AccelTCP: Accelerating Network Applications with Stateful TCP Offloading

YoungGyoun Moon
KAIST

SeungEon Lee
KAIST

Muhammad Asim Jamshed
Intel Labs

KyoungSoo Park
KAIST

Abstract

The performance of modern key-value servers or layer-7 load balancers often heavily depends on the efficiency of the underlying TCP stack. Despite numerous optimizations such as kernel-bypassing and zero-copying, performance improvement with a TCP stack is fundamentally limited due to the *protocol conformance overhead* for compatible TCP operations. Unfortunately, the protocol conformance overhead amounts to as large as 60% of the entire CPU cycles for short-lived connections or degrades the performance of L7 proxying by 3.2x to 6.3x.

This work presents AccelTCP, a hardware-assisted TCP stack architecture that harnesses programmable network interface cards (NICs) as a TCP protocol accelerator. AccelTCP can offload complex TCP operations such as connection setup and teardown completely to NIC, which simplifies the host stack operations and frees a significant amount of CPU cycles for application processing. In addition, it supports running connection splicing on NIC so that the NIC relays all packets of the spliced connections with zero DMA overhead. Our evaluation shows that AccelTCP enables short-lived connections to perform comparably to persistent connections. It also improves the performance of Redis, a popular in-memory key-value store, and HAProxy, a widely-used layer-7 load balancer, by 2.3x and 11.9x, respectively.

1 Introduction

Transmission Control Protocol (TCP) [24] is undeniably the most popular protocol in modern data networking. It guarantees reliable data transfer between two end-points without overwhelming either end-point nor the network itself. It has become ubiquitous as it simply requires running on the Internet Protocol (IP) [23] that operates on almost every physical network.

Ensuring the desirable properties of TCP, however, often entails a severe performance penalty. This is especially pronounced with the recent trend that the gap between CPU capacity and network bandwidth widens. Two notable scenarios where modern TCP servers suffer from poor performance are handling short-lived connections and layer-7 (L7) proxying. Short-lived connections incur a serious overhead in processing small control packets while an L7 proxy requires large compute cycles and memory bandwidth for relaying packets between two connections. While recent kernel-bypass TCP stacks [5, 30, 41, 55, 61] have substantially improved the performance of short RPC transactions, they still need to track flow states whose computation cost is as large as 60% of the entire CPU cycles (Section §2). An alternative might be to adopt RDMA [37, 43] or a custom RPC protocol [44], but the former requires an extra in-network support [7, 8, 70] while the latter is limited to closed environments. On the other hand, an application-level proxy like L7 load balancer (LB) may benefit from zero copying (e.g., via the `splice()` system call), but it must perform expensive DMA operations that would waste memory bandwidth.

The root cause of the problem is actually clear – the TCP stack must maintain mechanical protocol conformance regardless of what the application does. For instance, a key-value server has to synchronize the state at connection setup and closure even when it handles only two data packets for a query. An L7 LB must relay the content between two separate connections even if its core functionality is determining the back-end server.

AccelTCP addresses this problem by exploiting modern network interface cards (NICs) as a TCP protocol accelerator. It presents a dual-stack TCP design that splits the functionality between a host and a NIC stack. The host stack holds the main control of all TCP operations;

it sends and receives data reliably from/to applications and performs control-plane operations such as congestion and flow control. In contrast to existing TCP stacks, however, it accelerates TCP processing by selectively offloading *stateful* operations to the NIC stack. Once offloaded, the NIC stack processes connection setup and teardown as well as connection splicing that relays packets of two connections entirely on NIC. The goal of AccelTCP is to extend the performance benefit of traditional NIC offload to short-lived connections and application-level proxying while being complementary to existing offloading schemes.

Our design brings two practical benefits. First, it significantly saves the compute cycles and memory bandwidth of the host stack as it simplifies the code path. Connection management on NIC simplifies the host stack as the host needs to keep only the established connections as well as it avoids frequent DMA operations for small control packets. Also, forwarding packets of spliced connections directly on NIC eliminates DMA operations and application-level processing. This allows the application to spend precious CPU cycles on its main functionality. Second, the host stack makes an offloading decision flexibly on a per-flow basis. When an L7 LB needs to check the content of a response of select flows, it opts them out of offloading while other flows still benefit from connection splicing on NIC. When the host stack detects overload of the NIC, it can opportunistically reduce the offloading rate and use the CPU instead.

However, performing stateful TCP operations on NIC is non-trivial due to following challenges. First, maintaining consistency of transmission control blocks (TCBs) across host and NIC stacks is challenging as any operation on one stack inherently deviates from the state of the other. To address the problem, AccelTCP always transfers the ownership of a TCB along with an offloaded task. This ensures that a single entity solely holds the ownership and updates its state at any given time. Second, stateful TCP operations increase the implementation complexity on NIC. AccelTCP manages the complexity in two respects. First, it exploits modern smart NICs equipped with tens of processing cores and a large memory, which allows flexible packet processing with C and/or P4 [33]. Second, it limits the complexity by resorting to a stateless protocol or by cooperating with the host stack. As a result, the entire code for the NIC stack is only 1,501 lines of C code and 195 lines of P4 code, which is small enough to manage on NIC.

Our evaluation shows that AccelTCP brings an enormous performance gain. It outperforms mTCP [41] by 2.2x to 3.8x while it enables non-persistent connections to perform comparably to persistent connections on IX [30] or mTCP. AccelTCP's connection splicing offload achieves a full line rate of 80 Gbps for L7 proxying of 512-byte messages with only a single CPU core. In terms of real-world applications, AccelTCP improves the performance of Redis [17] and HAProxy [6] by a factor of 2.3x and 11.9x, respectively.

The contribution of our work is summarized as follows. (1) We quantify and present the overhead of TCP protocol conformance in short-lived connections and L7 proxying. (2) We present the design of AccelTCP, a dual-stack TCP processing system that offloads select features of stateful TCP operations to NIC. We explain the rationale for our target tasks of NIC offload, and present a number of techniques that reduce the implementation complexity on smart NIC. (3) We demonstrate a significant performance benefit of AccelTCP over existing kernel-bypass TCP stacks like mTCP and IX as well as the benefit to real-world applications like a key-value server and an L7 LB.

2 Background and Motivation

In this section, we briefly explain the need for an NIC-accelerated TCP stack, and discuss our approach.

2.1 TCP Overhead in Short Connections & L7 Proxying

Short-lived TCP connections are prevalent in data centers [31, 65] as well as in wide-area networks [54, 64, 66]. L7 proxying is also widely used in middlebox applications such as L7 LBs [6, 36] and application-level gateways [2, 19]. Unfortunately, application-level performance of these workloads is often suboptimal as the majority of CPU cycles are spent on TCP stack operations. To better understand the cost, we analyze the overhead of the TCP stack operations in these workloads. To avoid the inefficiency of the kernel stack [38, 39, 60], we use mTCP [41], a scalable user-level TCP stack on DPDK [10], as our baseline stack for evaluation. We use one machine for a server (or proxy) and four clients and four back-end servers, all equipped with a 40GbE NIC. The detailed experimental setup is in Section §6.

