path SmartNIC, leveraging parallelization to hide execution latencies of wimpy SmartNIC CPUs and data access across PCIe. Taking inspiration from Tonic and LineFS, but also from actor, and microservice-based approaches presented in iPipe [26], E3 [27], and Click [23, 38], FlexTOE shows how to decompose the TCP data-path into a fine-grained data-parallel pipeline to support full and flexible offload to on-path NPU-SmartNICs.

Parallel packet processing. RouteBricks [8] parallelizes across cores and cluster nodes for high-performance routing, achieving high line-rates but remaining flexible via software programmability. Routing relies on read-mostly state and is simple compared to TCP. FlexTOE applies fine-grained parallelization to complex, stateful code paths.

Specialized APIs and protocols. Another approach to lower CPU utilization is specialization. R2P2 [21] is a UDP-based protocol for remote procedure calls (RPCs) optimized for efficient and parallel processing, both at the end-hosts and in the network. eRPC [18] goes a step further and co-designs an RPC protocol and API with a kernel-bypass network stack to minimize CPU overhead per RPC. RDMA [49] is a popular combination of a networking API, protocol, and a (typically hardware) network stack. iWARP [50], in particular, leverages a TCP stack underneath RDMA, offloading both. These approaches improve processing efficiency, but at the cost of requiring application re-design, all-or-nothing deployments, and operational issues at scale [11], often due to inflexibility [36, 56]. FlexTOE instead offloads the TCP protocol in a flexible manner by relying on SmartNICs. Upper-layer protocols, such as iWARP, can also be implemented using FlexTOE.

2.3 On-path SmartNIC Architecture

On-path SmartNICs[1], such as Marvell Octeon [5], Pensando Capri [10, 55], and Netronome Agilio [39, 40], support massively parallel packet processing with a large pool of flow processing cores (FPCs), but they lack efficient support for sophisticated program control flow and complex computation [26].

We explore offload to the NFP-4000 NPU, used in Netronome Agilio CX SmartNICs [39]. We show the relevant architecture in Figure 1. Like other on-path SmartNICs, FPCs are organized into islands with local memory and processing resources, akin to NUMA domains. Islands are connected in a mesh via a high-bandwidth interconnect (arrows in Figure 1).

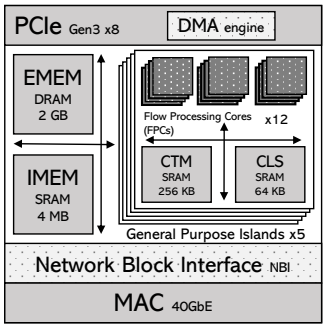

Figure 1. NFP-4000 overview.

[1] Mellanox BlueField [31] and Broadcom Stingray [4] are off-path SmartNICs that are not optimized for packet processing [26].

The *PCIe* island has up to two PCIe Gen3 x8 interfaces and a DMA engine exposing DMA transaction queues [41]. FPCs can issue up to 256 asynchronous DMA transactions to perform IO between host and NIC memory. The *MAC* island supports up to two 40 Gbps Ethernet interfaces, accessed via a *network block interface* (NBI).

Flow Processing Cores (FPCs). 60 FPCs are grouped into five general-purpose islands (each containing 12 FPCs). Each FPC is an independent 32-bit core at 800 MHz with 8 hardware threads, 32 KB instruction memory, 4 KB data memory, and CRC acceleration. While FPCs have strong data flow processing capabilities, they have small codestores, lack timers, as well as floating-point and other complex computational support, such as division. This makes them unsuitable to execute computationally and control intensive TCP functionality, such as congestion, connection, and complex retransmission control. For example, congestion avoidance involves computing an ECN-ratio (gradient). We found that it takes 1,500 cycles (1.9 μs) per RTT to perform this computation on FPCs.

Memory. The NFP-4000 includes multiple memories of various sizes and performance characteristics. General-purpose islands have 64KB of island-local scratch (*CLS*) and 256 KB of island target memory (*CTM*), with access latencies of up to 100 cycles from island-local FPCs for data processing and transfer, respectively. The internal memory unit (*IMEM*) provides 4 MB of SRAM with an access latency of up to 250 cycles. The external memory unit (*EMEM*) provides 2 GB of DRAM, fronted by a 3 MB SRAM cache, with up to 500 cycles latency.

Implications for flexible offload. The NFP-4000 supports a broad range of protocols, but the computation and memory restrictions require careful offload design. As FPCs are wimpy and memory latencies high, sequential instruction execution is much slower than on host processors. Conventional run-to-completion processing that assigns entire connections to cores [3, 17, 19] results in poor per-connection throughput and latency. In some cases, it is beyond the feasible instruction and memory footprint. Instead, an efficient offload needs to leverage more fine-grained parallelism to limit the per-core compute and memory footprint.

3 FlexTOE Design

In addition to flexibility, FlexTOE has the following goals:

- **Low tail latency and high throughput**. Modern datacenter network loads consist of short and long flows. Short flows, driven by remote procedure calls, require low tail completion time, while long flows benefit from high throughput. FlexTOE shall provide both.
- **Scalability**. The number of network flows and application contexts that servers must handle simultaneously is increasing. FlexTOE shall scale with this demand.

To achieve these goals and overcome SmartNIC hardware limitations, we propose three design principles:

1. **One-shot data-path offload.** We focus offload on the TCP RX/TX data-path, eliminating complex control, compute, and state, thereby also enabling fine-grained parallelization. Further, our data-path offload is one-shot for each TCP segment. Segments are never buffered on the NIC, vastly simplifying SmartNIC memory management.
2. **Modularity.** We decompose the TCP data-path into fine-grained, customizable modules that keep private state and communicate explicitly. New TCP extensions can be implemented as modules and hooked into the data-flow, simplifying development and integration.
3. **Fine-grained parallelism.** We organize the data-path modules into a data-parallel computation pipeline that maximizes SmartNIC resource use. We map stages to FPCs, allowing us to fully utilize all FPC resources. We employ TCP segment sequencing and reordering to support parallel, out-of-order processing of pipeline stages, while enforcing in-order segment delivery.

Decomposing TCP for offload. We use the TAS host TCP stack architecture [19] as a starting point. TAS splits TCP processing into three components: a data-path, a control-plane, and an application library. The data-path is responsible for scalable data transport of established connections: TCP segmentation, loss detection and recovery, rate control, payload transfer between socket buffers and the network, and application notifications. The control-plane handles connection and context management, congestion control, and complex recovery involving timeouts. Finally, the application library intercepts POSIX socket API calls and interacts with control-plane and data-path using dedicated context queues in shared memory. Data-path and control-plane execute in their own protection domains on dedicated cores, isolated from untrusted applications, and communicate through efficient message passing queues.

FlexTOE offload architecture. In FlexTOE we adapt this architecture for offload, by designing and integrating a *data-path running efficiently* on the SmartNIC (§3.1). The FlexTOE control-plane can run on the host or on a SmartNIC control CPU, with the same functionality as in TAS (cf. §D). The FlexTOE control-plane additionally manages the SmartNIC data-path resources. Similarly, our application library (libTOE) intercepts POSIX socket calls and is dynamically linked to unmodified processes that use FlexTOE, and communicates directly with the data-path.

Figure 2 shows the offload architecture of FlexTOE, with a host control-plane (each box is a protection domain). libTOE, data-path, and control-plane communicate via pairs of *context queues* (CTX-Qs), one for each communication direction. CTX-Qs leverage PCIe DMA and MMIO or shared memory for SmartNIC-host and intra-host communication, respectively.

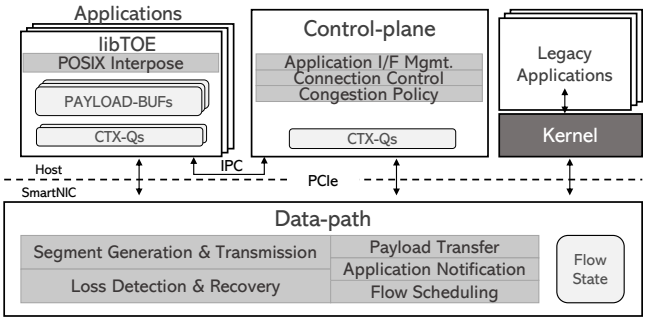

Figure 2. FlexTOE offload architecture (host control-plane).

