F4T: A Fast and Flexible FPGA-based Full-stack TCP Acceleration Framework

Junehyuk Boo
junehyuk@snu.ac.kr
Seoul National University
MangoBoost Inc.
Seoul, Republic of Korea

Yujin Chung
yujin.chung@snu.ac.kr
Seoul National University
MangoBoost Inc.
Seoul, Republic of Korea

Eunjin Baek
ebaek@snu.ac.kr
Seoul National University
MangoBoost Inc.
Seoul, Republic of Korea

Seongmin Na
seongmin.na@snu.ac.kr
Seoul National University
Seoul, Republic of Korea

Changsu Kim
changsu.kim@mangoboost.io
MangoBoost Inc.
Seoul, Republic of Korea

Jangwoo Kim*
jangwoo@snu.ac.kr
Seoul National University
MangoBoost Inc.
Seoul, Republic of Korea

ABSTRACT

As complex workloads that run on many servers are pursuing higher networking throughput, more CPU cycles are consumed to support the TCP stack. To mitigate the high CPU burden from executing the compute-intensive TCP, prior works have proposed to offload TCP processing to the embedded processors, ASICs, or FPGAs in network devices. However, none of the approaches satisfy all of the critical requirements of TCP simultaneously, which are high performance, many connections, and high flexibility. Embedded processors do not provide enough performance to fully offload the TCP stack, while ASICs fail to provide enough flexibility. Meanwhile, existing FPGA-based TCP accelerators either fail to provide high performance or give up some of the critical features and requirements to achieve high performance due to their inefficient processing architecture.

This paper proposes F4T, a fast and flexible FPGA-based full-stack TCP acceleration framework to effectively save CPU cycles by achieving high performance, supporting many connections, and providing high flexibility. By analyzing the existing FPGA-based TCP accelerators, we identify that the key performance bottleneck arises from two factors: (1) inefficient processing of stateful operations that incurs stalls and (2) lack of TCP state management necessary for utilizing multiple memory modules. To resolve the identified bottlenecks, we design F4T to process stateful operations back-to-back without stalls and efficiently manage the TCP states among multiple memory modules. F4T also provides a full software-hardware stack that allows applications to easily utilize F4T without any modifications. Our evaluation results show that F4T saturates the 100Gbps link bandwidth with only two CPU cores while supporting many connections and providing high flexibility. F4T saves 64% CPU cycles and provides 2.8× CPU cycles to applications compared to Linux's TCP stack when running the Nginx web server.

CCS CONCEPTS

• **Hardware** → **Reconfigurable logic and FPGAs**; **Hardware accelerators**; **Networking hardware**; • **Networks** → **Transport protocols**; **Data center networks**.

KEYWORDS

Transmission Control Protocol (TCP), TCP offload engine (TOE), FPGA prototyping, FPGA accelerator, Full-stack TCP acceleration, High performance networking

ACM Reference Format:
Junehyuk Boo, Yujin Chung, Eunjin Baek, Seongmin Na, Changsu Kim, and Jangwoo Kim. 2023. F4T: A Fast and Flexible FPGA-based Full-stack TCP Acceleration Framework. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3579371.3589090

1 INTRODUCTION

Modern server workloads run on a cluster of networked computers using Transmission Control Protocol (TCP) which provides reliable communication capabilities. However, a significant amount of CPU cycles can be consumed to support the compute-intensive TCP stack. For example, data centers use 30-40% of the total CPU cycles to support the TCP stack [30, 32, 45], and our analysis shows that, when running the Nginx web server [12], the TCP stack consumes 37% of the total CPU cycles. Furthermore, our analysis indicates that CPUs require 104 cores to saturate a 100Gbps network with 128B requests and 13 cores with 1024B requests, whereas modern servers have only tens to hundreds of CPU cores. The number of CPU cores required to saturate the network can further increase as the network bandwidth increases beyond 100Gbps.

To save host CPU cycles required to support the complex protocol, server architects have proposed to offload TCP processing to embedded processors [34, 46], ASICs [15], and FPGAs [18, 44] integrated with NICs (i.e., smartNICs). They successfully mitigate the CPU burden and enable higher per-core performance. However,

*Corresponding author

ISCA '23, June 17–21, 2023, Orlando, FL, USA

ACM 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589090

ASICs fail to provide enough flexibility, which is crucial for the TCP stack [33, 38, 48]. Embedded processors and FPGAs provide flexibility but fail to achieve high performance and support many connections, which are the critical requirements of TCP.

Specifically, embedded processors cannot provide enough performance to offload the full TCP stack due to their TCP-inefficient general processing architecture [25, 46]. It is also challenging to scale up the number of cores to offload the full TCP stack and at the same time meet all of its critical requirements due to their limited power and area budgets [25]. On the other hand, FPGAs have the potential to construct a TCP-efficient accelerator thanks to their abundant amount of reconfigurable logic. However, most of the existing FPGA-based TCP accelerators fail to deliver high performance due to their inefficient processing architecture. While a recent work [18] achieves high performance, it sacrifices some of the TCP features, number of connections, and flexibility to achieve high performance. Therefore, it has become a critical challenge to architect a high-performance FPGA-based TCP accelerator that supports many connections and provides high flexibility.

In this paper, we design and implement an FPGA-based TCP acceleration framework to achieve high TCP processing performance while supporting a large number of connections (i.e., high connectivity) and providing high flexibility.

To achieve the goal, we first analyze the existing FPGA-based TCP accelerators and identify two main challenges that disable FPGA-based accelerators from achieving high performance. First, stateful TCP operations, which are read-modify-write (RMW) operations, should be performed atomically [46]. However, performing these RMW operations atomically causes hardware stalls that degrade the accelerator's performance. Therefore, the accelerator must efficiently cope with these stalls to achieve high performance. Second, orchestrating the TCP states distributed among multiple memory modules, which are required for high connectivity (i.e., large memory to store the TCP's per-flow states), is challenging without degrading the performance. With multiple memory modules, the accelerator has to process TCP even when flows are migrating among modules. Furthermore, frequent migrations can degrade the performance of the accelerator.

To overcome the challenges, we design a TCP-optimized architecture that enables F4T to achieve high performance and provide high connectivity and flexibility. First, F4T reorders the TCP processing to avoid stalls. Specifically, F4T delays the TCP processing and stores all information required for delayed processing. Then, it reorders the processing in a way that separates the atomic operations, thus allowing consecutive processing without stalls. Second, F4T orchestrates the TCP states in multiple memory modules by tracking the up-to-date locations of the flows and implements an efficient migration protocol that does not degrade the performance. This allows F4T to support tens of thousands of flows by utilizing DRAM and still achieve high performance. In addition, F4T also allows users to implement complex TCP algorithms with high processing latencies without losing performance.

For evaluation, we implement F4T on a Xilinx FPGA [16] and execute real-world network applications. Our end-to-end prototype evaluation results show that F4T can saturate the 100Gbps link bandwidth with only two CPU cores. The evaluation also shows that F4T can process 396 million requests per second with 16B requests. When running the Nginx web server, F4T saves 64% CPU cycles and provides 2.8× CPU cycles to applications compared to the Linux TCP stack. For connection support, F4T easily supports 65,536 concurrent TCP connections while using 32% of the FPGA resources. To demonstrate the flexibility of F4T, we show that common TCP algorithms can be easily programmed on F4T without degrading the performance.

In summary, our work makes the following contributions:

- **Bottleneck identifications:** We analyze the existing FPGA-based TCP accelerators and identify their key performance bottlenecks which preclude high TCP processing performance and limit connectivity and flexibility.
- **Novel TCP accelerator architecture:** We propose a novel FPGA-based TCP accelerator architecture that efficiently resolves the identified bottlenecks with our TCP-optimal FPGA design innovations.
- **Full-system prototyping:** We implement F4T as a full-system on a commodity FPGA and run real-world network applications (e.g., Nginx).
- **Promising results:** F4T saturates 100Gbps with 128B requests with only two CPU cores. Compared to Linux's TCP stack, F4T saves 64% of host CPU cycles and provides 2.8× CPU cycles to applications when running Nginx. F4T also supports 64K flows and provides high flexibility.

To the best of our knowledge, F4T is the first work to successfully achieve high performance while simultaneously providing high connectivity and flexibility.

2 BACKGROUND AND MOTIVATION

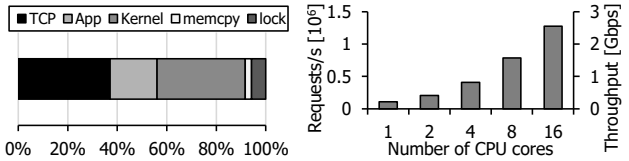
In this section, we first summarize the requirements of the TCP stack. Next, we introduce the previous TCP stacks that utilize host CPUs, embedded processors, ASICs, and FPGAs and analyze how well they meet the requirements.

2.1 Requirements of the TCP Stack

Performance. The TCP stack should achieve high performance regardless of the request size. The packet size in data centers varies from tens of bytes to more than a thousand bytes [19, 43]. Because the TCP stack has to handle more requests per second to saturate the link with small-sized requests, the TCP stack should be able to saturate the link bandwidth with small-sized requests, such as the median size of 128B [43], to meet the requirements of various workloads.