Small message transactions: To measure the overhead of a short-lived TCP connection, we compare the performance of non-persistent vs. persistent connections with a large number of concurrent RPC transactions. We spawn 16k connections where each transaction exchanges one small request and one small response (64B) between a client and a server. A non-persistent connection performs only a single transaction while a persistent connection repeats the transactions without

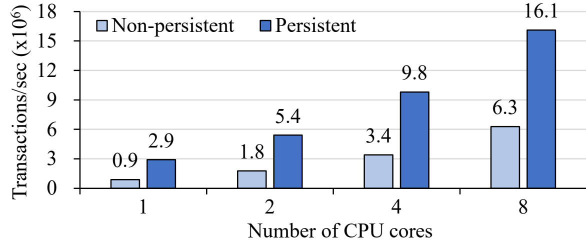


Figure 1: Small packet (64B) performance with non-persistent and persistent connections

a closure. To minimize the number of small packets, we patch mTCP to piggyback every ACK on the data packet.

Figure 1 shows that persistent connections outperform non-persistent connections by 2.6x to 3.2x. The connection management overhead is roughly proportional to the number of extra packets that it handles; two packets per transaction with a persistent connection vs. six [1] packets for the same task with a non-persistent connection. Table 1 shows the breakdown of the CPU cycles where almost 60% of them are attributed to connection setup and teardown. The overhead mainly comes from TCP protocol handling with connection table management, TCB construction and destruction, packet I/O, and L2/L3-level processing of control packets.

Our experiments may explain the strong preference to persistent connections in data centers. However, not all applications benefit from the persistency. When application data is inherently small or transferred sporadically [32, 69], it would result in a period of inactivity that taxes on server resources. Similarly, persistent connections are often deprecated in PHP applications to avoid the risk of resource misuse [28]. In general, supporting persistent connections is cumbersome and error-prone because the application not only needs to keep track of connection states, but it also has to periodically check connection timeout and terminate idle connections. By eliminating the connection management cost with NIC offload, our work intends to free the developers from this burden to choose the best approach without performance concern.

Application-level proxying: An L7 proxy typically operates by (1) terminating a client connection (2) accepting a request from the client and determining the back-end server with it, and creating a server-side connection, and (3) relaying the content between the client and the back-end server. While the key functionality of an L7 proxy is to map a client-side connection to a back-end server, it consumes most of CPU cycles on relaying the packets between the two connections. Packet

[1] SYN, SYN-ACK, ACK-request, response-FIN, FIN-ACK, and ACK.

Connection setup/ teardown	TCP processing and state update	24.0%	60.5%
	TCP connection state init/destroy	17.2%	
	Packet I/O (control packet)	10.2%	
	L2-L3 processing/forward	9.1%	
Message delivery	TCP processing and state update	11.0%	29.0%
	Message copy via socket buffer	8.4%	
	Packet I/O (data packet)	5.1%	
	L2-L3 processing/forward	4.5%	
Socket/epoll API calls			5.6%
Timer handling and context switching			3.5%
Application logic			1.4%

Table 1: CPU usage breakdown of a user-level TCP echo server (a single 64B packet exchange per connection)

	64B	1500B
L7 proxy (mTCP)	2.1 Gbps	5.3 Gbps
L7 proxy with splice() (mTCP)	2.3 Gbps	6.3 Gbps
L3 forward at host (DPDK)	7.3 Gbps	39.8 Gbps
L3 forward at NIC [2]	28.8 Gbps	40.0 Gbps

Table 2: L7 proxying and L3 forwarding performance on a single CPU core

relaying incurs a severe memory copying overhead as well as frequent context switchings between the TCP stack and the application. While zero-copying APIs like splice() can mitigate the overhead, DMA operations between the host memory and the NIC are unavoidable even with a kernel-bypass TCP stack.

Table 2 shows the 1-core performance of a simple L7 proxy on mTCP with 16k persistent connections (8k connections for clients-to-proxy and proxy-to-backend servers, respectively). The proxy exchanges n-byte (n=64 or 1500) packets between two connections, and we measure the wire-level throughput at clients including control packets. We observe that TCP operations in the proxy significantly degrade the performance by 3.2x to 6.3x compared to simple packet forwarding with DPDK [10], despite using zero-copy splice(). Moreover, DMA operations further degrade the performance by 3.8x for small packets.

Summary: We confirm that connection management and packet relaying consume a large amount of CPU cycles, severely limiting the application-level performance. Offloading these operations to NIC promises a large potential for performance improvement.

2.2 NIC Offload of TCP Features

There have been a large number of works and debates on NIC offloading of TCP features [35, 47, 50, 57]. While AccelTCP pursues the same benefit of saving CPU cycles

[2] All 120 flow-processing cores in Agilio LX are enabled.

and memory bandwidth, it targets a different class of applications neglected by existing schemes.

Partial TCP offload: Modern NICs typically support partial, fixed TCP function offloads such as TCP/IP checksum calculation, TCP segmentation offload (TSO), and large receive offload (LRO). These significantly save CPU cycles for processing large messages as they avoid scanning packet payload and reduce the number of interrupts to handle. TSO and LRO also improve the DMA throughput as they cut down the DMA setup cost required to deliver many small packets. However, their performance benefit is mostly limited to large data transfer as short-lived transactions deal with only a few of small packets.

Full Stack offload: TCP Offload Engine (TOE) takes a more ambitious approach that offloads entire TCP processing to NIC [34, 67]. Similar to our work, TOE eliminates the CPU cycles and DMA overhead of connection management. It also avoids the DMA transfer of small ACK packets as it manages socket buffers on NIC. Unfortunately, full stack TOE is unpopular in practice as it requires invasive modification of the kernel stack and the compute resource on NIC is limited [12]. Also, operational flexibility is constrained as it requires firmware update to fix bugs or to replace algorithms like congestion control or to add new TCP options. Microsoft's TCP Chimney [15] deviates from the full stack TOE as the kernel stack controls all connections while it offloads only data transfer to the NIC. However, it suffers from similar limitations that arise as the NIC implements TCP data transfer (e.g., flow reassembly, congestion and flow control, buffer management). As a result, it is rarely enabled these days [27].

In comparison, existing schemes mainly focus on efficient large data transfer, but AccelTCP targets performance improvement with short-lived connections and L7 proxying. AccelTCP is complementary to existing partial TCP offloads as it still exploits them for large data transfer. Similar to TCP Chimney, AccelTCP's host stack assumes full control of the connections. However, the main offloading task is completely the opposite: AccelTCP offloads connection management while the host stack implements entire TCP data transfer. This design substantially reduces the complexity on NIC while it extends the benefit to an important class of modern applications.