FlexTOE supports per-thread context queues for scalability. Each TCP socket keeps receive and transmit payload buffers (PAYLOAD-BUFs) in host memory. libTOE appends data for transmission into the per-socket TX PAYLOAD-BUF and notifies the data-path using a thread-local CTX-Q. The data-path appends received segments to the socket's RX PAYLOAD-BUF after reassembly and libTOE is notified via the same thread-local CTX-Q. Non-FlexTOE traffic is forwarded to the Linux kernel, which legacy applications may use simultaneously.

3.1 TCP Data-path Parallelization

To provide high offload performance using relatively wimpy SmartNIC FPCs, FlexTOE has to leverage all available parallelism within the TCP data-path. In this section, we analyze the TAS host TCP data-path to investigate what parallelism can be extracted. In particular, the TCP data-path in TAS has the following three workflows:

- **Host control (HC)**: When an application wants to transmit data, executes control operations on a socket, or when retransmission is necessary, the data-path must update the connection's transmit and receive windows accordingly.
- **Transmit (TX)**: When a TCP connection is ready to send—based on congestion and flow control—the data-path prepares a segment for transmission, fetching its payload from a socket transmit buffer and sending it out to the MAC.
- **Receive (RX)**: For each received segment of an established connection, the data-path must perform byte-stream reassembly—advance the TCP window, determine the segment's position in the socket receive buffer, generate an acknowledgment to the sender, and, finally, notify the application. If the received segment acknowledges previously transmitted segments, the data-path must also free the relevant payload in the socket transmit buffer.

Host TCP stacks, such as Linux or TAS, typically process each workflow to completion in a critical section accessing a shared per-connection state structure. HC workflows are typically processed on the program threads that trigger them, while TX and RX are typically triggered by NIC interrupts and processed on high-priority (kernel or dedicated) threads.

For efficient offload, we decompose this data-path into an up to five-stage parallel pipeline of processing modules: *pre-processing*, *protocol*, *post-processing*, *DMA*, and *context queue*

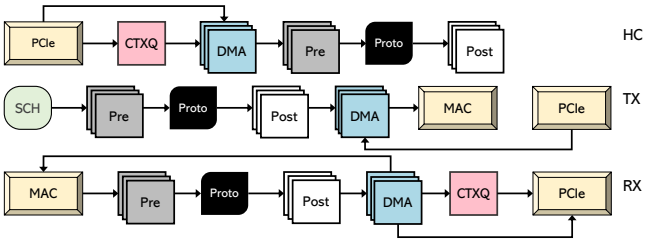

Figure 3. Per-connection data-path workflows. *Protocol* is atomic. Other stages may be replicated for parallelism.

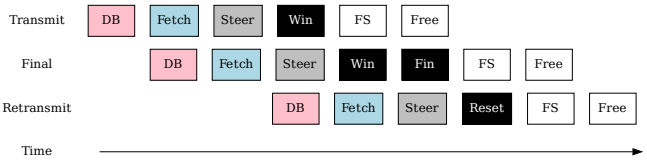

Figure 4. HC pipeline: Transmit, FIN, and retransmit.

(Figure 3). Accordingly, we partition connection state into module-local state (cf. §A). The pipeline stages are chosen to maximize data-path parallelism. Pre-processing accesses connection identifiers such as MAC and IP addresses for segment header preparation and filtering. The post-processing block handles application interface parameters, such as socket buffer addresses and context queues. These parameters are read-only after connection establishment and enable coordination-free scaling. Congestion control statistics are collected by the post-processor, but are only read by forward stages and can be updated out-of-order (updates commute). The protocol stage executes data-path code that must atomically modify protocol state, such as sequence numbers and socket buffer positions. It is the only *pipeline hazard*—it cannot execute in parallel with other stages. The DMA stage is stateless, while context queue stages may be sharded. Both conduct high-latency PCIe transactions and are thus separate stages that execute in parallel and scale independently.

We run pipeline stages on dedicated FPCs that utilize local memory for their portion of the connection state. Pipelining allows us to execute the data-path in parallel. It also allows us to replicate processing-intensive pipeline stages to scale to additional FPCs. With the exception of protocol processing, which is atomic per connection, all pipeline stages are replicated. To concurrently process multiple connections, we also replicate the entire pipeline. To keep flow state local, each pipeline handles a fixed *flow-group*, determined by a hash on the flow's 4-tuple (the flow's protocol type is ignored—it must be TCP). We now describe how we parallelize each data-path workflow by decomposing it into these pipeline stages.

3.1.1 Host Control (HC). HC processing is triggered by a PCIe doorbell (DB) sent via memory-mapped IO (MMIO) by the host to the context queue stage. Figure 4 shows the HC pipeline for two transmits (the second transmit closes the connection) triggered by libTOE, and a retransmit triggered by the control-plane. HC requests may be batched.

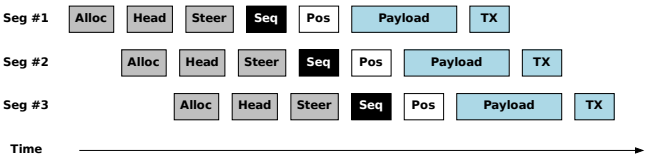

Figure 5. TX pipeline sending 3 segments.

The context queue stage polls for DBs. In response to a DB, the stage allocates a descriptor buffer from a pool in NIC memory. The limited pool size flow-controls host interactions. If allocation fails, processing stops and is retried later. Otherwise, the DMA stage fetches the descriptor from the host context queue into the buffer (Fetch). The pre-processor reads the descriptor, determines the flow-group, and routes to the appropriate protocol stage (Steer). The protocol stage updates connection receive and transmit windows (Win). If the HC descriptor contains a connection-close indication, the protocol stage also marks the connection as FIN (Fin). When the transmit window expands due to the application sending data for transmission, the post-processor updates the flow scheduler (FS) and returns the descriptor to the pool (Free).

Retransmissions in response to timeouts are triggered by the control-plane and processed the same as other HC events (fast retransmits due to duplicate ACKs are described in §3.1.3). The protocol stage resets the transmission state (Reset) to the last ACKed sequence number (go-back-N retransmission).

3.1.2 Transmit (TX).

Transmission is triggered by the flow scheduler (SCH) when a connection can send segments. Figure 5 shows the TX pipeline for 3 example segments.

The pre-processor allocates a segment in NIC memory (Alloc), prepares Ethernet and IP headers (Head), and steers the segment to the flow-group's protocol stage (Steer). The protocol stage assigns a TCP sequence number based on connection state and determines the transmit offset in the host socket transmit buffer (Seq). The post-processor determines the socket transmit buffer address in host memory (Pos). The DMA stage fetches the host payload into the segment (Payload). After DMA completes, it issues the segment to the NBI (TX), which transmits and frees it.

3.1.3 Receive (RX).

Figure 6 shows the RX pipeline for 3 example segments, where segment #3 arrives out of order.

Pre-processing. The pre-processor first validates the segment header (Val). Non-data-path segments[2] are filtered and forwarded to the control-plane. Otherwise, the pre-processor determines the connection index based on the segment's 4-tuple (Id) that is used by later stages to access connection state. The pre-processor generates a *header summary* (Sum), including only relevant header fields required by later pipeline stages and steers the summary and connection identifier to the protocol stage of its flow-group (Steer).

[2] *Data-path segments* have any of the ACK, FIN, PSH, ECE, and CWR flags and they may have the timestamp option.

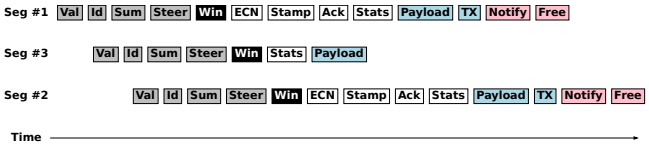

Figure 6. RX pipeline receiving 3 segments, 1 out of order.

Protocol. Based on the header summary, the protocol stage updates the connection's sequence and acknowledgment numbers, the transmit window, and determines the segment's position in the host socket receive payload buffer, trimming the payload to fit the receive window if necessary (Win). The protocol stage also tracks duplicate ACKs and triggers fast retransmissions if necessary, by resetting the transmission state to the last acknowledged position. Finally, it forwards a snapshot of relevant connection state to post-processing.

Out-of-order arrivals (segment #3 in Figure 6) need special treatment. Like TAS [19], we track one out-of-order interval in the receive window, allowing the protocol stage to perform reassembly directly within the host socket receive buffer. We merge out-of-order segments within the interval in the host receive buffer. Segments outside of the interval are dropped and generate acknowledgments with the expected sequence number to trigger retransmissions at the sender. This design performs well under loss (cf. §5.3).