Connectivity. The TCP stack should support tens of thousands of concurrent TCP connections. Data centers currently utilize thousands of flows [43], and several applications require more [9, 14], in the range of tens of thousands of flows. To support the majority of workloads with various connectivity requirements, the TCP stack should support tens of thousands of concurrent flows.

Flexibility. The TCP stack should be reconfigurable to allow the users to modify the stack. The TCP stack is a fast-changing stack where Linux modifies over 5K lines of code annually [38] due to frequent security patches (e.g., [24, 31]) and improvements (e.g., [23]). Therefore, the TCP stack should be flexible enough to accommodate these changes. In addition, the TCP stack should not limit



(a) CPU utilization **(b) Performance**

Figure 1: CPU utilization and performance of Nginx

users from implementing complex algorithms due to tight performance constraints [35]. We refer to this specific type of flexibility as versatility throughout the paper.

2.2 The TCP Stack on Host CPUs

The most widely used TCP implementation is using the host CPU, which provides high connectivity and flexibility. By utilizing the memory hierarchy (i.e., fast SRAM cache and large DRAM), host CPUs support a large number of flows. They also achieve high flexibility by supporting various TCP algorithms and allowing modifications to the TCP stack.

However, the extensive CPU cycles used to support the TCP stack limits the application performance by taking the CPU cycles away from the application. To show the number of CPU cycles consumed by the TCP stack, we profile the performance of Nginx [12]. We generate the HTTP requests with wrk [26], and Nginx responds with a 256B response including the HTTP header and the HTML payload. We use Intel Xeon 5118 dual-socket CPU (hyperthreading disabled) with Linux and connect the nodes with TSO [21] and checksum-offload [4] enabled 100Gbps NICs [10]. Figure 1a shows that, when running Nginx, 37% of the total CPU cycles are consumed to support the TCP stack. Consequently, as shown in Figure 1b, Nginx cannot saturate the 100Gbps network and achieves a low performance in the range of a few million requests per second.

Meanwhile, the versatility of host CPUs is limited. Due to tight performance constraints, the TCP stack often avoids complex operations and limits operations to simple window or rate arithmetics [35]. As a result, host CPUs cannot achieve their maximum performance with complex TCP algorithms.

2.3 TCP Offloading to Embedded Processors

One possible approach to relieve the host CPU cycles from the compute-intensive TCP stack is to offload the TCP stack to embedded processors. Embedded processors, which are wimpy general-purpose processors on devices (i.e., smartNICs), provide high flexibility because they share the same architecture as host CPUs. However, embedded processors also share the limitations of host CPUs; they fail to achieve high performance. Moreover, embedded processors might perform worse than host CPUs due to the low performance of the wimpy cores [25] and the lack of hardware units such as dividers and timers [46]. Meanwhile, adding an extensive amount of cores on devices to achieve higher performance is not a scalable solution because devices have limited power and area budgets [25]. In that sense, embedded processors are insufficient for offloading the full TCP stack.

On the other hand, there have been attempts to partially offload the TCP stack to embedded processors. AccelTCP [34] partially offloads stateless TCP operations (e.g., connection setup and teardown) to embedded processors. Similarly, FlexTOE [46] offloads only the data path (e.g., packet generation) while the control path (e.g., connection and congestion control) is processed in host CPUs. Although they succeed in reducing the host CPU utilization, the host CPU is still responsible for a large portion of TCP processing. Therefore, the benefit of TCP stack offloading to embedded processors is limited as most of the CPU-intensive TCP processing remains in host CPUs.

2.4 ASIC-based TCP Accelerators

ASIC-based TCP accelerators have the potential to achieve high performance and save CPU cycles by offloading the TCP stack to their dedicated circuits. ASICs can achieve high performance by targeting a specific workload and performing workload-specific optimizations. Their architecture can also provide high performance by hard-wiring their logic (i.e., datapath) and exploiting hardware-level parallelism (e.g., pipelining).

However, ASIC-based TCP accelerators are rarely deployed due to their limited flexibility [33, 38, 48]. Because of their limited flexibility, ASICs cannot support all of the TCP's security patches and innovations that require changes throughout the stack. As a result, [48] explains that Microsoft deprecated their ASIC-based TCP accelerator due to security vulnerabilities, and [38] explains that Netflix excludes ASIC accelerators because it is hard to apply innovations to the TCP stack. Also, [46] shows that a commodity ASIC-based TCP accelerator [15] provides marginal performance improvements compared to host CPUs in certain workloads, and reasons that it is because of the lack of optimizations caused by the long hardware development cycles.

2.5 FPGA-based TCP Accelerators

FPGAs can provide high performance and flexibility thanks to their large amount of reconfigurable processing logic [25]. FPGAs can implement logic circuits like ASICs by emulating the functionality of gates, thus achieving high performance. Furthermore, they are flexible as they can be reconfigured into arbitrary logic. Due to the advantages, FPGA-based accelerators have been proposed as a promising solution for TCP stack acceleration [18, 44, 47]. However, most of them can saturate 100Gbps only with large-sized requests. Among them, TONIC [18] is the only one that achieves 100Gbps even with small-sized (i.e., 128B) requests. To achieve its target performance, however, TONIC sacrifices byte-level transfer and adopts segment-level transfer instead. It also provides limited connectivity and versatility due to the tight performance constraint. We further investigate the characteristics of TONIC.

Performance. TCP maintains per-flow transmission states (i.e., transmission control block or TCB [41]) and processes TCP events (i.e., user requests, received packets, timeouts [41]) by performing read-modify-write (RMW) operations on the TCBs. These RMW operations must be performed atomically for each flow to keep the consistency of the flow's TCB. Therefore, the latter event has to stall the processing until all previous events of the same flow are processed to ensure atomic TCP processing. These stalls limit most of the FPGA-based TCP accelerators from achieving high performance, but TONIC avoids the stalls by sacrificing some functionalities.

Table 1: Summary of the existing TCP implementations

	Host CPUs	Embedded processors	ASICs	Existing FPGAs	F4T
Host CPU util.	✘	△†	✔	✔	✔
Connectivity	64K+	64K+	64K+	1K	64K+
Flexibility	△‡	△‡	✘	△‡	✔

†: Limited improvement.
‡: Low versatility (limited flexibility when targeting maximum performance).

Specifically, TONIC avoids RMW stalls (i.e., stalling to keep TCP processing atomic) by obligating the TCP logic to perform all RMW operations in a single cycle. In this way, TONIC can process events back-to-back without stalling. Specifically, to achieve 100Gbps, TONIC targets a 10ns clock period (i.e., 100MHz frequency) and targets to transfer a 128B payload every cycle (thus 100Gbps). Then, to perform the compute-heavy TCP operations within 10ns, TONIC transfers data in 128B segments rather than in arbitrary lengths and limits to transfer at most one segment per cycle. However, segment-level data transfer is not fully compatible with TCP, which should be able to send data in arbitrary lengths, from a single byte to thousands of bytes. In addition, TONIC cannot further increase its performance because its frequency and segment size are fixed; the frequency is tightly bounded by the logic complexity and they target to achieve high performance with 128B requests.

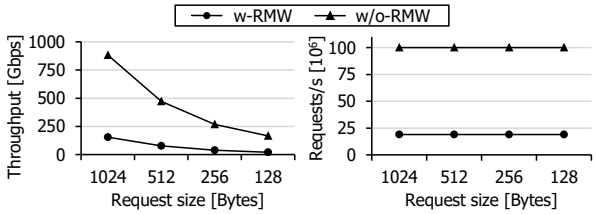
Connectivity. Like other FPGA-based TCP accelerators, TONIC supports about a thousand concurrent TCP connections because they store the TCBs only in SRAM. TONIC has no choice but to store the TCBs in SRAM because it needs fast memory to process the TCP's RMW operations (i.e., memory read, TCP processing, and memory write) in a single cycle. Also, only SRAM satisfies the TONIC's memory requirement of eight concurrent memory access every cycle required to achieve its target performance. Due to the limited connection support, the excessive flows will have to fall back to the traditional software TCP stack (Section 2.2), limiting the benefits of TCP offloading.

Flexibility. Although FPGAs are flexible due to their reprogrammability, the tight performance constraint of TONIC limits versatility. To ensure that all processing completes within a single clock period (i.e., 10ns), TONIC only supports TCP algorithms that can be processed in a single cycle. Therefore, it fails to implement complex algorithms or operations that incur long processing latencies. In addition, TONIC users must carefully implement algorithms to complete within a single cycle.

Table 1 summarizes the limitations of previous TCP stacks. First, CPU-based approaches consume a lot of host CPU cycles that can be provided to applications. Second, ASICs lack flexibility. Third, existing FPGAs support a small number of concurrent flows and provide limited flexibility (i.e., low versatility). Therefore, we propose F4T that saves host CPU cycles and achieves high performance, connectivity, and flexibility altogether.