2.3 Smart NIC for Stateful Offload

Smart NICs [1, 3, 14, 25] are gaining popularity as they support flexible packet processing at high speed with programming languages like C or P4 [33]. Recent smart NICs are flexible enough to run Open vSwitch [62], Berkeley packet filter [49], or even key-value lookup [53], often achieving 2x to 3x performance improvement over CPU-based solutions [16]. In this work, we use Netronome Agilio LX as a smart NIC platform to offload stateful TCP operations.

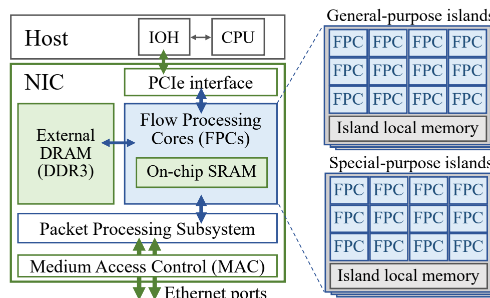


Figure 2: Architecture of SoC-based NIC (Agilio LX)

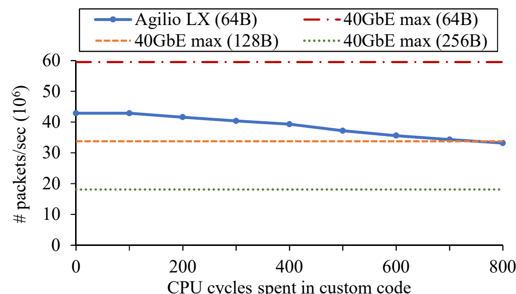


Figure 3: Packet forwarding performance on Agilio LX

As shown in Figure 2, Agilio LX employs 120 flow processing cores (FPCs) running at 1.2GHz. 36 FPCs are dedicated to special operations (e.g., PCI or Interlaken) while remaining 84 FPCs can be used for arbitrary packet processing programmed in C and P4. One can implement the basic forwarding path with a match-action table in P4 and add custom actions that require a fine-grained logic written in C. The platform also provides fast hashing, checksum calculation, and cryptographic operations implemented in hardware.

One drastic difference from general-purpose CPU is that FPCs have multiple layers of non-uniform memory access subsystem – registers and memory local to each FPC, shared memory for a cluster of FPCs called "island", or globally-accessible memory by all FPCs. Memory access latency ranges from 1 to 500 cycles depending on the location, where access to smaller memory tends to be faster than larger ones. We mainly use internal memory (IMEM, 8MB of SRAM) for flow metadata and external memory (EMEM, 8GB of DRAM) for packet contents. Depending on the flow metadata size, IMEM can support up to 128K to 256K concurrent flows. While EMEM would support more flows, it is 2.5x slower. Each FPC

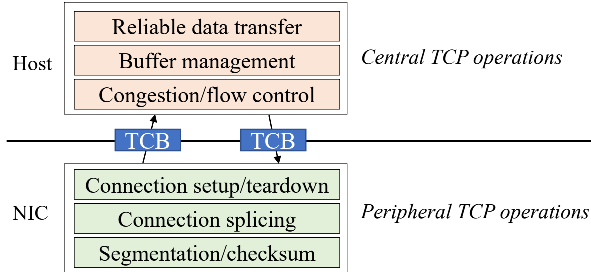


Figure 4: Split of TCP functionality in AccelTCP

can run up to 8 cooperative threads – access to slow memory by one thread would trigger a hardware-based context switch to another, which takes only 2 cycles. This hides memory access latency similarly to GPU.

Figure 3 shows the packet forwarding performance of Agilio LX as a function of cycles spent by custom C code, where L3 forwarding is implemented in P4. We see that it achieves the line rate (40 Gbps) for any packets larger than 128B. However, 64B packet forwarding throughput is only 42.9 Mpps (or 28.8 Gbps) even without any custom code. We suspect the bottleneck lies in scattering and gathering of packets across the FPCs. The performance starts to drop as the custom code spends more than 200 cycles, so minimizing cycle consumption on NIC is critical for high performance.

3 AccelTCP Design Rationale

AccelTCP is a dual-stack TCP architecture that harnesses NIC hardware as a TCP protocol accelerator. So, the primary task in AccelTCP's design is to determine the target for offloading. In this regard, AccelTCP divides the TCP stack operations into two categories: *central* TCP operations that involve application data transfer and *peripheral* TCP operations required for protocol conformance or mechanical operations that can bypass the application logic. Central TCP operations refer to all aspects of application data transfer – reliable data transfer with handling ACKs, inferring loss and packet retransmission, tracking received data and performing flow reassembly, enforcing congestion/flow control, and detecting errors (e.g., abrupt connection closure by a peer). These are typically complex and subject to flexible policies, which demands variable amount of compute cycles. One can optimize them by exploiting flow-level parallelism [5, 30, 41, 59] or by steering the tasks into fast and slow paths [48] on kernel-bypass stacks. However, the inherent complexity makes it a poor fit for NIC offloading as evidenced by the full stack TOE approach.

Peripheral operations refer to the remaining tasks whose operation is logically independent from the application. These include traditional partial NIC offload tasks [3], connection setup and teardown, and blind relaying of packets between two connections that requires no application-level intervention. Peripheral tasks are either stateless operations with a fixed processing cost or lightly stateful operations that synchronize the states for reliable data transfer. We mainly target these operations for offloading as they can be easily separated from the host side that runs applications.

Connection management offload: State synchronization at the boundary of a connection is a key requirement for TCP, but it is a pure overhead from the application's perspective. While NIC offload is logically desirable, conventional wisdom suggests otherwise due to complexity [15, 48]. Our position is that one can tame the complexity on recent smart NICs. First, connection setup operations can be made stateless with SYN-cookies [20]. Second, the common case of connection teardown is simple state transition, and modern smart NICs have enough resources to handle a few exceptions.

Connection splicing offload: Offloading connection splicing to NIC is conceptually complex as it requires state management of two separate connections on NIC. However, if the application does not modify the relayed content, as is often the case with L7 LBs, we can simulate a single logical connection with two physical connections. This allows the NIC to operate as a fast packet forwarder that simply translates the packet header. The compute cycles for this are fixed with a small per-splicing state.

To support the new offload tasks, we structure the dual-stack design with the following guidelines.

1. Full control by the host side: The host side should enforce full control of offloading, and it should be able to operate standalone. This is because the host stack must handle corner cases that cannot benefit from offload. For example, a SYN packet without the timestamp option should be handled by the host stack as SYN-cookie-based connection setup would lose negotiated TCP options (Section §4). Also, the host stack could decide to temporarily disable connection offload when it detects the overload of the NIC.

2. Single ownership of a TCB: AccelTCP offloads stateful operations that require updating the TCB. However, maintaining shared TCBs consistently across two stacks is very challenging. For example, a send buffer may have unacknowledged data along with the last FIN packet. The host stack may decide to deliver all data packets for itself while it offloads the connection teardown to NIC simultaneously. Unfortunately, handling

[3] Such as checksum calculation, TSO, and LRO.