Post-processing. The post-processor prepares an acknowledgment segment (Ack). FlexTOE provides explicit congestion notification (ECN) feedback and accurate timestamps for RTT estimation (Stamp) in acknowledgments. It also collects congestion control and transmit window statistics, which it sends to the control-plane and flow scheduler (Stats). Finally, it determines the physical address of the host socket receive buffer, payload offset, and length for the DMA stage. If libTOE is to be notified, the post-processor allocates a context queue descriptor with the appropriate notification.

DMA. The DMA stage first enqueues payload DMA descriptors to the PCIe block (Payload). After payload DMA completes, the DMA stage forwards the notification descriptor to the context queue stage. Simultaneously, it sends the prepared acknowledgment segment to the NBI (TX), which frees it after transmission. This ordering is necessary to prevent the host and the peer from receiving notifications before the data transfer to the host socket receive buffer is complete.

Context queue. If necessary, the context queue stage allocates an entry on the context queue and issues the context queue descriptor DMA to notify libTOE of new payload (Notify) and frees the internal descriptor buffer (Free).

3.2 Sequencing and Reordering

TCP requires that segments of the same connection are processed in-order for receiver loss detection. However, stages in FlexTOE's data-parallel processing pipeline can have varying processing time and hence may reorder segments. Figure 7

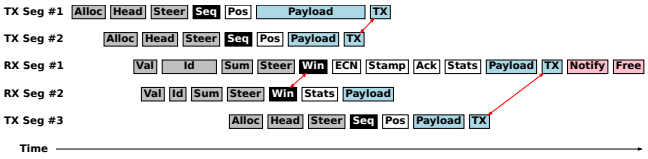

Figure 7. Undesirable pipeline reordering (red arrows).

shows three examples on a bidirectional connection where undesirable segment reordering occurs.

1. **TX.** TX segment #1 stalls in DMA across a congested PCIe link, causing it to be transmitted on the network after TX segment #2, potentially triggering receiver loss detection.
2. **RX.** RX segment #1 stalls in flow identification during pre-processing, entering the protocol stage later than RX segment #2. The protocol stage detects a hole and triggers unnecessary out-of-order processing.
3. **ACK.** TX segment #3 is processed after RX segment #1 in the protocol stage. RX segment #1 generates an ACK, but RX post-processing is complex, resulting in TX segment #3 with a higher sequence number being sent before ACK segment #1.

To avoid reordering, FlexTOE's data-path pipeline sequences and reorders segments if necessary. In particular, we assign a sequence number to each segment entering the pipeline. The parallel pipeline stages can operate on each segment in any order. The protocol stage requires in-order processing and we buffer and re-order segments that arrive out-of-order before admitting them to the protocol stage. Similarly, we buffer and re-order segments for transmission before admitting them to the NBI. We leverage additional FPCs for sequencing, buffering, and reordering.

3.3 Flexibility

Data center networks evolve quickly, requiring TCP stacks to be easily modifiable by operators, not just vendors [29, 42, 43]. Many desirable data center features require TOE modification and are adapted frequently by operators. FlexTOE provides flexibility necessary to implement and maintain these features even beyond host stacks such as TAS, by relying on a programmable SmartNIC. To simplify development and modification of the TCP data-path, FlexTOE provides an extensible, data-parallel pipeline of self-contained modules, similar to the Click [38] extensible router.

Module API. The FlexTOE module API provides developers one-shot access to TCP segments and associated meta-data. Meta-data may be created and forwarded along the pipeline by any module. Modules may also keep private state. For scalability, private state cannot be accessed by other modules or replicas of the same module. Instead, state that may be accessed by further pipeline stages is forwarded as meta-data.

The replication factor of pipeline stages and assignment to FPCs is manual and static in FlexTOE. As long as enough FPCs are available, this approach is acceptable. Operators can determine an appropriate replication factor that yields acceptable TCP processing bandwidth for a pipeline stage via throughput microbenchmarks at deployment. Stages that modify connection state atomically may be deployed by inserting an appropriate steering stage that steers segments of a connection to the module in the atomic stage, holding their state (cf. protocol processing stage in §3.1).

XDP modules. FlexTOE also supports eXpress Data Path (XDP) modules [14–16], implemented in eBPF. XDP modules operate on raw packets, modify them if necessary, and output one of the following result codes: (i) `XDP_PASS`: Forward the packet to the next FlexTOE pipeline stage. (ii) `XDP_DROP`: Drop the packet. (iii) `XDP_TX`: Send the packet out the MAC. (iv) `XDP_REDIRECT`: Redirect the packet to the control-plane.

XDP modules may use BPF maps (arrays, hash tables) to store and modify state atomically [25], which may be modified by the control-plane. For example, a firewall module may store blacklisted IPs in a hash map and the control-plane may add or remove entries dynamically. The module can consult the hash map to determine if a packet is blacklisted and drop it. XDP stages scale like other pipeline stages, by replicating the module. FlexTOE automatically reorders processed segments after a parallel XDP stage (§3.2).

Using these APIs, we modified the FlexTOE data-path many times, implementing the features listed in §2.1 (evaluation in §5.1). Further, ECN feedback and segment timestamping (cf. §3.1.3) are optional TCP features that support our congestion control policies. Operators can remove the associated post-processing modules if they are not needed.

By handling atomicity, parallelization, and ordering concerns, FlexTOE allows complex offloads to be expressed using few lines of code. For example, we implement AccelTCP's connection splicing in 24 lines of eBPF code (cf. Listing 1 in the appendix). The module performs a lookup on the segment 4-tuple in a BPF hashmap. If a match is not found, we forward the segment to the next pipeline stage. Otherwise, we modify the destination MAC and IP addresses, TCP ports, and translate sequence and acknowledgment numbers using offsets configured by the control-plane, based on the connection's initial sequence number. Finally, we transmit. FlexTOE handles sequencing and updating the checksum of the segment. Additionally, when we receive segments with control flags indicating connection closure, we atomically remove the hashmap entry and notify the control-plane.

3.4 Flow Scheduling

FlexTOE leverages a work-conserving flow scheduler on the NIC data-path. The flow scheduler obeys transmission rate-limits and windows configured by the control-plane's congestion control policy. For each connection, the flow scheduler keeps track of how much data is available for transmission and the configured rate. Transmission rates and windows

are stored in NIC memory and are directly updated by the control-plane using MMIO.

We implement our flow scheduler based on Carousel [53]. Carousel schedules a large number of flows using a time wheel. Based on the next transmission time, as computed from rate limits and windows, we enqueue flows into corresponding slots in the time wheel. As the time slot deadline passes, the flow scheduler schedules each flow in the slot for transmission (§3.1.2). To conserve work, the flow scheduler only adds flows with a non-zero transmit window into the time wheel and bypasses the rate limiter for uncongested flows. These flows are scheduled round-robin.

4 Agilio-CX40 Implementation

This section describes FlexTOE's Agilio-CX40 implementation. Due to space constraints, the x86 and BlueField ports are described in detail in §E. FlexTOE's design across the different ports is identical. We do not merge or split any of the fine-grained modules or reorganize the pipeline across ports.

FlexTOE is implemented in 18,008 lines of C code (LoC). The offloaded data-path comprises 5,801 lines of C code. We implement parts of the data-path in assembly for performance. libTOE contains 4,620 lines of C, whereas the control path contains 5,549 lines of C. libTOE and the control plane are adapted from TAS. We use the NFP compiler toolchain version 6.1.0.1 for SmartNIC development.

Driver. We develop a Linux FlexTOE driver based on the `igb_uio` driver that enables libTOE and the control plane to perform MMIO to the SmartNIC from user space. The driver supports MSI-X based interrupts. The control-plane registers an `eventfd` for each application context in the driver. The interrupt handler in the driver pings the corresponding `eventfd` when an interrupt is received from the data-path for the application context. This enables libTOE to sleep when waiting for IO and reduces the host CPU overhead of polling.

Host memory mapping. To simplify virtual to physical address translation for DMA operations, we allocate physically contiguous host memory using 1G hugepages. The control-plane maps a pool of 1G hugepages at startup and allocates socket buffers and context queues out of this pool. In the future, we can use the IOMMU to eliminate the requirement of physically contiguous memory for FlexTOE buffers.