3 CHALLENGES OF DESIGNING AN FPGA-BASED TCP ACCELERATOR

In this section, we introduce two main challenges in designing an FPGA-based TCP accelerator that achieves high performance, connectivity, and flexibility, which are (1) effectively coping with RMW stalls and (2) efficiently orchestrating multiple memory modules.



(a) Throughput **(b) Request processing rate**

Figure 2: Bulk data transfer performance

3.1 Effectively Coping with RMW Stalls

TCP operations consist of processing the TCP events with the TCB, which holds all transmission states. Specifically, the TCP stack executes RMW operations using TCBs to process an event. It reads the TCB, processes the event with the TCB, and stores the updated TCB back in memory. However, the stalls to keep the RMW operations atomic (i.e., RMW stalls) cause hardware underutilization and thus limit the accelerator from achieving high performance. At the same time, the accelerator should not limit its functionality (e.g., arbitrary length transfer) to avoid performance loss from RMW stalls.

To demonstrate the performance loss caused by RMW stalls, we show the performance of a design that stalls for 17 cycles between processing events (`w-RMW`) in Figure 2. The design comes from an existing 100Gbps-capable FPGA-based TCP accelerator [44] which operates at 322MHz and uses 17 cycles to process an event. We also show the performance of a theoretical design that does not have RMW stalls (`w/o-RMW`). The design comes from TONIC [18], which processes events in one cycle and operates at 100MHz. Unlike TONIC, however, we assume that the design can support requests of arbitrary lengths, while TONIC is fixed with 128B segments. The results are simulated, and we assume there is no link bandwidth bottleneck. The large performance gap between `w-RMW` and `w/o-RMW` in Figure 2 shows the performance loss caused by RMW stalls.

3.2 Orchestrating Multiple Memory Modules

Existing FPGA-based TCP accelerators utilize only SRAM to store the TCBs because SRAMs (i.e., BRAMs) in FPGAs have single-cycle access latency and provide high memory bandwidth. However, to support a large number of concurrent flows, the accelerator should have a large memory to store all TCBs, which exist per flow. As a result, for high performance and high connectivity, it is necessary to utilize both large memory (e.g., DRAM) to store all TCBs and fast SRAM to cache a portion of TCBs.

However, it is challenging to orchestrate the multiple memory modules efficiently without degrading performance. To process events when there are multiple memory modules, the accelerator must route the events to the memory where each event's TCB is stored. Furthermore, the TCBs can migrate among memory modules (e.g., swap in and out to and from DRAM), making it more challenging to route the events. For example, events should not be routed when their TCBs are migrating and do not have a fixed location. An event can still be dropped if the event's TCB migrates from the memory after the event is routed but before the event arrives at the memory. At the same time, the orchestration mechanism should not degrade the performance of the accelerator.

4 F4T ARCHITECTURE

In this section, we propose F4T, a novel FPGA-based TCP acceleration framework that provides high performance, connectivity, and flexibility. To begin with, we explain the overview of F4T from the software stack to our FPGA accelerator, FtEngine (Section 4.1). Then, we introduce our efficient TCP processing architecture in the following sections. We first explain how FtEngine avoids RMW stalls and thus achieves high performance (Section 4.2). Second, we explain our memory orchestration mechanism that allows us to efficiently utilize the memory hierarchy (Section 4.3). Third, we explain our additional optimizations to increase the performance of FtEngine (Section 4.4). We then discuss the flexibility of FtEngine (Section 4.5). Next, we provide the details of the software stack (Section 4.6). Finally, we provide the implementation results of FtEngine (Section 4.7).

4.1 F4T Overview

Figure 3 shows F4T's overview consisting of two parts: the software stack that links applications to the hardware and the FPGA TCP accelerator (FtEngine) that operates at 250MHz.

4.1.1 The software stack. The software stack of F4T consists of two parts: F4T library and F4T runtime.

F4T library. F4T library allows applications to utilize F4T without any modifications by providing the same functionality as POSIX socket API. By overriding the POSIX socket API via `LD_PRELOAD` [8], socket API calls are linked to the F4T library. In addition, F4T library runs as the same thread as the application thread, changing the socket API from system calls to function calls. All functions except for `epoll()` directly communicate with FtEngine and do not incur TCP processing in the software. Only a handful amount of metadata, such as TCP window pointers, are stored and managed in the software to implement the functionalities of the socket API. For example, when the TCP data buffer is full, `send()` requests are blocked on blocking sockets or returned with an error on non-blocking sockets. Meanwhile, F4T library provides the functionality of `epoll()` by maintaining an internal data structure (i.e., linked list of events) and returning them to applications.

F4T runtime. F4T runtime functions as a userspace device driver, enabling direct communication between F4T library and FtEngine. Specifically, F4T runtime mmaps the FtEngine's PCIe BAR region into userspace for F4T library to signal the hardware (i.e., ringing the hardware doorbell or HW DB) via memory-mapped I/O (MMIO). The runtime also registers hugepages into the IOMMU for DMA. On the hugepages, command queues of depth 1024, each entry holding a 16B command, are allocated per thread for the F4T library and FtEngine to send commands to each other. Requests such as `connect()`, `send()`, and `recv()` are sent to FtEngine with 16B commands, and FtEngine sends ACKed data and received data pointers to the software with 16B commands. TCP data buffers are also allocated in the hugepages to store data. On the other hand, FtEngine writes the software doorbell (SW DB) in the DMA buffer, as well as commands and payloads, to notify the software. The software later polls the software doorbell in memory when it needs to receive commands. We provide details on how F4T software achieves high performance without consuming a lot of CPU cycles in Section 4.6.

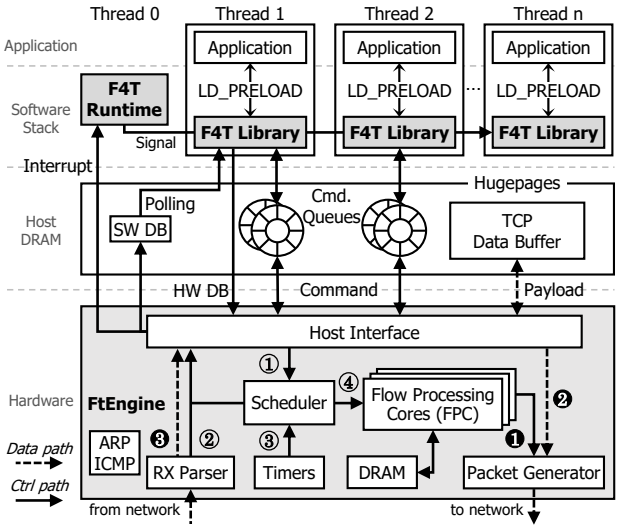


Figure 3: Overview of F4T

4.1.2 FtEngine. FtEngine consists of two parts: the control path and the data path. First, the control path provides the main functionalities of TCP by processing the TCP algorithms. In essence, it decides which packet to send and when to send the packet. Second, the data path assists the control path by generating packets to send, parsing packets received from the network, and handling the payload. FtEngine also implements ARP [39] for MAC address resolution and ICMP [40] for diagnostics such as ping.

Control path. The control path of FtEngine processes three types of events: user requests, received packets, and timeouts [41]. ① User requests, such as `send()` or `recv()`, are forwarded from the software to the host interface. ② Packets received from the network are pre-processed in the RX parser, and the RX parser generates events correspondingly. ③ Timers create timeout events. These events are first forwarded to the scheduler (Section 4.3), which orchestrates all flows. Then, ④ the scheduler forwards the events to their destination, such as flow processing core (FPC) or DRAM. FPCs process TCP algorithms by processing the forwarded events (Section 4.2).

TX data path. In the TX data path, the packet generator passively generates packets when FPC requests a data transfer (including retransmission). Specifically, ❶ based on FPC's request, the packet generator generates TCP/IP headers and ❷ fetches data from the TCP data buffer. The data is simply appended to the TCP header without any processing just before the packet is sent to the network. If the requested data transfer size exceeds the maximum segment size, the packet generator splits the request into multiple requests.

RX data path. In the RX data path, the RX parser first retrieves the received packet's flow ID by looking up a cuckoo hash table [3] with the 4-tuple (i.e., source and destination's IP address and port number). Flow ID is a unique ID used globally in F4T to identify a flow. Next, the RX parser ❸ DMAs the payload to the TCP data buffer if it fits in the receive window (regardless of whether it is in order) and drops if not. Applications, however, are notified about the received data only when the data is reassembled in order. This allows the hardware to reassemble data logically without actually

manipulating the data. To reassemble data in order, the RX parser stores the information of out-of-sequence data chunks and merges the received data into its adjacent data chunks.

4.2 Flow Processing Core Architecture

In this section, we thoroughly explain the flow processing core's architecture (FPC in Figure 3). FPC is the core module that efficiently processes all TCP algorithms. To avoid RMW stalls, FPC keeps a distance between the processing of the same flow. To achieve high performance even with the distance between processing, FPC first accumulates multiple events in memory by exploiting the properties of TCP. Then, the flow processing unit (FPU) in FPC processes the accumulated events later all at once.