ACKs and retransmission across two stacks require careful synchronization of the TCB. To avoid such a case, AccelTCP enforces an exclusive ownership of the TCB at any given time – either host or NIC stack holds the ownership but not both. In the above case, the host stack offloads the entire data to the NIC stack and forgets about the connection. The NIC stack handles remaining data transfer as well as connection teardown.

3. Minimal complexity on NIC: Smart NICs have limited compute resources, so it is important to minimize complex operations on NIC. A tricky case arises at connection teardown as the host stack can offload data transfer as well. In that case, the host stack limits the amount of data so that the NIC stack avoids congestion control and minimizes state tracking of data packets.

4 AccelTCP NIC Dataplane

In this section, we present the design of AccelTCP NIC stack in detail. Its primary role is to execute three offload tasks requested by the host stack. Each offload task can be enabled independently and the host side can decide which flows to benefit from it. The overall operation of NIC offload is shown in Figure 5.

4.1 Connection Setup Offload

An AccelTCP server can offload the connection setup process completely to the NIC stack. For connection setup offload, the server installs the metadata such as local IP addresses and ports for listening on NIC, and the NIC stack handles all control packets in a three-way handshake. Then, only the established connections are delivered to the host stack.

AccelTCP leverages SYN cookies [20] for stateless handshake on NIC. Stateless handshake enables a more efficient implementation as most smart NICs support fast one-way hashing functions in hardware [1, 3, 14]. When a SYN packet arrives, the NIC stack responds with an SYN-ACK packet whose initial sequence number (ISN) is chosen carefully. The ISN consists of 24 bits of a hash value produced with the input of the 4-tuple of a connection and a nonce, 3 bits of encoded maximum segment size (MSS), and time-dependent 5 bits to prevent replay attacks. When an ACK for the SYN-ACK packet arrives, the NIC stack verifies if the ACK number matches (ISN + 1). If it matches, the NIC stack passes the ACK packet up to the host stack with a special marking that indicates a new connection and the information on negotiated TCP options. To properly handle TCP options carried in the initial SYN, the NIC stack encodes all negotiated options in the TCP Timestamps option [22] of the SYN-ACK packet [9]. Then, the NIC stack can retrieve the information from the TSecr value echoed back with the ACK packet. In addition, we use extra one bit in the timestamp field to differentiate a SYN-ACK packet from other packets. This would allow the NIC stack to bypass ACK number verification for normal packets. The TCP Timestamps option is popular (e.g., enabled on 84% of hosts in a national-scale network [51]), and enabled by default on most OSes, but in case a client does not support it, the NIC stack hands the setup process over to the host stack.

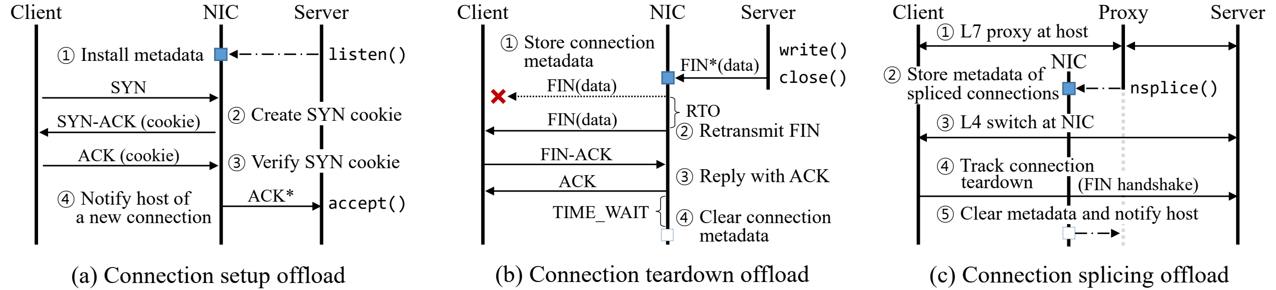
One case where SYN cookies are deprecated is when the server must send the data first after connection setup (e.g., SMTP server). In this case, the client could wait indefinitely if the client-sent ACK packet is lost as the SYN-ACK packet is never retransmitted. Such applications should disable connection setup offload and have the host stack handle connection setup instead.

4.2 Connection Teardown Offload

The application can ask for offloading connection teardown on a per-flow basis. If the host stack decides to offload connection teardown, it hands over the ownership of the TCB and remaining data in the send buffer to the NIC stack. Then, the host stack removes the flow entry from its connection table, and the NIC stack continues to handle the teardown.

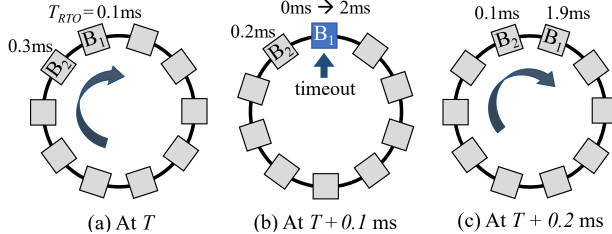
Connection teardown offload is tricky as it must maintain per-flow states while it should ensure reliable delivery of the FIN packet with the offloaded data. To minimize the complexity, the host stack offloads connection teardown only when the following conditions are met. First, the amount of remaining data should be smaller than the send window size. This would avoid complex congestion control on NIC while it still benefits most short-lived connections. [4] Second, if the application wants to confirm data delivery at close(), the host stack should handle the connection teardown by itself. For example, an application may make close() to block until all data is delivered to the other side (e.g., *SO_LINGER* option). In that case, processing the teardown at the host stack is much simpler as it needs to report the result to the application. Fortunately, blocking close() is rare in busy TCP servers as it not only kills the performance, but a well-designed application-level protocol may avoid it. Third, the number of offloaded flows should not exceed a threshold, determined by available memory size on NIC. For each connection teardown, the host stack first checks the number of connection closures being handled by the NIC, and the host stack carries out the connection teardown if the number exceeds the threshold.

[4] RFC 6928 [21] suggests 10 MSS as the initial window size.



(a) Connection setup offload (b) Connection teardown offload (c) Connection splicing offload

Figure 5: AccelTCP NIC offload (We show only active close() by server for (b), but it also supports passive close().)



(a) At T (b) At $T+0.1$ ms (c) At $T+0.2$ ms

Figure 6: Timer bitmap wheel for RTO management on NIC. T_{RTO} represents the remaining time until retransmission.

The NIC stack implements the teardown offload by extending the TSO mechanism. On receiving the offload request, it stores a 26-byte flow state [5] at the on-chip SRAM (e.g., 8MB of IMEM), segments the data into TCP packets, and send them out. Then, it stores the entire packets at the off-chip DRAM (e.g., 8GB of EMEM) for potential retransmission. This would allow tracking over 256k concurrent flows being closed on NIC.