Context queues. Context queues use shared memory on the host, but communication between SmartNIC and host requires PCIe. We use scalable and efficient PCIe communication techniques [44] that poll on host memory locations when executing in the host and on NIC-internal memory when executing on the NIC. The NIC is notified of new queue entries via MMIO to a NIC doorbell. The context queue manager notifies applications through MSI-X interrupts, converted by the driver to an `eventfd`, after a queue has been inactive.

4.1 Near-memory Processing

An order of magnitude difference exists in the access latencies of different memory levels of the NFP-4000. For performance, it is critical to maximize access to local memory. The NFP-4000 also provides certain near-memory acceleration, including a lookup engine exposing a content addressable memory (CAM) and a hash table for fast matching, a queue memory engine exposing concurrent data structures such as linked lists, ring buffers, journals, and work-stealing queues. Finally, synchronization primitives such as ticket locks and inter-FPC signaling are exposed to coordinate threads and to sequence packets. We build specialized caches at multiple levels in the different pipeline stages using these primitives. Other NICs have similar accelerators.

Caching. We use each FPC's CAM to build 16-entry fully-associative local memory caches that evict entries based on LRU. The protocol stage adds a 512-entry direct-mapped second-level cache in CLS. Across four islands, we can accommodate up to 2K flows in this cache. The final level of memory is in EMEM. When an FPC processes a segment, it fetches the relevant state into its local memory either from CLS or from EMEM, evicting other cache entries as necessary. We allocate connection identifiers in such a way that we minimize collisions on the direct-mapped CLS cache.

Active connection database. To facilitate connection index lookup in the pre-processing stage, we employ the hardware lookup capability of IMEM to maintain a database of active connections. CAM is used to resolve hash collisions. The pre-processor computes a CRC-32 hash on a segment's 4-tuple to locate the connection index using the lookup engine. The pre-processor caches up to 128 lookup entries in its local memory via a direct-mapped cache on the hash value.

FPC mapping. FlexTOE's pipeline fully leverages the Agilio CX40 and is extensible to further FPCs, e.g. of the Agilio LX [40]. For island-local interactions among modules, we use CLS ring buffers. CLS supports the fastest intra-island producer-consumer mechanisms. Among islands, we rely on work-queues in IMEM and EMEM.

We use all but one general-purpose islands for the first three stages of the data-path pipeline (*protocol islands*). Each island manages a *flow-group*. While protocol and post-processing FPCs are local to a flow-group, pre-processors handle segments for any flow. We assign 4 FPCs to pre-/post-processing stages in each flow-group. Each island retains 3 unassigned FPCs that can run additional data-path modules (§5.1).

On the remaining general-purpose island (called *service island*), we host remaining pipeline stages and adjacent modules, such as context queue FPCs, the flow scheduler (SCH), and DMA managers. DMA managers are replicated to hide PCIe latencies. The number of FPCs assigned to each functionality is determined such that no functionality may become a

bottleneck. Sequencing and reordering FPCs are located on a further island with miscellaneous functionality.

Flow scheduler. We implement Carousel using hardware queues in EMEM. Each slot is allocated a hardware queue. To add a flow to the time wheel, we enqueue it on the queue associated with the time slot. Note that the order of flows within a particular slot is not preserved. EMEM support for a large number of hardware queues enables us to efficiently implement a time wheel with a small slot granularity and large horizon to achieve high-fidelity congestion control. Converting transmission rates to deadlines requires division, which is not supported on the NFP-4000. Thus, the control-plane computes transmission intervals in cycles/byte units from rates and programs them to NIC memory. This enables the flow scheduler to compute the time slot using only multiplication.

5 Evaluation

We answer the following evaluation questions:

- **Flexible offload.** Can flexible offload improve throughput, latency, and scalability of data center applications? Can we implement common data center features? (§5.1)
- **RPCs.** How does FlexTOE's data-path parallelism enable TCP offload for demanding RPCs? Do these benefits generalize across hardware architectures? Does FlexTOE provide low latency for short RPCs? Does FlexTOE provide high throughput for long RPCs? To how many simultaneous connections can FlexTOE scale? (§5.2)
- **Robustness.** How does FlexTOE perform under loss and congestion? Does it provide connection-fairness? (§5.3)

Testbed cluster. Our evaluation setup consists of two 20-core Intel Xeon Gold 6138 @ 2 GHz machines, with 40 GB RAM and 48 MB aggregate cache. Both machines are equipped with Netronome Agilio CX40 40 Gbps (single port), Chelsio Terminator T62100-LP-CR 100 Gbps and Intel XL710 40 Gbps NICs. We use one of the machines as a server, the other as a client. As additional clients, we also use two 2×18-core Intel Xeon Gold 6154 @ 3 GHz systems with 90 MB aggregate cache and two 4-core Intel Xeon E3-1230 v5 @ 3.4 GHz systems with 9 MB aggregate cache. The Xeon Gold machines are equipped with Mellanox ConnectX-5 MT27800 100 Gbps NICs, whereas the Xeon E3 machines have 82599ES 10 Gbps NICs. The machines are connected to a 100 Gbps Ethernet switch.

Baseline. We compare FlexTOE performance against the Linux TCP stack, Chelsio's kernel-based TOE[3], and the TAS kernel-bypass stack[4]. TAS does not perform well with the Agilio CX40 due to a slow NIC DPDK driver. We run TAS on the Intel XL710 NIC, as in [19], unless mentioned otherwise. We use identical application binaries across all baselines. DCTCP is our default congestion control policy.

[3]Chelsio does not support kernel-bypass.

[4]TAS [19] performs better than mTCP [17] on all of our benchmarks. Hence, we omit a comparison to mTCP and AccelTCP [37], which uses mTCP.

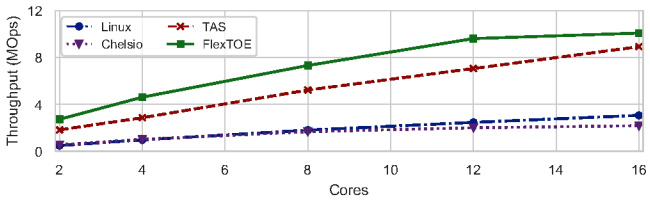

Figure 8. Memcached throughput scalability.

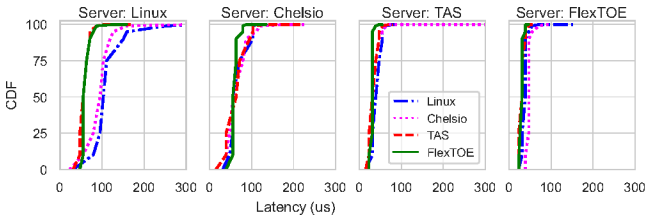

Figure 9. Latency of different server-client combinations.

5.1 Benefit of Flexible Offload

Application throughput scalability. Offloaded CPU cycles may be used for application work. We quantify these benefits by running a Memcached server, as in §2.1, varying the number of server cores. Figure 8 shows that, by saving host CPU cycles (cf. Table 1), FlexTOE achieves up to 1.6× TAS, 4.9× Chelsio, and 5.5× Linux throughput. FlexTOE and TAS scale similarly—both use per-core context queues. The Agilio CX becomes a compute-bottleneck at 12 host cores. Linux and Chelsio are slow for this workload, due to system call overheads, and do not scale well due to in-kernel locks.

Low (tail) latency. We repeat a single-threaded version of the same Memcached benchmark for all server-client network stack combinations. Latency distributions are shown in Figure 9. We can see that FlexTOE consistently provides the lowest median and tail Memcached operation latency across all stack combinations. Offload provides excellent performance isolation by physically separating the TCP data-path, even though FlexTOE's pipelining increases minimum latency in some cases (cf. §5.2).

Flexibility. Unlike fixed offloads and in-kernel stacks, FlexTOE provides full user-space programmability via a module API, simplifying development. Customizing FlexTOE is simple and does not require a system reboot. For example, we have developed logging, statistics, and profiling capabilities that can be turned on only when necessary. We make use of these capabilities during development and optimization of FlexTOE. We implemented up to 48 different tracepoints (including examples from bpftrace [13]) in the data-path pipeline, tracking transport events such as per-connection drops, out-of-order packets and retransmissions, inter-module queue occupancies, and critical section lengths in the protocol module for various event types. Table 2 shows that profiling degrades data-path performance versus the baseline by up to 24% when all 48 tracepoints are enabled. We also implement `tcpdump`-style traffic logging, including packet filters based on header

Build	Throughput (MOps)
Baseline FlexTOE	11.35
Statistics and profiling	8.67
`tcpdump` (no filter)	6.52
`XDP` (null)	10.87
`XDP` (vlan-strip)	10.83

Table 2. Performance with flexible extensions.