4.2.1 Event Handling. FPC first accumulates the input events and then processes them later (Section 4.2.2). To keep the information of the events until they are processed, the event handler in FPC stores the event's information in memory (i.e., TCB) without processing the events. We refer to this storing of the event information as event handling in the following sections.

The event handler (Figure 4) accumulates input events by overwriting the event information for each flow, utilizing the properties of TCP. TCP's byte-stream abstraction represents the transmission state using cumulative pointers in the byte sequence space [37]. For example, TCP uses sequence (SEQ) number to indicate the boundary of data sent (i.e., all data until SEQ have been sent) and acknowledgment (ACK) number to indicate the boundary of successfully received data by the peer (i.e., all data until ACK have been received by the peer). Because an increased pointer contains the information of the previous pointer, events can be accumulated in a fixed-sized memory while preserving all information by overwriting the old transmission state to its new state.

As mentioned earlier, FtEngine processes three types of events: user requests, received packets, and timeouts. We further explain how the event handler handles the events by simply writing the up-to-date value to the TCB.

User Requests. User requests (e.g., `send()` or `recv()`) exchange information related only to the transmission state, such as send request (REQ). Assuming the previous REQ is 1000, it indicates that the application has requested to send all data from SEQ to sequence 1000. Then, when the user requests to send 300B, the event handler simply writes 1300 to REQ, overwriting the previous value of 1000. In detail, F4T library sends the pointer itself (e.g., 1300) instead of the request length (e.g., 300B) to support such event handling for user requests.

Received packets. Received packets contain transmission states (SEQ and ACK), window size, and flags. Transmission states and window size can be accumulated similarly to user requests because the last value holds the up-to-date value. Flags other than ACK only indicate the occurrence of each flag and therefore can be accumulated (ACK is discussed in the last paragraph).

Timeouts. Like flags in received packets, the occurrence of timeouts only matters and thus can be accumulated.

Counting (incrementing) duplicate ACKs was the only case that we could not simply write into memory. Although regular ACKs can be accumulated, when we receive a duplicate ACK, we need to increment the duplicate-ACK-count variable, which is an RMW

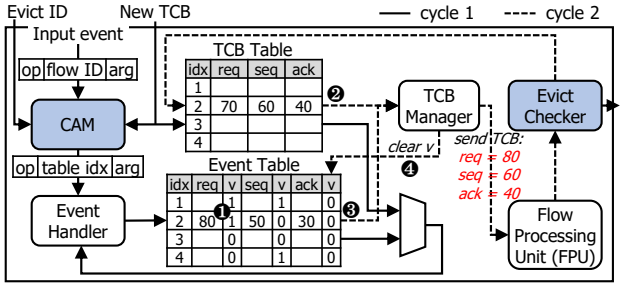


Figure 4: FPC architecture for TCP processing. Blue-colored modules are introduced in Section 4.3.

operation (i.e., read, increment, write). However, we can handle it immediately as incrementing can be done in a single cycle, not compromising back-to-back handling. In addition, we can append the event information to an additional buffer instead of handling it to support future TCP algorithms that might require long RMW handling.

4.2.2 Stateless TCP Processing. With the events accumulated by the event handler, the TCB manager and flow processing unit (FPU) process the events, each responsible for controlling the processing order and processing all TCP algorithms. The TCB manager controls the processing order by sending the TCBs to FPU in the desired order. Specifically, it sends the TCBs in a round-robin manner to keep the distance between the processing of the same flow. FPU is a stateless processing unit that processes all TCP algorithms only when it receives a TCB from the TCB manager. It can be stateless because all necessary information required to process TCP algorithms is in the TCB. After FPU receives a TCB, it processes TCP algorithms with the TCB and writes the updated TCB back to memory. Specifically, it decides which data to transfer (congestion and flow control), sends ACK for received data, advertises the receive window size, performs retransmission, and sends probe packets.

The TCB manager and FPU allow FPC to achieve high single-flow and multi-flow performance. First, because events accumulate into a single cumulative value, FPU can process the accumulated events all at once, achieving high single-flow performance. For example, when REQ and SEQ are 1000 and eight 100B requests accumulate, REQ becomes 1800, which is equivalent to a single 800B request. Second, FPU can be fully pipelined due to its stateless property and round-robin scheduling, achieving high multi-flow performance. Because round-robin scheduling selects the flows sequentially, FPU does not receive the same TCB from the TCB manager until it iterates on all flows. Therefore, there is no data hazard between TCP processing. Also, there are no structural hazards such as memory contention because FPU is stateless. Therefore, FPU can be fully pipelined and can process events of different flows back-to-back.

Because FPC is essentially delaying the processing of events, the delay could affect the end-to-end latency of the accelerator. However, the latency overhead is negligible as the maximum delay is bounded to the period of round-robin iteration. With 128 TCBs per FPC, the latency is bounded by 1μs, which is in the range of a single PCIe transaction latency [36]. Even with 1024 TCBs per FPC, the incurred latency of less than 8μs is still several orders lower than a typical TCP latency of milliseconds.

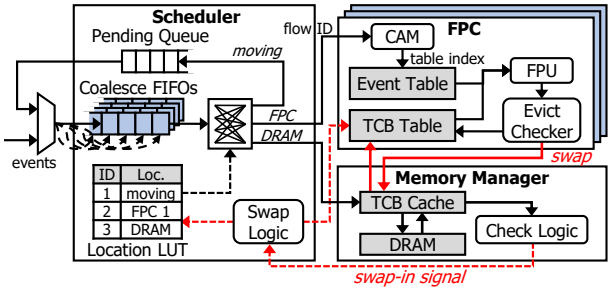


Figure 5: Memory orchestration architecture. The scheduler routes events (black arrows) and migrates TCBs among FPCs and DRAM (red arrows). Dotted lines indicate control signals and blue-colored modules are introduced in Section 4.4.

4.2.3 Dual Memory. Because both the event handler (Section 4.2.1) and FPU (Section 4.2.2) update the TCB, TCBs can become inconsistent due to overwrites. To keep the consistency of TCBs, we propose a dual memory architecture. Specifically, we allocate separate memory modules for each writer (i.e., FPU and event handler) to prevent them from overwriting values written by each other.

With dual memory, the TCB manager has to construct a valid, up-to-date TCB because there are two values for each entry in the two memories. Specifically, the TCB must be constructed as if all FPU processing (i.e., RMW operations) are done instantaneously (i.e., atomically). However, because FPU has a non-zero latency in practice, handled events are written into memory before FPU finishes processing. In this case, late writes from FPU are stale compared to the event handler's early writes. Therefore, to construct the up-to-date TCB, the TCB manager selects the values from the event table which were written after the previous TCB construction (i.e., when FPU starts processing). Otherwise, the TCB manager selects the values from the TCB table.

To implement the behavior, we add a valid bit for every entry in the event table. We further explain the mechanism with an example. As shown in Figure 4, when the event handler receives a send request with pointer 80, ❶ it writes 80 to the event table's `req` entry and sets the valid bit for `req`. When the TCB manager later decides to send the TCB to FPU, ❷ it reads the TCB from the TCB table (`req=70`, `seq=60`, `ack=40`), ❸ overwrites the TCB with the values in the event table with the valid bit set (`req=80`, `seq=60`, `ack=40`), and ❹ clears all valid bits for that flow.

Because FPGAs provide dual port memories (i.e., BRAM), we periodically perform the required memory accesses every two cycles. The two memories (i.e., TCB table and event table) allow four reads and four writes in two cycles. Thus, we schedule accepting input TCBs, writing updated TCBs from FPU, and writing handled events from the event handler as well as reads to construct the valid TCB in two cycles. In one cycle (solid arrow), the TCB table accepts input TCBs, and the event table stores handled events. In the other cycle (dotted arrow), the TCB table stores the updated TCB from FPU, and the TCB manager clears the valid bits in the event table. Both memories are read every cycle to construct the TCB for the event handler (for single-cycle RMW operations) and the TCB manager. As a result, FPC can process an event every two cycles or process 125 million events per second at 250MHz.

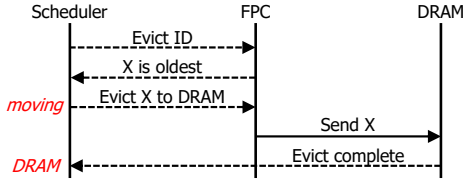


Figure 6: TCB migration process. Dotted arrows, solid arrows, and red italic words indicate requests, TCB migration, and updates on location LUT, respectively.

4.3 Efficient Memory Orchestration

In this section, we explain the design of our efficient memory orchestration mechanism, which allows us to utilize DRAM and thus support 64K concurrent flows (i.e., high connectivity) without performance degradation. Specifically, we utilize the on-board DRAM to store the extensive amount of TCBs and efficiently migrate TCBs between FPC and DRAM.

4.3.1 Memory Hierarchy. Because there are multiple memory locations (i.e., FPC and DRAM), events must be routed to the correct destination, where the events' TCBs are stored. To route the events, the scheduler (Figure 5) tracks the locations of the TCBs and routes events by looking up the location lookup table (location LUT).