Timeout management: The teardown process requires timeout management for packet retransmission and for observing a timeout in the TIME_WAIT state. AccelTCP uses three duplicate ACKs and expiration of retransmission timeout (RTO) as a trigger for packet retransmission. For teardown offload, however, RTO is the main mechanism as the number of data packets is often too small for three duplicate ACKs. Also, any side that sends the FIN first would end up in the TIME_WAIT state for a timeout. A high-performance server typically avoids this state by having the clients initiate the connection closure, but sometimes it is inevitable. AccelTCP supports the TIME_WAIT state, but it shares the same mechanism as RTO management for the timer.

Unfortunately, an efficient RTO implementation on NIC is challenging. For multicore CPU systems, a list or a hash table implementation would work well as each CPU core handles only its own flows affinitized to it without a lock. However, smart NICs often do not guarantee flow-core affinity, so a list-based implementation would incur huge lock contention with many processor cores.

We observe that RTO management is write-heavy as each offloaded flow (and each packet transmission) would register for a new RTO. Thus, we come up with a data structure called *timer bitmap wheel*, which allows concurrent updates with minimal lock contention. It consists of N timer bitmaps where each bitmap is associated with a distinct timeout value. The time interval between two neighboring timer bitmaps is fixed (e.g., 100 us for Figure 6). When one time interval elapses, all bitmaps rotate in the clockwise direction by one interval, like Figure 6-(b). Bitmap rotation is efficiently implemented by updating a pointer to the RTO-expired bitmap every time interval. Each timer bitmap records all flows with the same RTO value, where the location of a bit represents a flow id (e.g., n-th bit in a bitmap refers to a flow id, n). When the RTO of a timer bitmap expires, all flows in the bitmap retransmit their unacknowledged packets. From the location of each bit that is set, one can derive the corresponding flow id and find the pointer to its flow state that holds all the metadata required for retransmission. Then, all bits in the bitmap are reset to zero and its RTO is reset to (N x (time interval)). RTO-expired flows register for a new RTO. When an ACK for the FIN of a flow arrives, the flow is removed from its RTO bitmap. One can implement an RTO larger than the maximum by keeping a counter in the flow state that decrements every expiration of the maximum RTO.

The timer bitmap wheel allows concurrent updates by multiple flows as long as their flow ids belong to different 32-bit words in the bitmap. Only the flows whose ids share the same 32-bit word contend for a lock for access. On the down side, it exhibits two overheads: memory space for bitmaps and bitmap scanning at RTO expiration. The memory consumption is not a big concern as it requires only 8KB for each bitmap for 64k concurrent flows being closed. We reduce the scanning overhead by having multiple cores scan a different bitmap region in parallel. Keeping a per-region counter might further

[5] a 4-tuple of the connection, TCP state, expected sequence and ACK numbers, and current RTO.

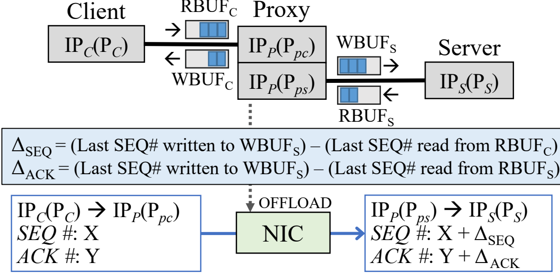


Figure 7: Connection splicing on NIC dataplane. IP_C, IP_P, IP_S: IP addresses of client, proxy, and server, P_C, P_S: port numbers of client and server, P_{pc}, P_{ps}: port numbers of proxy for the client side and the server side, $RBUF_C$, $RBUF_S$: read buffers on each side, $WBUF_C$, $WBUF_S$: write buffers on each side.

reduce the scanning overhead, but we find that the cost for counter update is too expensive even with atomic increment/decrement.

4.3 Connection Splicing Offload

Connection splicing offload on NIC allows zero-DMA data transfer. The key idea is to simulate a single connection by exploiting the NIC as a simple L4 switch that translates the packet header. An L7 proxy can ask for connection splicing on NIC if it no longer wants to relay packets of the two connections in the application layer. On a splicing offload request, the host stack hands over the states of two connections to NIC, and removes their TCBs from its connection table. The NIC stack takes over the ownership, and installs two L4 forwarding rules for relaying packets. The host stack keeps track of the number of spliced connections offloaded to NIC, and decides whether to offload more connections by considering the available memory on NIC.

Figure 7 shows the packet translation process. It simply swaps the 4-tuples of two connections and translates the sequence/ACK numbers and TCP/IP checksums of a packet with pre-calculated offsets. While the Figure assumes that the proxy does not modify any content, but one can easily support such a case. For example, if a proxy modifies request or response headers before splicing, the host stack only needs to reflect the extra delta in sequence and ACK numbers into the pre-calculated offsets. One limitation in our current scheme is that the proxy may not read or modify the packets any more after splicing offload.

Efficient TCP/IP checksum update: Translating a packet header requires TCP/IP checksum update. However, recalculating the TCP checksum is expensive as it scans the entire packet payload. To avoid the overhead, AccelTCP adopts *differential checksum update*, which exploits the fact that the one's complement addition is

1. On splicing offload for a flow from $IP_C(P_C)$ to $IP_S(P_S)$:
2. $\quad CSO_{IP} \leftarrow IP_S + IP_C$
3. $\quad CSO_{TCP} \leftarrow CSO_{IP} + P_S + P_{ps} - P_C - P_{pc} + \Delta_{SEQ} + \Delta_{ACK}$
4. $\quad$ **Store** CSO_{IP} and CSO_{TCP}
5. For any next incoming packets from $IP_C(P_C)$ to $IP_S(P_S)$:
6. $\quad$ **Load** CSO_{IP} and CSO_{TCP}
7. $\quad CS_{IP} \leftarrow CS_{IP} + CSO_{IP}$
8. $\quad CS_{TCP} \leftarrow CS_{TCP} + CSO_{TCP}$
9. $\quad$ If $(SEQ\ \#) > (-\Delta_{SEQ})$, then $CS_{TCP} \leftarrow CS_{TCP} - 1$
10. $\quad$ If $(ACK\ \#) > (-\Delta_{ACK})$, then $CS_{TCP} \leftarrow CS_{TCP} - 1$

Figure 8: Differential checksum update. CSO: checksum offset, CS: checksum. Other notations are in Figure 7. Note that + and – indicate 1's complement addition and subtraction.

both associative and distributive. Since only the 4-tuple of a connection and sequence and ACK numbers are updated, we only need to add the difference (or offset) of these values to the checksum. Figure 8 shows the algorithm. Upon splicing offload request, the NIC stack pre-calculates the offsets for IP and TCP checksums, respectively (Line 2-4). For each packet for translation, it adds the offsets to IP and TCP checksums, respectively (Line 7-8). One corner case arises if a sequence or an ACK number wraps around. In that case, we need to subtract 1 from the checksum to conform to 1's complement addition (Line 9-10).

Tracking teardown state: Since connection splicing operates by merging two connections into one, the NIC stack only needs to passively monitor connection teardown by the server and the client. When the spliced connection closes completely or if it is reset by any peer, the NIC stack removes the forwarding rule entries, and notifies the host stack of the closure. This allows reusing TCP ports or tracking connection statistics at the host.