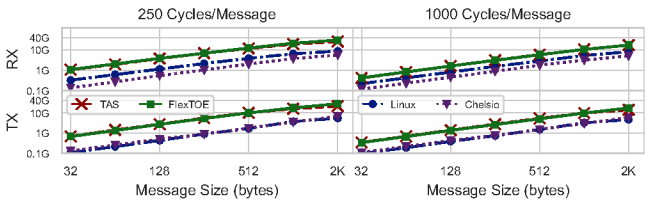

Figure 10. RPC throughput for saturated server.

fields. Logging naturally has high overhead (up to 43% when logging all packets). FlexTOE provides the flexibility to implement these features and to turn them on only when necessary.

Furthermore, new data-plane functionality leveraging the XDP API may be dynamically loaded into FlexTOE as eBPF programs. eBPF programs can be compiled to NFP assembly. This level of dynamic flexibility is hard to achieve with an FPGA as it requires instruction set programmability (overlays [52]). We measure the overhead of FlexTOE XDP support by running a null program that simply passes on every packet without modification. We observe only 4% decline in throughput. Common XDP modules, such as stripping VLAN tags on ingress packets, also have negligible overhead. Finally, connection splicing (cf. Listing 1 in the appendix) achieves a maximum splicing performance of 6.4 million packets per second, enough to saturate the NIC line rate with MTU-sized packets, leveraging only idle FPCs[5].

5.2 Remote Procedure Calls (RPCs)

RPCs are an important but difficult workload for flexible offload. Latency and client scalability requirements favor fast processing engines with large caches, such as found in CPUs and ASICs. Neither are available in on-path SmartNICs. We show that flexible offload can be competitive with state-of-the-art designs. We then show that FlexTOE's data-path parallelism is necessary to provide the necessary performance.

Typical RX / TX performance. We start with a typical server scenario, processing RPCs of many (128) connections, produced in an open loop by multiple (16) clients (multiple pipelined RPCs per connection). To simulate application processing, our server waits for an artificial delay of 250 or 1,000 cycles for each RPC. We run single-threaded to avoid the network being a bottleneck. We quantify RX and TX throughput separately, by switching RPC consumer and producer roles among clients and servers, over different RPC sizes.

[5] We are compute-limited by our Agilio CX. Using an Agilio LX, like AccelTCP, would allow us to achieve even higher throughput.

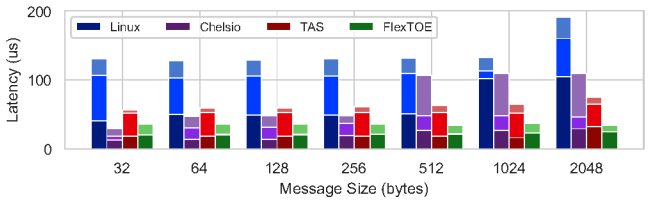

Figure 11. Median, 99p and 99.99p RPC RTT.

Figure 10 shows the results. For 250 cycles of processing overhead, FlexTOE provides up to 4× better throughput than Linux and 5.3× better throughput than Chelsio when receiving. For 2 KB message size, both TAS and FlexTOE reach 40 Gbps line rate, whereas Linux and Chelsio barely reach 10 Gbps and 7 Gbps, respectively. When sending packets, the difference in performance between Linux and FlexTOE is starker. FlexTOE shows over 7.6× higher throughput over both Linux and Chelsio for all message sizes. The gains remain at over 2.2× as we go to 1,000 cycles/RPC. Performance of TAS and FlexTOE track closely for all message sizes. This is expected as the single application server core is saturated by both network stacks (TAS runs on additional host cores).

We break down this result by studying the performance sensitivity of each TCP stack, varying each RPC parameter within its sensitive dynamic range. For these benchmarks, we evaluate the raw performance of the stacks, without application processing delays.

RPC latency. A client establishes a single connection to the server and measures single RPC RTT. Figure 11 shows the median and tail RTT for various small message sizes (stacked bars). The inefficiency of in-kernel networking is reflected in the median latency of Linux, which is at least 5× worse compared to other stacks. For message sizes < 256 B, FlexTOE's median latency (20 us) is 1.4× Chelsio's median latency (14 us) and 1.25× TAS's median latency (16 us). FlexTOE's data-path pipeline across many wimpy FPCs increases median latency for single RPCs. However, FlexTOE has an up to 3.2× smaller tail compared to Chelsio and nearly constant per-segment overhead as the RPC size increases. In case of a 2 KB RPC (larger than the TCP maximum segment size), FlexTOE's latency distribution remains nearly unchanged. FlexTOE's fine-grain parallelism is able to hide the processing overhead of multiple segments, providing 22% lower median and 50% lower tail latency than TAS.

Per-connection throughput. In this setup, a client transfers a large RPC message to the server. In the first case (Figure 12a), the server responds with a 32 B response whereas in the second case (b), the server echoes the message back to the client (TAS performance is unstable with messages > 2 MB in this case—we omit these results). In the short-response case, Chelsio performs 20% better than the other stacks—Chelsio is a 100 Gbps NIC optimized for unidirectional streaming. However, it has 20% lower throughput as compared to FlexTOE in

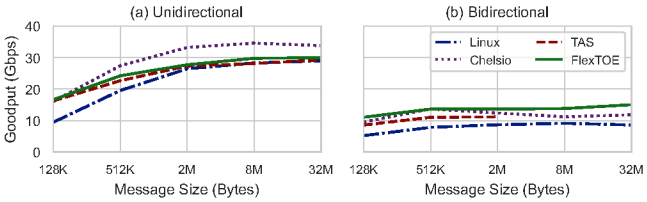

Figure 12. Large RPC throughput with varying RPC size.

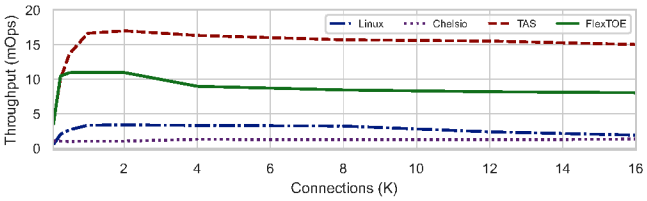

Figure 13. Connection scalability benchmark.

the echo case. Other stacks cannot parallelize per-connection processing, leading to limited throughput[6], while FlexTOE's throughput is limited by its protocol stage. FlexTOE currently acknowledges every incoming packet. For bidirectional flows, this quadruples the number of packets processed per second. Implementing delayed ACKs would improve FlexTOE's performance further for large flows.

Connection scalability. We establish an increasing number of RPC client connections from all 5 client machines to a multi-threaded echo server. To stress TCP processing, each connection leaves a single 64 B RPC in-flight. Figure 13 shows the throughput as we vary the number of connections. This workload is very challenging for FlexTOE as it exhausts fast memory and prevents per-connection batching, causing a cache miss at every pipeline stage for every segment. Up to 2K connections, FlexTOE shows a throughput of 3.3× Linux. TAS performs 1.5× better than FlexTOE for this workload. FlexTOE is compute-bottlenecked[7] at the protocol stage, which uses 8 FPCs in this benchmark. Agilio CX caches 2K connections in CLS memory. Beyond this, the protocol stage must move state among local memory, CLS, and EMEM. EMEM's SRAM cache is increasingly strained as the number of connections increases. FlexTOE's throughput declines by 24% as we hit 8k connections and plateaus beyond that[8]. TAS's fast-path exhibits better connection scalability, as it has access to the larger host CPU cache, while Linux's throughput declines significantly. Chelsio has poor performance for this workload, as `epoll()` overhead dominates.

Benefit of data-path parallelism. To break down the impact of FlexTOE's data-parallel design on RPC performance,

[6]With multiple unidirectional flows, all stacks achieve line rate (Figure 15b).
[7]We expect that running FlexTOE on the Agilio LX with 1.2 GHz FPCs—1.5× faster than Agilio CX—would boost the peak throughput to match TAS performance. Agilio LX also doubles the number of FPCs and islands. It would allow us to exploit more parallelism and cache more connections.
[8]While we evaluate up to 16K connections, FlexTOE can leverage the 2 GB on-board DRAM to scale to 1M+ connections.