We must handle the events routed to the DRAM similarly to those which are routed to FPC. Therefore, we implement the memory manager (Figure 5) that handles the events routed to DRAM. The memory manager does not process TCP algorithms but handles them like the event handler in FPC, and the handled events are later processed in FPC. It also includes a direct-mapped TCB cache (Figure 5) to handle the frequently accessed TCBs more efficiently.

To swap flows back into FPC, the memory manager checks whether each flow can send packets and swaps only the necessary flows to FPC. Specifically, after the memory manager handles an event, it sends the TCB to the check logic, and it checks whether the flow can send packets (e.g., send or retransmit data, send probe or ACK, etc.). If the flow can send packets, the check logic signals the scheduler to swap in the TCB to FPC. If the flow cannot send packets, it can wait in the memory manager until it can send packets. Because the check logic does not process TCP algorithms and does not write back the TCB, the memory manager can handle events directly to TCBs in the memory.

4.3.2 TCB Scheduling. The scheduler should migrate TCBs between FPC and DRAM and route the events to their proper destination even when TCBs are migrating. Most importantly, it should not degrade the performance of TCP processing.

To track the locations of the TCBs, the scheduler controls the whole process of all TCB migrations. Figure 6 shows an example case of how the scheduler migrates a flow from FPC to DRAM. First, the scheduler asks FPC for a flow to evict. Then, FPC returns the ID of its least recently active (i.e., coldest) flow. Next, the scheduler requests to evict the flow to DRAM and changes the location LUT's entry to moving state (i.e., the flow is migrating). When FPC receives the evict signal, it sets the TCB's evict flag. We also place the evict checker in FPC (Figure 4) that intercepts processed TCBs from FPU and evict TCBs with the evict flag set. When the evicted TCB

arrives at the DRAM (i.e., memory manager), the DRAM sends an evict-complete signal to the scheduler to change the location LUT's entry to its new location, DRAM.

To efficiently migrate TCBs without degrading the performance, we carefully design the eviction and swap-in operations to be pipelined. First, the evict checker does not consume an additional memory port to evict the TCBs, thus preserving the performance of FPC. Also, because FPU outputs a TCB every two cycles, the evict checker can evict TCBs forwarded from FPU at the same rate. Second, FPC can accept new TCBs from DRAM back-to-back thanks to the dedicated write memory port (that supports a write every two cycles) to accept new TCBs (Section 4.2.3). Therefore, the scheduler can continuously migrate TCBs between FPC and DRAM.

We further explain how we handle two problematic scenarios when routing events. First, a TCB does not have a valid location when it is being migrated. To resolve this issue, the scheduler allocates a pending queue (Figure 5) and temporarily stores the events that do not have a destination. The scheduler retries the routing after 12 cycles, and it always succeeds because all migration completes within 12 cycles. This guarantees that the pending queue will not become full. Second, TCBs can evict before routed events are processed. Eviction with unprocessed events may lead to deadlocks because both the TCP's opponent and the memory manager can wait for the unprocessed event. However, the scheduler prevents such a scenario by blocking the routing of input events to their destinations with the moving state, which is set at the same time as the evict request. Also, the evict checker in FPC ensures that TCBs are always processed before they are evicted by evicting only the processed TCBs forwarded from FPU.

4.4 Additional Performance Optimizations

In this section, we explain the additional performance optimizations applied to FtEngine to increase its performance. First, to achieve higher single-flow performance, we coalesce events of the same flow in the scheduler to reduce the total number of events routed to FPCs. Second, to achieve higher multi-flow performance, we implement parallel FPCs to process different flows in parallel.

4.4.1 Event Coalescing. To increase the performance of processing events of the same flow, we coalesce events in the scheduler before they are routed. The scheduler coalesces events similarly as the event handler accumulates events, but only if no information is lost (e.g., duplicate ACK). Specifically, user requests can always coalesce because they send only the transmission state (Section 4.2.1), and received packet events can coalesce only if there are no packet drops or reordering, similar to GRO [22]. To coalesce events of the same flow, events are first hashed into four 16-entry FIFOs (coalesce FIFOs in Figure 5). If no information is lost, each FIFO checks whether there exists an event in the FIFO with the same flow ID and coalesces the events if the flow ID matches. If not, the event is pushed into the FIFO. As a result, events of the same flow coalesce in the scheduler before they are routed, reducing the number of events routed to FPC and DRAM.

4.4.2 Parallel FPCs. To increase the event processing rate of different flows, we can allocate multiple FPCs to process the events in parallel. Events of different flows can be processed in parallel because there are no dependencies between different flows.

Table 2: Target situations of F4T's solutions

Target Situation	F4T's Solution
Targets all situations	FPC architecture
Events of the same flow	Scheduler (Event coalescing)
Events of different flows	Parallel FPCs
Event load imbalance	Scheduler (FPC migration)

To support parallel FPCs, we need two modifications: faster event routing in the scheduler and TCB table index lookup in FPC. First, to route the events at a higher rate, we implement the location LUT with actual LUTs [2] instead of SRAM and partition them into multiple groups to support concurrent access per cycle. Then, the scheduler can route multiple events per cycle by hashing the input events to different LUT groups. For example, to support eight FPCs, each processing an event every two cycles, we need four LUT partitions to route four events per cycle. Second, FPCs should manage the mapping between the global flow ID and the local TCB table index. Therefore, to identify each event's TCB table index, we implement a content-addressable memory (CAM) in each FPC to look up the table index with the flow ID, as shown in Figure 4. Because the scheduler always routes the events to their correct destination (Section 4.3.2), we can ensure that the CAM lookup always hits on one entry. Therefore, we implement the CAM with a comparator array and a binary log module.

To fully utilize the parallel FPCs, we dynamically allocate the TCBs among FPCs. With static TCB allocation, flows can be concentrated to a few FPCs because workloads continuously establish and terminate flows. Furthermore, even if the flows are distributed, input events can be biased to a single FPC, causing load imbalance among FPCs. To resolve these issues, the scheduler (Figure 5) allocates new flows to the FPC with the lowest flow count and migrates flows from congested FPCs. Specifically, the scheduler detects the backpressure of events routed to each FPC and migrates the first flow headed to the congested FPC to the idlest FPC.

Moreover, it is possible to scale the number of FPCs without changing their operating frequency or pipeline depth because each FPC operates independently. As a reference design, we implement eight FPCs in parallel, each with 128 flows, supporting a total of thousand flows in FPCs. Other configurations are also possible for different needs. We only have to scale up the glue logic (e.g., switches that connect FPCs) with the number of FPCs. Performance-wise, the data path modules (e.g., packet generator, RX parser) should increase their parallelism for higher throughput. Specifically, the packet generator can be easily parallelized as its operation is stateless and does not have dependencies between pipeline stages. The RX parser can parallelize packet parsing and flow ID lookup by partitioning the memory and accessing them in parallel.

Table 2 summarizes how F4T achieves high performance in all situations. First, FPC architecture boosts the performance to 125 million events per second in all situations by handling events back-to-back. Second, to further increase the performance of processing events of the same flow, the scheduler coalesces events of the same flow to reduce the number of events routed to FPCs. Third, parallel FPCs accelerate the processing of different flows in parallel FPCs.

Finally, the scheduler resolves event imbalance among FPCs by migrating TCBs from the congested FPCs.

4.5 Versatility and Programmability

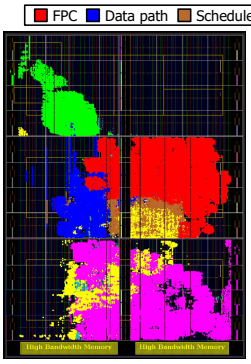
Unlike previous FPGA-based TCP accelerators, FtEngine can support complex TCP algorithms with long processing latencies without degrading the performance (i.e., high versatility). In F4T, the latency of FPU increases as the TCP processing latency increases. However, longer FPU latency does not degrade the event processing performance of FPC. For events of the same flow, the performance depends solely on the event handling rate, regardless of FPU latency, because FPU processes the handled events all at once. Next, for events of different flows, FPU can process the events back-to-back regardless of its latency because it is fully pipelined. As a result, we can implement complex algorithms without performance loss. For example, we can implement the congestion control algorithm CUBIC TCP [42], which requires cube and cubic root operations, without any performance degradation.

FPC architecture also allows users to easily program the TCP stack. Because FPU processes all TCP algorithms, users need to modify only the FPU to program the TCP stack. Furthermore, FPC is designed with high-level synthesis [6], which allows users to implement hardware logic by describing its functionality in C++. Because there are no structure and data hazards in FPU (Section 4.2.2), users can program TCP algorithms in C++ with almost zero hardware knowledge. After users write their code in the placeholder, which includes TCB input/output and pipelining pragma, the compiler automatically pipelines and synthesizes the FPU.