5 AccelTCP Host Stack

The AccelTCP host stack is logically independent of the NIC stack. While our current implementation is based on mTCP [41], one can extend any TCP stack to harness our NIC offloading.

5.1 Socket API Extension

AccelTCP allows easy porting of existing applications by reusing the epoll()-based POSIX-like socket API of mTCP. In addition, it extends the API to support flexible NIC offloading as shown in Figure 9. First, AccelTCP adds extra socket options to `mtcp_setsockopt()` to enable connection setup and teardown offload to NIC. Note that the connection teardown offload request is advisory, so the host stack can decide not to offload the closure if the conditions are not met (Section §4.2). Second, AccelTCP adds `mtcp_nsplice()` to initiate splicing two connections on NIC. The host stack waits until all data

```
/* enable/disable setup and teardown offload
   - level  : IPPPROTO_TCP
   - optname: TCP_SETUP_OFFLOAD or TCP_TEARDOWN_OFFLOAD
   - optval : 1 (enable) or 0 (disable) */
int mtcp_setsockopt(mctx_t m, int sock, int level, int optname,
                    void *optval, socklen_t optlen);

/* offload connection splicing of two connections */
int mtcp_nsplice(mctx_t m, int sock_c, int sock_s, callback_t* cb);

/* notified upon a closure of spliced connections */
typedef void (*callback_t)(nsplice_meta_t * meta);
```

Figure 9: Socket API extension for AccelTCP

in the send buffer are acknowledged while buffering any incoming packets. Then, it installs forwarding rules onto NIC, sending the buffered packets after header translation. After calling this function, the socket descriptors should be treated as if they are closed in the application. Optionally, the application may specify a callback function to be notified when the spliced connections finish. Through the callback function, AccelTCP provides (i) remote addresses of the spliced connections, (ii) the number of bytes transferred after offloaded to NIC dataplane, and (iii) how the connections are terminated (e.g., normal teardown or reset by any peer).

5.2 Host Stack Optimizations

We optimize the host networking stack to accelerate small message processing. While these optimizations are orthogonal to NIC offload, they bring a significant performance benefit to short-lived connections.

Lazy TCB creation: A full TCB of a connection ranges from 400 to 700 bytes even on recent implementations [5, 41]. However, we find that many of the fields are unnecessary for short-lived connections whose message size is smaller than the initial window size. To avoid the overhead of a large TCB, AccelTCP creates the full TCB only when multiple transactions are observed. Instead, the host stack creates a small quasi-TCB (40 bytes) for a new connection. If the application closes the connection after a single write, the host stack offloads the teardown and destroys the quasi-TCB.

Opportunistic zero-copy: Recent high-performance TCP stacks [30, 61, 68] bypass the socket buffer to avoid extra memory copying. However, this often freezes the application-level buffer even after sending data, or overflows the host packet buffers if the application does not read the packets in a timely manner. AccelTCP addresses this problem by opportunistically performing a zero-copy I/O. When a stream of packets arrive in order, the application waiting for a read event will issue a read call. Then, the content of the packets is copied directly to the application buffer while any leftover is written to the receive socket buffer. When an application sends data on an empty socket buffer, the data is directly written to the host packet buffer for DMA'ing to NIC. Only when the host packet buffer is full, the data is written to the send socket buffer. Our scheme observes the semantics of standard socket operations, allowing easy porting of existing applications. Yet, this provides the benefit of zero-copying to most short-lived connections.

User-level threading: mTCP spawns two kernel-level threads: a TCP stack thread and an application thread on each CPU core. While this allows independent operations of the TCP thread (e.g., timer operations), it incurs a high context switching overhead. To address the problem, we modify mTCP to use cooperative user-level threading [13]. We find that this not only reduces the context switching overhead, but it also allows other optimizations like lazy TCB creation and opportunistic zero-copying.

6 Evaluation

We evaluate AccelTCP by answering following questions. First, does stateful TCP offloading and host stack optimizations demonstrate a high performance in a variety of workloads? (§6.1) Second, does it deliver the performance benefit to real-world applications? (§6.2) Finally, is the extra cost of a smart NIC justifiable? (§6.3)

Experiment setup: Our experimental setup consists of one server (or a proxy), four clients, and four back-end servers. The server machine has an Intel Xeon Gold 6142 @ 2.6GHz with 128 GB of DRAM and a dual-port Netronome Agilio LX 40GbE NIC (NFP-6480 chipset). Each client has an Intel Xeon E5-2640 v3 @ 2.6GHz, and back-end servers have a mix of Xeon E5-2699 v4 @ 2.2GHz and Xeon E5-2683 v4 @ 2.1GHz. The client and backend server machines are configured with Intel XL710-QDA2 40GbE NICs. All the machines are connected to a Dell Z9100-ON switch, configured to run at 40 GbE speed. For TCP stacks, we compare AccelTCP against mTCP [41] and IX [30]. All TCP stacks employ DPDK [10] for kernel-bypass packet I/O. Clients and back-end servers run mTCP patched to use cooperative user-level threading as AccelTCP. For IX experiments, we use two dual-port Intel X520-DA2 10GbE NICs, and enable all four ports bonded with a L3+L4 hash to balance the load as IX does not support 40GbE NICs. We verify that any single 10GbE port does not become the bottleneck based on port-level statistics at the switch. Hyperthreading is disabled for mTCP and AccelTCP, and enabled for IX when it improves the performance.

Our current prototype uses CRC32 to generate SYN cookies for connection setup. To prevent state explosion attacks, one needs to use a cryptographic hash function (such as MD5 or SHA2). Unfortunately, the API sup-

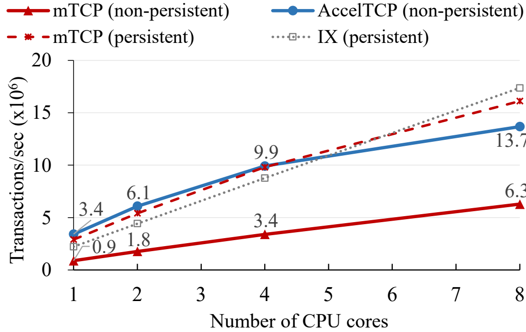


Figure 10: Throughputs of 64B packet transactions

Action	Mtps	Speedup
Baseline (w/o NIC offload)	0.89	1.0x
+ Enable setup offload (§4.1)	1.21	1.4x
+ Enable teardown offload (§4.2)	2.06	2.3x
+ Enable opportunistic TCB creation & opportunistic zero-copy (§5.2)	3.42	3.8x

Table 3: Breakdown of contribution by each optimization on a single CPU core (64B packet transactions)

port for hardware-assisted cryptographic operations in Agilio NICs is currently incomplete (for both C and P4 code), so we use CRC32 instead here.

6.1 Microbenchmark

We evaluate AccelTCP's performance for handling short-lived TCP connections and L7 proxying, and compare against the performance of the state-of-the-art TCP stacks: mTCP [41] and IX [30].