Design	Throughput (Mbps)	×	Latency (us) 50p	99.99p
Baseline	79.32	1	1,179	6,929
+ Pipelining	3,640.49	46	183	684
+ Intra-FPC parallelism	8,194.34	103	128	148
+ Replicated pre/post	11,086.93	140	94	106
+ Flow-group islands	22,684.69	286	46	58

Table 3. FlexTOE data-path parallelism breakdown.

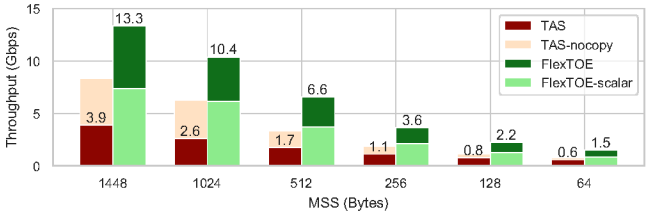

Figure 14. FlexTOE benefits on BlueField SmartNIC.

we repeat the echo benchmark with 64 connections, with each connection leaving a single 2 KB RPC in-flight (to be able to evaluate both intra and inter connection parallelism). Table 3 shows the performance impact as we progressively add data-path parallelism. Our baseline runs the entire TCP processing to completion on the SmartNIC before processing the next segment. Pipelining improves performance by 46× over the baseline. As we enable 8 threads on the FPCs (2.25× gain), we hide the latency of memory operations and improve FPC utilization. Next, we replicate the pre-processing and post-processing stages, leveraging sequencing and reordering for correctness, to extract 1.35× improvement and finally, with four flow-group islands, we see a further 2× improvement. We can see that each level of data-path parallelism is necessary, improving RPC throughput and latency by up to 286×.

Do these benefits generalize? We investigate whether data-path parallelism provides benefits across platforms. In particular, we investigate single connection throughput of pipelined RPCs across a range of maximum segment sizes (MSS) on a Mellanox BlueField [31] MBF1M332A-ASCAT 25 Gbps SmartNIC and on a 32-core AMD 7452 @ 2.35 GHz host with 128 GB RAM, 148 MB aggregate cache, and a conventional 100 Gbps ConnectX-5 NIC. We use a single-threaded RPC sink application, running on the same platform[9]. We compare TAS's core-per-connection processing to FlexTOE's data-parallelism. We replicate each of FlexTOE's pre and post processing stages 2×, resulting in 9 FlexTOE cores. Further gains may be achievable by more replication. To break down FlexTOE's benefits, we also compare to a FlexTOE pipeline without replicated stages (FlexTOE-scalar), using 7 cores.

Figure 14 shows BlueField results. FlexTOE outperforms TAS by up to 4× on BlueField (and 2.4× on x86). Depending on RPC size, FlexTOE accelerates different stages of the TCP data path. For large RPCs, FlexTOE accelerates data copy to

[9]BlueField is an off-path SmartNIC that is not optimized for packet processing offload to host-side applications (§2.3).

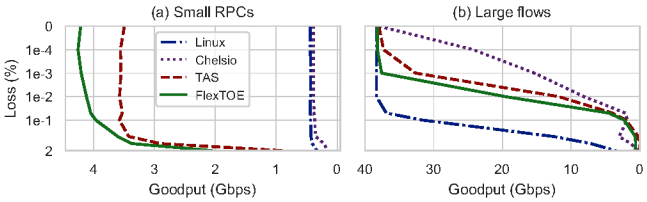

Figure 15. Throughput, varying packet loss rate.

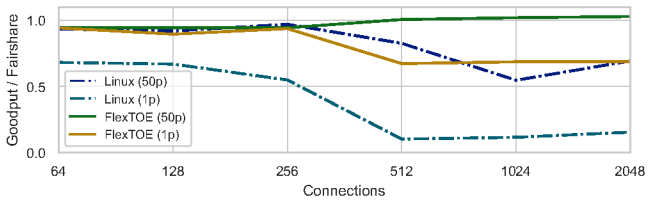

Figure 16. Throughput distribution at line rate.

socket payload buffers. To show this, we eliminate the step in TAS (TAS-nocopy), allowing TAS to perform at 0.5× FlexTOE on BlueField (and identical to FlexTOE on x86). For smaller RPCs, TAS-nocopy benefits diminish and FlexTOE supports processing higher packet rates. FlexTOE-scalar achieves only up to 2.3× speedup over TAS on BlueField (and 1.47× on x86), showing that only part of the benefit comes from pipelining. Finally, FlexTOE speedup is greater on the wimpier BlueField, resembling our target architecture (§2.3), than on x86. To save powerful x86 cores, some stages may be collapsed, even dynamically (cf. Snap [29]), at little performance cost.

5.3 Robustness

Packet loss. We artificially induce packet losses in the network by randomly dropping packets at the switch with a fixed probability. We measure the throughput between two machines for 100 flows running 64 B echo-benchmark as we vary the loss probability, shown in Figure 15a. We configure the clients to pipeline up to 8 requests on each connection to trigger out-of-order processing when packets are lost. FlexTOE's throughput at 2% losses is at least twice as good as TAS and an order of magnitude better than the other stacks for this case. We repeat the unidirectional large RPC benchmark with 8 connections and measure the throughput as we increase the packet loss rate. For this case (b), Chelsio has a very steep decline in throughput even with 10^{-4}% loss probability. Linux is able to withstand higher loss rates as it implements more sophisticated reassembly and recovery algorithms, including selective acknowledgments—FlexTOE and TAS implement single out-of-order interval tracking on the receiver-side and go-back-n recovery on the sender. FlexTOE's behavior under loss is still better than TAS. FlexTOE processes acknowledgments on the NIC, triggering retransmissions sooner, and its predictable latency, even under load, helps FlexTOE recover faster from packet loss. We note that RDMA tolerates up to 0.1% losses [35], while eRPC falters at 0.01% loss rate [18]. Unlike FlexTOE, RDMA discards all out-of-order packets on the receiver side [35]. TAS [19] provides further evaluation of the benefits of receiver out-of-order interval tracking.

Fairness. To show scalability of FlexTOE's SCH (§3.4), we measure the distribution of connection throughputs of bulk flows between two nodes at line rate for 60 seconds. Figure 16 shows the median and 1st percentile throughput of FlexTOE and Linux as we vary the number of connections. For FlexTOE, the median closely tracks the fair share throughput and the tail is 0.67× of the median. Linux's fairness is significantly affected beyond 256 connections. Jain's fairness index (JFI) drops to 0.36 at 2K connections for Linux, while FlexTOE achieves 0.98. Above 1K connections, Linux' median throughput is worse than FlexTOE's 1st percentile.

deg.	# con.	Tpt. (G)		Lat. 99.99p (ms)		JFI	
		on	off	on	off	on	off
4	16	9.51	9.47	5.98	11.58	0.98	0.95
4	64	9.51	9.23	10.75	44.39	0.96	0.73
4	128	9.48	8.96	13.74	64.25	0.99	0.53
10	10	3.66	1.04	2.50	18.26	0.95	0.78
20	20	1.76	0.36	7.35	138.32	0.95	0.46

Table 4. FlexTOE congestion control under incast.

Incast. We simulate incast by enabling traffic shaping on the switch to restrict port bandwidth to various incast degrees and we configure WRED to perform tail drops when the switch buffer is exhausted. In this experiment, the client transfers 64 KB RPCs and the server responds with a 32 B response on each connection. As shown in Table 4, control-plane-driven congestion control in FlexTOE is able to achieve the shaped line rate, maintain low tail latency, and ensure fairness among flows under congestion. Disabling it causes excessive drops, inflating tail latency by 18.8× and skewing fairness by 2×.

6 Conclusion

FlexTOE is a flexible, yet high-performance TCP offload engine to SmartNICs. FlexTOE leverages fine-grained parallelization of the TCP data-path and segment reordering for high performance on wimpy SmartNIC architecture, while remaining flexible via a modular design. We compare FlexTOE to Linux, the TAS software TCP accelerator, and the Chelsio Terminator TOE. We find that Memcached scales up to 38% better on FlexTOE versus TAS, while saving up to 81% host CPU cycles versus Chelsio. FlexTOE provides competitive performance for RPCs, even with wimpy SmartNICs, and is robust under adverse operating conditions. FlexTOE's API supports XDP programs written in eBPF. It allows us to implement popular data center transport features, such as TCP tracing, packet filtering and capture, VLAN stripping, flow classification, firewalling, and connection splicing.

Acknowledgments. We thank the anonymous reviewers and our shepherd, Brent Stephens, for their helpful comments and feedback. This work was supported by NSF grant 1751231.