4.6 F4T's Software Stack

In this section, we describe the various optimization techniques applied to F4T's software stack and discuss its scalability. F4T offloads almost all TCP processing to hardware, and the software stack performs only a few functionalities, such as supporting blocking API when the TCP buffer is full (or empty). Also, the socket API overheads are minimal because socket API functions are plain function calls to F4T library instead of mode-switching system calls. When sending commands to FtEngine, F4T library performs MMIO batching to reduce the number of CPU's PCIe transactions. We use hugepages for the DMA buffer to reduce TLB misses, and Intel data direct I/O [11] caches the buffers in LLC instead of DRAM. To save CPU cycles when F4T is waiting for the network's response, the library can go to sleep after polling for blocking APIs for a certain amount of time (e.g., 10μs). Then, F4T runtime signals and thus wakes the sleeping thread when there are new commands. Because F4T library runs only when applications call the socket API, F4T software does not consume CPU cycles when there are no requests.

F4T's software stack is scalable with respect to the number of CPU cores because it does not share any internal variables among threads and avoids using locks, similar to DPDK [29]. We allocate per-thread command queues that are dedicated to each application thread. To effectively support multithreading, FtEngine supports the `SO_REUSEPORT` option [1], which evenly distributes new flows to the application threads. Also, FtEngine supports receive-side scaling [5], which sends commands of the same flow to a single queue for the CPU to better exploit the CPU's cache locality. As a result, the performance of the software stack scales linearly with the number of application threads.



(a) FtEngine (8FPC) layout

	LUT (1.3M)	FF (2.6M)	BRAM (2016)
FtEngine (1FPC)	211K	274K	539
FtEngine (8FPC)	303K	387K	651
Components of FtEngine			
- 1 FPC†	13K	16K	16
- 1 CAM	4188	2424	0
- Memory manager	32K	60K	65
- Network	15K	37K	0
- PCIe	105K	106K	180
- Others	48K	66K	278

†: includes the resources of a CAM

(b) Resource utilization

Figure 7: Implementation result of FtEngine with HBM

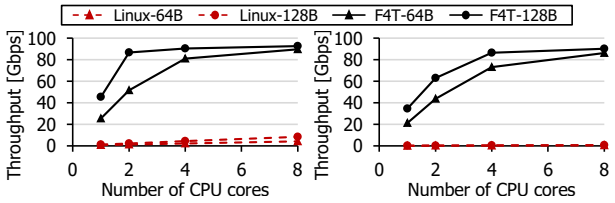
4.7 Implementation

We implement FtEngine on a Xilinx U280 FPGA board [16]. Figure 7a shows the FtEngine's implementation result synthesized with Vivado 2020.2 [7]. The modules that communicate with the network (i.e., ARP, ICMP, packet generator, RX parser) operate at 322MHz, which is the Ethernet IP's frequency, and all other modules operate at 250MHz. As shown in Figure 7b, FtEngine with a single FPC utilizes 16% LUTs, 11% FFs, and 27% BRAMs and FtEngine with eight FPCs utilizes 23% LUTs, 15% FFs, and 32% BRAMs in the FPGA. While the top two rows of Figure 7b (i.e., FtEngine) show the total amount of resources used in the hardware, the table also shows the resource utilization of the F4T's components below. The remaining logic can be used to implement complex algorithms, more FPCs for higher performance, or other networking functionalities. For DRAM, we utilize two types of memory: (1) DDR4 DRAM which provides 38GB/s, or (2) high bandwidth memory (HBM) which provides 460GB/s. The resource utilization (Figure 7) comes from the design using HBM, and the numbers do not differ significantly when using DRAM.

Although we have prototyped FtEngine on an FPGA, the architecture of F4T does not rely on FPGA-specific features except for flexibility. Therefore, if users prioritize performance and power efficiency over flexibility, it is possible to implement FtEngine in ASIC by compromising flexibility.

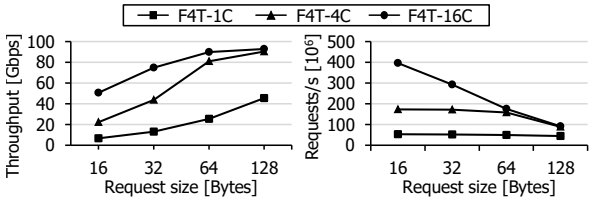
5 EVALUATION

In this section, we answer the following five questions to show that F4T achieves the TCP requirements in Section 2.1. (1) Can F4T saturate the link bandwidth with various request sizes and save CPU cycles (Section 5.1)? (2) Depending on the request size, how much throughput can F4T achieve and how many requests can F4T process per second (Section 5.1)? (3) Can F4T save CPU cycles of real-world workloads (Section 5.2)? (4) Can F4T support tens of thousands of concurrent connections and still provide high performance (Section 5.3)? (5) Does F4T provide high flexibility for users to implement arbitrary TCP algorithms (Section 5.4)?



(a) Bulk data transfer (b) Round-robin requests

Figure 8: Throughput with different request patterns



(a) Throughput (b) Request processing rate

Figure 9: Bulk data transfer with various request sizes

Evaluation setup. We test F4T as an end-to-end system, where the software stack runs on a dual-socket Intel Xeon Gold 5118 CPU at 2.3GHz, and disable hyperthreading. We install Ubuntu 20.04 with Linux kernel v5.4 on the servers and equip them with NVIDIA ConnectX-5 NICs [10]. Also, we set up the network by directly connecting (1) two NICs, (2) a NIC to FtEngine, and (3) two FtEngines. The TCP buffer size is 512KB, and the maximum segment size is 1460B.

To evaluate F4T in an end-to-end system, we compare F4T using eight FPCs with Linux's TCP stack (host CPU). To the best of our knowledge, existing FPGA-based TCP accelerators do not provide a full-stack implementation. For example, [44] implemented the TCP stack in an FPGA, but it targets applications that run on FPGAs and does not provide a software interface. Also, TONIC [18], another FPGA-based accelerator, implements only the TX path and lacks TCP features. Specifically, it does not support byte-level transfer and thus requires an additional software layer to support the socket API, which is currently not provided. Therefore, we evaluate end-to-end F4T with Linux. Note that we also use a simulation environment introduced in Section 3.1 to evaluate FPGA-specific features (e.g., FPC versatility test) in Section 5.4.

5.1 Performance

To evaluate the end-to-end performance of F4T, we perform data transfer between two hosts connected with F4T and a 100Gbps link. To evaluate the performance of F4T in various situations, we test F4T in two extreme situations. First, to show the performance of processing events of the same flow, we perform bulk data transfer with iPerf [28], where each CPU core generates send requests for a single flow (bulk data transfer). Because FtEngine reads multiple commands from each command queue at once, events of the same flow are injected consecutively. Second, to show the performance of processing events of different flows, we build a benchmark where each CPU core generates send requests in a round-robin manner for 16 flows (round-robin requests). Each CPU core uses a distinct set of 16 flows. FtEngine receives events of different flows because adjacent requests in each queue are from different flows.

Figure 8a shows the bulk data transfer throughput of Linux and F4T with 64B and 128B requests. The result shows that Linux fails to saturate the 100Gbps link with 128B requests and achieves only 8.3Gbps with eight cores. On the other hand, F4T achieves 45Gbps (44 million requests per second or 44Mrps) with a single core and reaches 87Gbps (85Mrps) with 128B requests using only two cores. The throughput is saturated at 92.6Gbps (90.4Mrps) with eight cores. This shows that F4T can hit near-maximum throughput with just two cores. With 64B requests and eight cores, F4T achieves 89.7Gbps (175Mrps). F4T is achieving its theoretical maximum throughput in a 100Gbps environment because the application (i.e., iPerf) measures the goodput. It counts only the payload and not the 78B packet overheads (i.e., 40B TCP/IP headers, 18B ethernet header, 8B preamble, and 12B inter-frame gap).

On the other hand, Figure 8b shows the throughput of Linux and F4T when sending 64B and 128B requests in a round-robin manner. The result shows the throughput of Linux is lower than 1Gbps, supporting 0.126Gbps with a single core and 0.833Gbps with eight cores. However, the throughput of F4T reaches 35Gbps (34Mrps) with a single core, 63Gbps (62Mrps) with two cores, and 90Gbps (88Mrps) with eight cores.

Both Linux and F4T show lower performance with round-robin requests compared to bulk data transfer because the low flow locality results in many small-sized packets. With small packets, the portion of the 78B per-packet overhead increases, resulting in lower goodput (e.g., with 128B packets, the goodput is $100\text{Gbps} \times 128 \div (128 + 78) = 62.1\text{Gbps}$). For F4T, the large number of packets also increases the number of hardware commands for the F4T software to process, resulting in lower per-core throughput. However, F4T eventually achieves high throughput near 90Gbps because FPC can always handle events back-to-back even when the network is bottlenecked. When the network is bottlenecked, the packet generation rate decreases, and the increased packet generation period provides more time for the events to be accumulated before they are processed. Then, the packet size increases as the accumulated events are equivalent to a single large-sized request.

Next, we measure F4T's bulk data transfer performance with smaller request sizes to better observe its performance. With request sizes as small as 16B, we can mitigate the link bandwidth bottleneck and measure higher performance in requests per second.

Figure 9 shows the performance of F4T with different request sizes and different numbers of CPU cores. The postfix nC indicates that n CPU cores were used in the experiment. The result shows that F4T achieves 50.7Gbps (Figure 9a) or 396Mrps (Figure 9b) with 16B requests and using 16 CPU cores. We observe that this performance is bounded by the PCIe bandwidth, where each 16B request requires a 16B command and 16B payload DMA.