6.1.1 Short-lived Connection Performance

We evaluate the benefit of connection management offload by comparing the performance of TCP echo servers that perform 64B packet transactions with persistent vs. non-persistent connections. The TCP echo servers maintain 16k concurrent connections, and the performance results are averaged over one-minute period for five runs in each experiment. In the non-persistent case, a new connection is created immediately after every connection closure. AccelTCP offloads connection setup and offload to NIC while mTCP handles them using CPU. For IX, we evaluate only the persistent connection case as IX experiences a crash when handling thousands of concurrent connections with normal teardown.

Figure 10 compares the throughputs over varying numbers of CPU cores. AccelTCP achieves 2.2x to 3.8x better throughputs than non-persistent mTCP, comparable to those of persistent connections. Surprisingly, AccelTCP outperforms persistent connections by 13% to 54% for up to four CPU cores. This is because AccelTCP

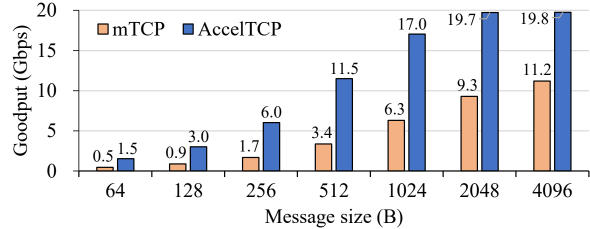


Figure 11: Performance of short-lived connections for varying message sizes on a single CPU core

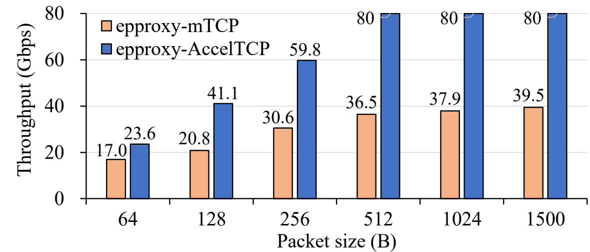


Figure 12: Comparison of L7 proxying throughputs

benefits from lazy TCB creation (§5.2) while persistent connections suffer from a CPU bottleneck. However, its eight-core performance is 22% lower than that of persistent IX, implying a bottleneck on NIC. Overall, connection management offload brings a significant performance benefit, which enables short-lived connections to perform comparably to persistent connections.

Table 3 shows the breakdown of performance in terms of the contribution by each optimization. We find that connection setup and teardown offload improve the baseline performance by 2.3x while other host stack optimizations contribute by extra 1.5x.

Figure 11 compares the goodputs over varying message sizes on a single CPU core. AccelTCP maintains the performance benefit over different message sizes with a speedup of 2.5x to 3.6x. The performance of messages larger than one MSS is limited at 20 Gbps, which seems impacted by our software TSO implementation on NIC. The current Agilio NIC SDK does not provide an API to exploit hardware TSO for programmable dataplane. We believe the single core performance would further improve with proper hardware support.

6.1.2 Layer-7 Proxing Performance

We now evaluate connection splicing offload with a simple L7 LB called epproxy that inspects the initial request, determines a back-end server, and relays the content between the client and the server. We measure the wire-level, receive-side throughput (including control packets) on the client side over different message sizes. Clients spawn 8k concurrent connections with epproxy, and the proxy creates 8k connections with back-

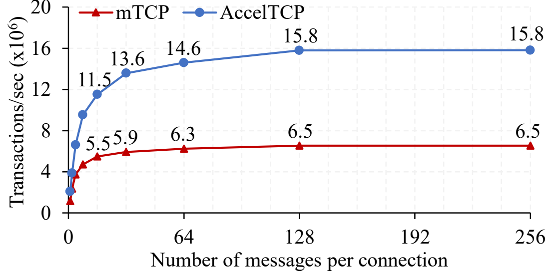


Figure 13: L7 proxying performance over varying numbers of message transactions per connection with 64B packets

end servers. We confirm that both clients and back-end servers are not the bottleneck. We configure epproxy-mTCP to use eight cores while epproxy-AccelTCP uses only a single core as CPU is not the bottleneck. All connections are persistent, and we employ both ports of the Agilio LX NIC here. The NIC is connected to the host via 8 lanes of PCIe-v3 [6].

Figure 12 shows that AccelTCP-proxy outperforms epproxy-mTCP by 1.4x to 2.2x even if the latter employs 8x more CPU cores. We make two observations here. First, the performance of epproxy-AccelTCP reaches full 80 Gbps from 512-byte messages, which exceeds the PCIe throughput of the NIC. This is because epproxy-AccelTCP bypasses host-side DMA and fully utilizes the forwarding capacity of the NIC. Second, epproxy-AccelTCP achieves up to twice as large goodput as the epproxy-mTCP. For example, epproxy-AccelTCP actually performs 2.8x more transactions per second than epproxy-mTCP for 64B messages. This is because AccelTCP splices two connections into a single one while mTCP relays two connections. For each request from a client, epproxy-mTCP must send an ACK as well as a response packet from the back-end server. In contrast, epproxy-AccelTCP replays only the response packet with a piggybacked ACK from the back-end server.

We move on to see if epproxy-AccelTCP fares well on non-persistent connections. Figure 13 shows the performance over varying numbers of message transactions per connection. AccelTCP performs 1.8x better at a single transaction, and the performance gap widens as large as 2.4x at 128 transactions per connection. This confirms that proxying non-persistent connections also benefit from splicing offload of AccelTCP.

6.2 Application Benchmark

We investigate if AccelTCP delivers the performance benefit to real-world applications.

Key-value store (Redis): We evaluate the effectiveness of AccelTCP with Redis (v.4.0.8) [17], a popular in-memory key-value store. We use Redis on mTCP as a baseline server while we port it to use AccelTCP for comparison. We test with the USR workload from Facebook [29], which consists of 99.8% GET requests and 0.2% SET requests with short keys (< 20B) and 2B values. For load generation, we use a Redis client similar to memtier_benchmark [18] written in mTCP. We configure the Redis server and the clients to perform a single key-value transaction for each connection to show the behavior when short-lived connections are dominant.

[6] Theoretical maximum throughput is 63 Gbps according to [58].

	1-core	8-core
Redis-mTCP (kernel thread)	0.19 Mtps	1.38 Mtps
Redis-mTCP (user-level thread)	0.28 Mtps	1.94 Mtps
Redis-AccelTCP	0.44 Mtps	3.06 Mtps

Table 4: Redis performance for short-lived connections

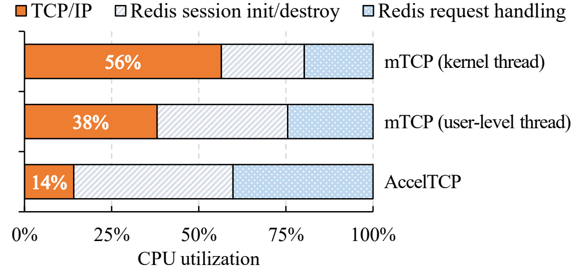


Figure 14: CPU breakdown of Redis on a single CPU core

Table 4 compares the throughputs. Redis-AccelTCP achieves 1.6x to 2.3x better performance than Redis-mTCP, and its performance scales well with the number of CPU cores. Figure 14 shows that mTCP consumes over a half of CPU cycles on TCP stack operations. In contrast, AccelTCP saves up to 75% of the CPU cycles for TCP processing. With AccelTCP, session initialization and destruction of Redis limits the performance. Our investigation reveals that the overhead mostly comes from dynamic memory (de)allocation (`zmalloc()` and `zfree()`) for per-connection metadata, which incurs a severe penalty for handling short-lived connections.