References

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

[2] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed NICs. In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation*, NSDI '20, pages 93–110, USA, 2020. USENIX Association.

[3] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pages 49–65, USA, 2014. USENIX Association.

[4] Broadcom. Broadcom Stingray SmartNICs. https://www.broadcom.com/products/ethernet-connectivity/smartnic/ps225, 2018.

[5] Cavium. Cavium OCTEON Development Kits. https://cavium.com/octeon-software-develop-kit.html, 2018.

[6] Chelsio Communications. T6 ASIC: High performance, dual port unified wire 1/10/25/40/50/100Gb Ethernet controller. https://www.chelsio.com/wp-content/uploads/resources/Chelsio-Terminator-6-Brief.pdf, 2017.

[7] Andy Currid. TCP offload to the rescue: Getting a toehold on TCP offload engines—and why we need them. *Queue*, 2(3):58–65, May 2004.

[8] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, SOSP '09, pages 15–28, New York, NY, USA, 2009. Association for Computing Machinery.

[9] 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. Azure accelerated networking: SmartNICs in the public cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI '18, pages 51–64, USA, 2018. USENIX Association.

[10] Michael Galles and Francis Matus. Pensando distributed services architecture. *IEEE Micro*, 41(2):43–49, 2021.

[11] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity ethernet at scale. In *Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '16, pages 202–215, New York, NY, USA, 2016. Association for Computing Machinery.

[12] Intel Corporation. Intel 82599 10 GbE controller datasheet. Revision 3.4, November 2019. https://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html.

[13] IO Visor Project, Linux Foundation. bpftace: High-level tracing language for Linux eBPF. https://github.com/iovisor/bpftrace, 2021.

[14] IO Visor Project, Linux Foundation. XDP: express data path. https://www.iovisor.org/technology/xdp, 2021.

[15] Jakub Kicinski and Nicolaas Viljoen, Netronome Systems. ebpf hardware offload to smartnics: cls bpf and xdp. https://www.netronome.com/media/documents/eBPF_HW_OFFLOAD_HNiMne8_2_.pdf, 2021.

[16] Jakub Kicinski and Nicolaas Viljoen, Netronome Systems. Xdp hardware offload: Current work, debugging and edge cases. https://www.netronome.com/media/documents/viljoen-xdpoffload-talk_2.pdf, 2021.

[17] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: A highly scalable user-level TCP stack for multicore systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI '14, pages 489–502, USA, 2014. USENIX Association.

[18] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter RPCs can be general and fast. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI '19, pages 1–16, USA, 2019. USENIX Association.

[19] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC offload of a distributed file system with pipeline parallelism. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles*, SOSP '21, pages 756–771, New York, NY, USA, 2021. Association for Computing Machinery.

[21] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making rpcs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference*, USENIX ATC '19, pages 863–879, USA, 2019. USENIX Association.

[22] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: Efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 36–51, New York, NY, USA, 2021. Association for Computing Machinery.

[23] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '16, pages 1–14, New York, NY, USA, 2016. Association for Computing Machinery.

[24] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. 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*, ASPLOS '16, pages 339–352, New York, NY, USA, 2016. Association for Computing Machinery.

[25] Linux. bpf(2) — linux manual page. https://man7.org/linux/man-pages/man2/bpf.2.html, 2021.

[26] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto SmartNICs using IPipe. In *Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-efficient microservices on SmartNIC-accelerated servers. In *Proceedings of the 2019 USENIX Annual Technical Conference*, USENIX ATC '19, pages 363–378, USA, 2019. USENIX Association.

[28] David A. Maltz and Pravin Bhagwat. TCP splice application layer proxy performance. *Journal of High Speed Networks*, 8(3):225–240, January 2000.

[29] 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. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[30] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 USENIX Winter Conference*, USENIX '93, page 2, USA, 1993. USENIX Association.

[31] Mellanox. Mellanox BlueField Platforms. http://www.mellanox.com/related-docs/npu-multicore-processors/PB_BlueField_Ref_Platform.pdf, 2018.

[32] memcached. Memcached, 2020. https://memcached.org/.

[33] Microsoft. Information about the TCP Chimney offload, receive side scaling, and network direct memory access features in Windows Server 2008. https://docs.microsoft.com/en-US/troubleshoot/windows-server/networking/information-about-tcp-chimney-offload-rss-netdma-feature.

[34] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based congestion control for the datacenter. In *Proceedings of the 2015 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '15, pages 537–550, New York, NY, USA, 2015. Association for Computing Machinery.

[35] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 313–326, New York, NY, USA, 2018. Association for Computing Machinery.

[36] Jeffrey C. Mogul. Tcp offload is a dumb idea whose time has come. In *Proceedings of the 9th USENIX Conference on Hot Topics in Operating Systems*, HotOS '03, page 5, USA, 2003. USENIX Association.

[37] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating network applications with stateful TCP offloading. In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation*, NSDI '20, pages 77–92, USA, 2020. USENIX Association.

[38] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click modular router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, SOSP '99, pages 217–231, New York, NY, USA, 1999. Association for Computing Machinery.

[39] Netronome. Netronome Agilio CX SmartNIC. https://www.netronome.com/products/agilio-cx/, 2018.

[40] Netronome. Netronome Agilio LX SmartNIC. https://www.netronome.com/products/agilio-lx/, 2018.

[41] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 327–341, New York, NY, USA, 2018. Association for Computing Machinery.

[42] Zhixiong Niu, Hong Xu, Peng Cheng, Qiang Su, Yongqiang Xiong, Tao Wang, Dongsu Han, and Keith Winstein. NetKernel: Making network stack part of the virtualized infrastructure. In *Proceedings of the 2020 USENIX Annual Technical Conference*, USENIX ATC '20, USA, 2020. USENIX Association.

[43] Zhixiong Niu, Hong Xu, Dongsu Han, Peng Cheng, Yongqiang Xiong, Guo Chen, and Keith Winstein. Network stack as a service in the cloud. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 65–71, New York, NY, USA, 2017. Association for Computing Machinery.

[44] NVM Express Workgroup. NVM Express: Base specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4a-2020.03.09-Ratified.pdf, 2020.

[45] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 337–350, New York, NY, USA, 2012. Association for Computing Machinery.

[46] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 1–16, Broomfield, CO, October 2014. USENIX Association.

[47] I. Pratt and K. Fraser. Arsenic: a user-accessible gigabit ethernet interface. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Society*, volume 1 of *INFOCOM '01*, pages 67–76 vol.1, 2001.

[48] 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. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 13–24. IEEE Press, 2014.

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

[50] Renato J. Recio, Paul R. Culley, Dave Garcia, Bernard Metzler, and Jeff Hilland. A Remote Direct Memory Access Protocol Specification. RFC 5040, October 2007.

[51] Redis Labs. memtier_benchmark: Load generation and bechmarking NoSQL key-value databases. https://github.com/RedisLabs/memtier_benchmark, 2020.

[52] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S. Berger, James C. Hoe, Aurojit Panda, and Justine Sherry. We need kernel interposition over the network dataplane. In *Proceedings of the 2021 Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 152–158, New York, NY, USA, 2021. Association for Computing Machinery.

[53] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 404–417, New York, NY, USA, 2017. Association for Computing Machinery.

[54] Pravin Shinde, Antoine Kaufmann, Timothy Roscoe, and Stefan Kaestle. We need to talk about NICs. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS '13, page 1, USA, 2013. USENIX Association.

[55] Pensando Systems. Pensando DSC-25 distributed services card. https://pensando.io/wp-content/uploads/2020/03/Pensando-DSC-25-Product-Brief.pdf, 2020.

[56] The Linux Foundation. toe. https://wiki.linuxfoundation.org/networking/toe.

Field	Bits	Description
Pre-processor (connection identification)—15B:		
`peer_mac`	48	Remote MAC address
`peer_ip`	32	Remote IP address
`local|remote_port`	32	TCP ports
`flow_group`	2	hash(4-tuple) % 4
Protocol (TCP state machine)—43B:		
`rx|tx_pos`	64	RX/TX buffer head
`tx_avail`	32	Bytes ready for TX
`rx_avail`	32	Available RX buffer space
`remote_win`	16	Remote receive window
`tx_sent`	32	Sent unack. TX bytes
`seq`	32	TCP seq. number
`ack`	32	TCP remote seq. number
`ooo_start|len`	64	Out-of-order interval
`dupack_cnt`	4	Duplicate ACK count
`next_ts`	32	Peer timestamp to echo
Post-processor (ctx queue, congestion control)—51B:		
`opaque`	64	App connection id
`context`	16	Context-queue id
`rx|tx_base`	128	RX/TX buffer base
`rx|tx_size`	64	RX/TX buffer size
`cnt_ackb|ecnb`	64	ACK'd and ECN bytes
`cnt_fretx`	8	Fast-retransmits count
`rtt_est`	32	RTT estimate
`rate`	32	TX rate

Table 5. Connection state partitions (total: 108B).