5.2 Real-world Application Benchmark

To demonstrate the real-world applicability and effectiveness of F4T, we measure the performance of a popular web server, Nginx [12], running on F4T and Linux. Specifically, we show the improvement of the request processing rate compared to Linux and further

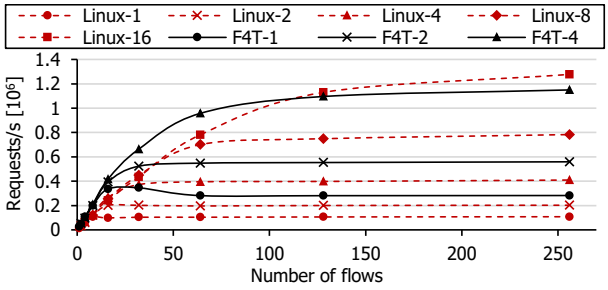


Figure 10: Request processing rate of Nginx

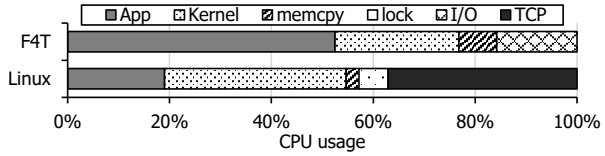


Figure 11: CPU utilization breakdown of Nginx

analyze the results with the CPU utilization breakdown and the latency results. We use wrk [26] running on Linux to generate HTTP requests and use 256B response for Nginx, including the HTTP header and the HTML payload. We use 256B responses instead of 128B because the HTTP header generated by Nginx is larger than 128B.

Figure 10 shows the request processing rate of Nginx running on F4T and Linux. When using one to four CPU cores, F4T achieves 2.6× to 2.8× higher performance than Linux at the performance saturation point (i.e., 256 flows). Therefore, F4T saves up to 64% of CPU cycles compared to Linux to provide the same throughput.

Next, Figure 11 shows the CPU utilization breakdown when F4T uses a single CPU core and services 64 flows in Nginx. The result shows that F4T successfully removes all CPU cycles used to perform TCP computations and provides 2.8× more CPU cycles to the application. The number of 2.8× is similar to the performance gap between Linux and F4T shown in Figure 10, explaining that the CPU cycles provided to the application improved the request processing rate. Meanwhile, we observe that F4T still consumes a significant amount of CPU cycles in the kernel and identify that it is caused by the filesystem access (e.g., `vfs_read`) to fetch the HTML files.

In the same setup as Figure 11, Figure 12 shows the median and 99th tail latency of Nginx running on Linux and F4T. The result shows that although FtEngine delays the processing of events, it shows lower latencies (3.7× shorter median latency and 26× shorter 99th latency) compared to Linux thanks to its efficient architecture. Because latency dominates the performance in low-load situations (i.e., a small number of flows), F4T achieves higher performance than Linux in Figure 10 with fewer flows.

5.3 Connectivity

To evaluate F4T's connectivity, we run an echoing benchmark that sends a 128B payload when it receives a message from the other (i.e., ping-pong). In this echoing benchmark, each flow has to wait for a response to send the next message. Thus, the TCB access

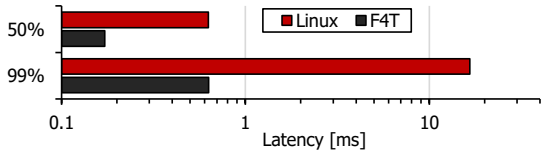


Figure 12: Median and 99th tail latency of Nginx

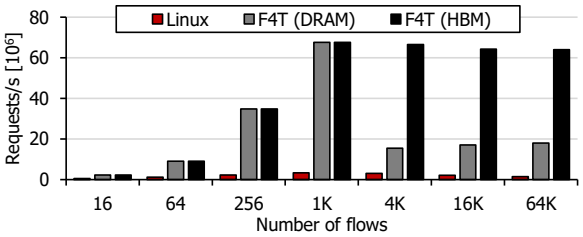


Figure 13: Request processing rate of 128B echoing with various numbers of flows

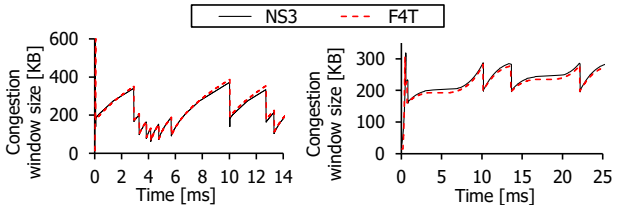
pattern has a very low temporal locality and results in the worst-case performance when utilizing DRAM. We evaluate Linux and F4T utilizing either DRAM or HBM running on eight CPU cores.

Figure 13 shows the request processing rate of the echoing benchmark with different numbers of flows. The result shows that F4T achieves higher performance for all numbers of flows, providing 20× higher request processing rate than Linux at 1K flows and 12× and 44× higher request processing rates with DRAM and HBM, respectively, at 64K flows. Although Linux supports a large number of flows, it fails to provide high throughput. On the other hand, TONIC supports 1K flows, which is far less than Linux or F4T.

F4T with DRAM experiences a performance drop when supporting more than 1024 flows. As our reference design holds 1024 flows in FPCs, requests frequently incur DRAM swaps when the number of active flows exceeds 1024. Therefore, the performance is throttled by the DRAM bandwidth. On the other hand, F4T with HBM achieves high performance even when supporting a large number of flows because the high bandwidth of HBM allows to access a TCB every cycle. For different request patterns with higher flow locality (e.g., a few high-throughput flows and many low-throughput flows, commonly observed in data centers [17]), we expect better performance as SRAM in FPCs will hold the high-throughput flows for a longer amount of time, reducing DRAM swaps.

5.4 Flexibility and Versatility

To show the flexibility of F4T, we implement two common congestion control algorithms, NEW RENO [27] and CUBIC TCP [42], on F4T and compare the results with a well-known network simulator, NS3 [13]. We could implement these congestion control algorithms simply by programming the behavior of FPU in C++ and adding some entries in the TCB. With cycle-accurate RTL simulation [7], we perform a single-flow bulk data transfer and inject occasional packet drops. Figure 14 shows the congestion window change of F4T and NS3. The result shows that F4T faithfully implements the behavior of congestion control algorithms. Other TCP modifications (e.g., flow control algorithm) can be supported similarly by modifying the FPU.



(a) New RENO (b) CUBIC TCP

Figure 14: Congestion window size of NS3 and F4T

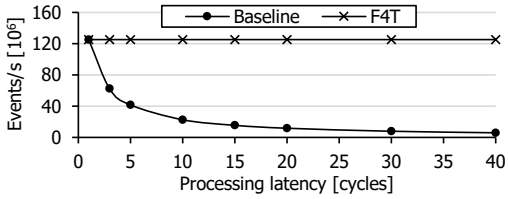


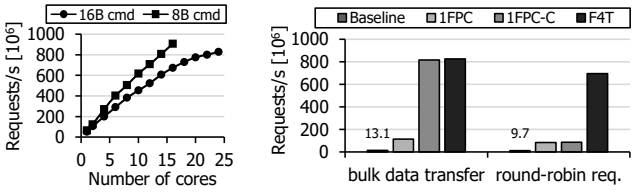
Figure 15: Event processing rate of Baseline and F4T with various processing latencies

Next, we show the versatility of F4T by measuring the event processing rate of FPCs in simulation while varying the processing latency of the FPU. `Baseline` is a design based on the existing FPGA TCP accelerator [44], which stalls to keep the atomicity of TCP operations. Figure 15 shows the performance of F4T and `Baseline` with various processing latencies. The result shows that while `Baseline`'s performance decreases when the processing latency increases, F4T always maintains its event processing rate regardless of the processing latency. As a result, F4T can implement complex algorithms with high processing latencies without degrading performance. For example, we implemented TCP Vegas [20], and FPU takes 68 cycles to process due to integer divisions. However, it still achieves the same maximum performance as NEW RENO and CUBIC, whose processing latency is 14 and 41, respectively.

6 PERFORMANCE POTENTIAL ANALYSIS

In this section, we target to further analyze the performance potential of F4T. Because current FPGAs support a maximum 100Gbps link bandwidth (i.e., Ethernet and PCIe), the maximum real-world throughput we can get is 100Gbps. Therefore, we emulate the situation without the link bottleneck to further analyze the performance potential of F4T. Specifically, we prototype a special hardware where two FtEngines are connected to each other within an FPGA, and each transfers packets without the payload (i.e., header). One FtEngine is controlled by the F4T software while another is controlled by a custom command generator implemented in hardware. By excluding payload transfer, we can measure the header processing rates of F4T when there is no link bandwidth bottleneck. Although the results do not show the end-to-end request processing rate of F4T, we believe these results will help predict the overall performance because TCP processing is mainly done with the header.