A TCP Connection State Partitioning

To enable fine-grained parallelism, we partition connection state across pipeline stages. Table 5 shows the per-connection state variables, grouped by pipeline stage. Pre-processor state contains connection identifiers (MAC, IP addresses; TCP port numbers). Protocol state contains TCP windows, sequence and acknowledgment numbers, and host payload buffer positions. Post-processor state contains host payload buffer and context queue locations, and data-path congestion control state. DMA and context queue stages are stateless.

In aggregate, each TCP connection has 108 bytes of state, allowing us to offload millions of connections to the SmartNIC. In particular, we can manage 16 connections per protocol FPC, 512 connections per flow-group, and 16K connections in the EMEM cache. Using all of EMEM, we can support up to 8M connections.

B Connection Splicing Implementation

We implement AccelTCP's connection splicing in 24 lines of eBPF code. Listing 1 shows the entire code.

C TAS TCP/IP Processing Breakdown

Table 6 shows a breakdown of the per-packet TCP/IP processing overheads (summarized as *TCP/IP stack* in Table 1) in TAS for the Memcached benchmark conducted in §2.1. For

```
BPF_MAP_HASH_DECLARE(splice_tbl, SPLICE_MAX_FLOWS, \
    sizeof(struct pkt_4tuple_t), sizeof(struct tcp_splice_t));

int bpf_xdp_prog(struct xdp_md* ctx)
{
  struct tcp_splice_t state;
  struct pkt_hdr_t    *hdr = BPF_XDP_ADDR(ctx->data);
  struct pkt_4tuple_t *key = &hdr->ip.src;

  // Filter non-IPv4/TCP segments to control-plane
  if (!segment_ipv4_tcp(hdr))
    return XDP_REDIRECT;

  // Connection Control: Segments with SYN, FIN, RST
  // Atomically remove map entry and forward to control-plane
  if (segment_tcp_ctrlflags(hdr)) {
    BPF_MAP_DELETE_ELEM(splice_tbl, key);
    return XDP_REDIRECT;
  }

  if (BPF_MAP_LOOKUP_ELEM(splice_tbl, key, &state) < 0)
    return XDP_PASS;    // Send to data-plane

  patch_headers(hdr, &state);
  return XDP_TX;    // Send out the MAC
}

void patch_headers(struct pkt_hdr_t *hdr,
                   struct tcp_splice_t *state)
{
  hdr->eth.src   = hdr->eth.dst;
  hdr->eth.dst   = state->remote_mac;
  hdr->ip.src    = hdr->ip.dst;
  hdr->ip.dst    = state->remote_ip;
  hdr->tcp.sport = state->local_port;
  hdr->tcp.dport = state->remote_port;

  hdr->tcp.seq  += state->seq_delta;
  hdr->tcp.ack  += state->ack_delta;
}
```

Listing 1. Connection splicing with XDP in FlexTOE.

each request, TAS performs loss detection (and potentially recovery) that involves processing the incoming request segment, generating an acknowledgement for it, and additionally, processing the acknowledgement for the response segment, consuming 42% of the total per-packet processing cycles. TAS spends 9% of the total cycles to prepare the response TCP segment for transmission and an additional 12% to schedule flows

Function	**Cycles**	**%**
Segment generation	130	9
Loss detection (and recovery)	606	42
Payload transfer	10	1
Application notification	381	26
Flow scheduling	172	12
Miscellaneous	141	10
Total	1,440	100

Table 6. Breakdown of TCP/IP stack overheads in TAS.

based on the rate configured by the congestion control protocol. TAS spends 26% of per-packet cycles interacting with the application, to notify when a request is received, to admit a response for transmission, and to free the transmission buffer when it is acknowledged. For small request-response pairs (32B in this case), the payload copy overheads are negligible.

D Control Plane

FlexTOE's control plane is similar to that of existing approaches that separate control and data-plane activities, such as TAS [19]. Using it, we implement control-plane policies, such as congestion control, per-connection rate limits, per-application connection limits, and port partitioning among applications (cf. [52]). We briefly describe connection and congestion control in this appendix. Retransmissions are described in §3.1.1 and §3.1.3. TAS [19] provides further description and evaluation of the control plane (named "slow-path" in the TAS paper).

Connection control. Connection control involves complex control logic, such as ARP resolution, port and buffer allocation, and the TCP connection state machine. The data-path forwards control segments to the control-plane. The control-plane notifies libTOE of incoming connections on listening ports. If the application decides to `accept()` the connection, the control-plane finishes the TCP handshake, allocates host payload buffers and a unique connection index for the data-path. It then sets up connection state in the data-path at the index location. Similarly, libTOE forwards `connect()` calls to the control-plane, which establishes the connection. On `shutdown()`, the control-plane disables the connection and removes the corresponding data-path state.

Congestion control. FlexTOE provides a generic control-plane framework to implement different rate and window-based congestion control algorithms, akin to that in TAS [19]. The control-plane runs a loop over the set of active flows to compute a new transmission rate, periodically. The interval between each iteration of the loop is determined by the round-trip time (RTT) of each flow. In each iteration, the control-plane reads per-flow congestion control statistics from the data-path to calculate a new rate or window for the flow. The rate or window is then set in the data-path flow scheduler (§3.4) for enforcement. We also monitor retransmission timeouts in the control iteration. FlexTOE implements DCTCP [1] and TIMELY [34] in this way.

E FlexTOE x86 and BlueField Ports

We have ported the FlexTOE data-path to the x86 and BlueField platforms. FlexTOE's design across the different ports is identical. We do not merge or split any of the fine-grained modules or reorganize the pipeline across ports. FlexTOE's decomposition, pipeline parallelism, and per-stage replication all generalize across platforms. Both ports are also almost identical to the Agilio-CX40 implementation (cf. §4) and were completed within roughly 2 person-weeks, demonstrating the great development velocity of a software TCP offload engine. We describe the implementation differences of each port to the Agilio-CX40 version in this section.

Hardware cache management. The hardware-managed cache hierarchies of x86 and BlueField obviate the need for software-managed caching that was implemented on Agilio. Instead of leveraging near-memory processing acceleration of the NFP-4000 (cf. §4.1), our ports implement multi-core ring buffers, flow lookup and packet sequencers in software. The more powerful x86 and BlueField cores make up for the difference in performance.

Symmetric core mapping. Unlike the NFP-4000, where FPCs are organized into islands, cores on x86 and BlueField have mostly symmetric communication properties, so the assignment of modules to cores is arbitrary and the manual FPC mapping step is omitted. However, we note that core mapping may still be beneficial, for example to leverage shared caches and node locality on multi-socket x86 systems. Each instance of a module runs on its own core. Apart from the six fine-grained pipeline modules: *pre-processing*, *protocol*, *post-processing*, *DMA*, *context queue*, and *SCH* shown in Figure 3, the ports utilize an additional *netif* module to interface with DPDK NIC queues to receive and transmit packets. Therefore, FlexTOE-scalar uses 7 cores and the FlexTOE-2× configuration uses 2 additional cores to replicate the pre and post-processing stages for a total of 9 cores.

Context queues use only shared memory. Our x86 and BlueField ports currently only support applications running on the same platform as FlexTOE. Hence, context queues always use shared memory rather than DMA. The corresponding DMA pipeline stage executes the payload copies in software using shared memory, rather than leveraging a DMA engine.

Platform-specific parameters. The replication factor of each pipeline stage is platform dependent. Stage-specific microbenchmarks on each platform can determine it. Our generalization experiments (§5.2) are designed to show that FlexTOE's data-parallelism can improve single connection throughput. Hence, we configure only one instance of the FlexTOE data-path pipeline in these versions (no flow-group islands—we do not process multiple connections in these experiments). Each port's pipeline uses the same number of stages as the Agilio-CX40 version, but we set different replication factors for the pre and post processing stages on x86 and BlueField (no replication and 2× replication). We do not attempt to find the optimal replication factor for best performance nor compact stages to reduce wasted CPU cycles.