Figure 16a shows the header processing rate of bulk data transfer while changing the number of CPU cores. The result shows that F4T's header processing performance scales when there is no link



(a) Performance with various number of cores (b) Performance of intermediate hardware designs

Figure 16: Header processing rate. Payload handling is excluded.

bandwidth bottleneck. The result also shows that the software stack can support high request processing rates in such scenarios. At this rate, the PCIe bandwidth saturates even with transferring only the 16B commands used for software-hardware communication. To eliminate this bottleneck, we simplify the commands from 16B to 8B commands and observe that the performance of F4T scales linearly until around 900Mrps.

To further understand the effect of F4T's solution, we also measure the performance of intermediate designs of F4T in such a system. `Baseline` is a design that always incurs RMW stalls for 17 cycles to wait for the previous processing to complete, similar to [44]. `1FPC` is a design that adopts one FPC (Section 4.2) from the baseline, and `1FPC-C` adds event coalescing (Section 4.4.1) to `1FPC`. We utilize 24 CPU cores for all designs.

Figure 16b shows the header processing rate of F4T's designs with bulk data transfer and round-robin requests. While `Baseline` shows low performance due to RMW stalls, `1FPC` shows 8.6× and 8.4× higher request processing rates in bulk data transfer and round-robin requests, respectively. Next, the bulk data transfer performance of `1FPC-C` shows that event coalescing is effective in processing events of the same flow, achieving 62.3× higher rate. However, it is not effective in processing events of different flows (8.6× in round-robin requests). Finally, F4T shows that parallel FPCs are effective in processing events of different flows, achieving 63.1× and 71.3× higher rates in bulk data transfer and round-robin requests, respectively.

7 CONCLUSION

F4T is an end-to-end TCP acceleration framework that can saturate 100Gbps with only two CPU cores, supports 64K concurrent connections, and provides high flexibility. F4T achieves these features by adopting a TCP-optimal architecture that (1) processes events back-to-back without violating the atomicity of TCP processing and (2) manages the TCB states efficiently to allow the utilization of a memory hierarchy including SRAM and DRAM. Users can easily implement arbitrary algorithms, including those that are complex, into F4T without any performance degradation.

ACKNOWLEDGMENTS

This work was supported in part by MangoBoost Inc., and in part by the Automation and Systems Research Institute (ASRI) and Inter-university Semiconductor Research Center at Seoul National University.

REFERENCES

[1] 2013. The SO_REUSEPORT socket option. https://lwn.net/Articles/542629/.
[2] 2017. UltraScale Architecture Configurable Logic Block User Guide. https://docs.xilinx.com/v/u/en-US/ug574-ultrascale-clb.
[3] 2019. HLS packet processing. https://github.com/Xilinx/HLS_packet_processing.
[4] 2020. Checksum Offload. https://www.kernel.org/doc/html/latest/networking/checksum-offloads.html.
[5] 2020. Receive Side Scaling. https://www.kernel.org/doc/Documentation/networking/scaling.txt.
[6] 2020. Vivado Design Suite User Guide:High-Level Synthesis. https://www.xilinx.com/support/documentation-navigation/design-hubs/2020-2/dh0090-vitis-hls-hub.html.
[7] 2020. Vivado Design Tools. https://www.xilinx.com/products/design-tools/vivado.html.
[8] 2022. ld-linux.so dynamic linker/loader. https://man7.org/linux/man-pages/man8/ld-linux.so.8.html.
[9] 2022. Microsoft SQL Server Configuration. https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/configure-the-user-connections-server-configuration-option?view=sql-server-ver15.
[10] 2022. NVIDIA ConnectX-5 Infiniband Adapter Card. https://nvdam.widen.net/s/pkxbnmbgkh/networking-infiniband-datasheet-connectx-5-2069273.
[11] 2023. Intel Data Direct I/O Technology. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html.
[12] 2023. Nginx HTTP Web Server. https://nginx.org/en.
[13] 2023. NS3 Network Simulator. https://www.nsnam.org.
[14] 2023. Quotas and constraints for Amazon RDS. https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Limits.html.
[15] 2023. TCP Offload Engine. https://www.chelsio.com/nic/tcp-offload-engine
[16] 2023. Xilinx Alveo U280 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.
[17] Dennis Abts and Bob Felderman. 2012. A Guided Tour of Datacenter Networking. *Communications of the ACM – ACM Queue* 55, number 6 (2012), 44–51. http://dx.doi.org/10.1145/2184319.2184335 DOI:10.1145/2184319.2184335.
[18] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling programmable transport protocols in high-speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 93–109.
[19] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. https://doi.org/10.1145/1879141.1879175. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement* (Melbourne, Australia) *(IMC '10)*. Association for Computing Machinery, New York, NY, USA, 267–280. https://doi.org/10.1145/1879141.1879175
[20] L.S. Brakmo and L.L. Peterson. 1995. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on Selected Areas in Communications* 13, 8 (1995), 1465–1480. https://doi.org/10.1109/49.464716
[21] Glenn William Connery, William Paul Sherer, Gary Jaszewski, and James S. Binder. 1999. Offload of TCP Segmentation to a Smart Adapter.
[22] Jonathna Corbet. 2009. Generic Receive Offload. https://lwn.net/Articles/358910/.
[23] Jonathan Corbet. 2016. BBR congestion control. https://lwn.net/Articles/701165/.
[24] Jake Edge. 2019. The TCP SACK panic. https://lwn.net/Articles/791409/.
[25] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 51–66.
[26] Will Glozer. 2013. WRK. https://github.com/wg/wrk.
[27] Andrei Gurtov, Tom Henderson, Sally Floyd, and Yoshifumi Nishida. 2012. The NewReno Modification to TCP's Fast Recovery Algorithm. https://rfc-editor.org/rfc/rfc6582.txt. https://doi.org/10.17487/RFC6582
[28] Chung-Hsing Hsu and Ulrich Kremer. 1998. IPERF: A framework for automatic construction of performance prediction models. In *Workshop on Profile and Feedback-Directed Compilation (PFDC), Paris, France*. Citeseer.
[29] Intel. 2023. Intel DPDK: Data Plane Development Kit. https://www.dpdk.org/.
[30] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mtcp: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 489–502.
[31] JSOF Research Lab. 2020. Ripple20: 19 Zero-Day Vulnerabilities Amplified by the Supply Chain. https://www.jsof-tech.com/disclosures/ripple20/.
[32] Ilias Marinos, Robert N.M. Watson, and Mark Handley. 2014. Network Stack Specialization for Performance. *SIGCOMM Comput. Commun. Rev.* 44, 4 (aug 2014), 175–186. https://doi.org/10.1145/2740070.2626311
[33] Jeffrey C. Mogul. 2003. TCP Offload Is a Dumb Idea Whose Time Has Come. https://www.usenix.org/conference/hotos-ix/tcp-offload-dumb-idea-whose-time-has-come. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*. USENIX Association, Lihue, HI.
[34] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. Acceltcp: Accelerating network applications with stateful TCP offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 77–92.
[35] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2018. Restructuring Endpoint Congestion Control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 30–43. https://doi.org/10.1145/3230543.3230553
[36] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 327–341. https://doi.org/10.1145/3230543.3230560
[37] Larry L Peterson and Bruce S Davie. 2007. *Computer Networks: A Systems Approach*. Elsevier.
[38] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. 2021. Autonomous NIC Offloads. https://doi.org/10.1145/3445814.3446732. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA). Association for Computing Machinery, New York, NY, USA, 18–35. https://doi.org/10.1145/3445814.3446732
[39] David Plummer. 1982. An Ethernet Address Resolution Protocol. https://www.rfc-editor.org/rfc/rfc826.txt. https://doi.org/10.17487/RFC0826
[40] Jon Postel. 1981. Internet Control Message Protocol. https://www.rfc-editor.org/rfc/rfc792.txt. https://doi.org/10.17487/RFC0792
[41] Jon Postel. 1981. Transmission Control Protocol. https://www.rfc-editor.org/rfc/rfc793.txt. https://doi.org/10.17487/RFC0793
[42] Injong Rhee, Lisong Xu, Sangtae Ha, Alexander Zimmermann, Lars Eggert, and Richard Scheffenegger. 2018. CUBIC for Fast Long-Distance Networks. https://rfc-editor.org/rfc/rfc8312.txt. https://doi.org/10.17487/RFC8312
[43] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. 2015. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 123–137.
[44] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio López-Buedo. 2019. Limago: An fpga-based open-source 100 gbe tcp/ip stack. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 286–292.
[45] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable Traffic Shaping at End Hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 404–417. https://doi.org/10.1145/3098822.3098852
[46] Rajath Shashidhara, Timothy Stamler, Antoine Kaufmann, and Simon Peter. 2022. FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism. https://www.usenix.org/conference/nsdi22/presentation/shashidhara. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA.
[47] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. 2015. Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 36–43. https://doi.org/10.1109/FCCM.2015.12
[48] Brandon Wilson. 2017. Why Are We Deprecating Network Performance Features? https://techcommunity.microsoft.com/t5/core-infrastructure-and-security/why-are-we-deprecating-network-performance-features-kb4014193/ba-p/259053.