L7 LB (HAProxy): We see if AccelTCP improves the performance of HAProxy (v.1.8.0) [6], a widely used HTTP-based L7 LB. We first port HAProxy to use mTCP and AccelTCP, respectively, and evaluate the throughput with the SpecWeb2009[26]-like workload. The workload consists of static files whose size ranges from 30 to 5,670 bytes with an average file size of 728 bytes. For a fair comparison, we disable any header rewriting in the both version after delivering the first HTTP request. We spawn 8k persistent connections, using simple epoll-based clients and back-end servers running on mTCP. Table 5 compares the throughputs with with 1 core and

	1-core	8-core
HAProxy-mTCP	4.3 Gbps	6.2 Gbps
HAProxy-AccelTCP	73.1 Gbps	73.1 Gbps

Table 5: L7 LB performance for SpecWeb2009-like workload

	E5-2650v2	Gold 6142
mTCP (XL710-QDA2)	1.00	1.25
AccelTCP (Agilio LX)	1.93	1.96

Table 6: Comparison of normalized performance-per-dollar

8 cores. HAProxy-AccelTCP achieves 73.1 Gbps, a 11.9x better throughput than HAProxy-mTCP. The average response time of HAProxy-AccelTCP is 0.98 ms, 13.6x lower than that of HAProxy-mTCP. We observe that the performance benefit is much larger than in Section 6.1.2 because HAProxy has a higher overhead in application-level request processing and packet relaying.

6.3 Cost-effectiveness Analysis

AccelTCP requires a smart NIC, which is about 3-4x more expensive than a normal NIC at the moment. For fairness, we try comparing the cost effectiveness by the performance-per-dollar metric. We draw hardware prices from Intel [11] and Colfax [4] pages (as of August 2019), and use the performance of 64B packet transactions on short-lived connections. Specifically, we compare the performance-per-dollar with a system that runs mTCP with a commodity NIC (Intel XL710-QDA2, $440) vs. another system that runs AccelTCP with a smart NIC (Agilio LX, $1,750). For CPU, we consider Xeon E5-2650v2 ($1,170) and Xeon Gold 6142 ($2,950). For simplicity, we only consider CPU and NIC as hardware cost. Table 6 suggests that NIC offload with AccelTCP is 1.6x to 1.9x more cost-effective, and the gap would widen further if we add other fixed hardware costs.

7 Related Work

Kernel-bypass TCP stacks: Modern kernel-bypass TCP stacks such as mTCP [41], IX [30], SandStorm [55], F-Stack [5] deliver high-performance TCP processing of small message transactions. Most of them employ a fast user-level packet I/O [10], and exploit high parallelism on multicore systems by flow steering on NIC. More recently, systems like ZygOS [63], Shinjuku [42], and Shenango [59] further improve kernel-bypass stack by reducing the tail latency, employing techniques like task stealing, centralized packet distribution, and dynamic core reallocation. We believe that these works are largely orthogonal but complementary to our work as AccelTCP would enhance these stacks by offloading connection management tasks to NIC.

NIC offload: Existing TCP offloads mostly focus on improving large message transfer either by offloading the whole TCP stack [50] or by selectively offloading common send-receive operations [46]. In contrast, our work focuses on connection management and proxying whose performance is often critical to modern network workloads, while we intentionally avoid the complexity of application data transfer offloading. UNO [52] and Metron [45] strive to achieve optimal network function (NF) performance with NIC offload based on runtime traffic statistics. We plan to explore dynamic offloading of a subset of networking stack features (or connections) in response to varying load in the future. To offload TCP connection management, any L2-L4 NFs that should run prior to TCP stack (e.g., firewalling or host networking) must be offloaded to NIC accordingly. Such NFs can be written in P4 [40, 45, 56] and easily integrated with AccelTCP by properly placing them at ingress/egress pipelines of the NIC dataplane.

L7 proxing and short RPCs: Our connection splicing is inspired by the packet tunneling mechanism of Yoda L7 LB [36]. However, Yoda operates as a packet-level translator without a TCP stack, so it cannot modify any of relayed content. In contrast, an AccelTCP application can initiate the offload after any content modification. Also, AccelTCP packet translation runs on NIC hardware, promising better performance. Finally, we note that eRPC [44] achieves 5.0 Mtps RPC performance (vs. 3.4 Mtps of AccelTCP) on a single core. However, ePRC is limited to data center environments while AccelTCP is compatible to TCP and accommodates any TCP clients.

8 Conclusion

In this paper, we have presented AccelTCP that harnesses modern programmable NICs as a TCP protocol accelerator. Drawing the lessons from full stack TOE, AccelTCP's design focuses on minimizing the interaction with the host stack by offloading only select features of stateful TCP operations. AccelTCP manages the complexity on NIC by stateless handshake, single ownership of a TCB, and conditional teardown offload. In addition, it simplifies connection splicing by efficient packet header translation. We have also presented a number of optimizations that significantly improve the host stack.

We have demonstrated that AccelTCP brings a substantial performance boost to short-message transactions and L7 proxying. AccelTCP delivers a 2.3x speedup to Redis on a kernel-bypass stack while it improves the performance of HAProxy by a factor of 11.9. AccelTCP is available at `https://github.com/acceltcp`, and we hope our effort will renew the interest in selective NIC offload of stateful TCP operations.

Acknowledgments

We would like to thank our shepherd Andrew Moore and anonymous reviewers of NSDI 2020 for their insightful comments and feedback on the paper. We also thank Ilwoo Park for applying user-level threading to the mTCP stack. This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grants funded by the Korea government (MSIT) (2018-0-00693, Development of an ultra low-latency user-level transfer protocol and 2016-0-00563, Research on Adaptive Machine Learning Technology Development for Intelligent Autonomous Digital Companion) as well as the SK-Hynix grant for Storage Media Solutions Center.

Appendix A. Host-NIC Communication Interface

The host and NIC stacks communicate with each other by piggybacking control information in the normal packets most time. It encodes the type of offload as unused EtherType values in the Ethernet header, and tags along other information in the special header between the Ethernet and IP headers.

Connection setup: When an application listens on a socket whose setup offload option is enabled, the host stack sends a special control packet to NIC, carrying the listening address/port and TCP options that must be delivered to the remote host during connection setup (e.g., MSS, Window scale factor, Selective ACK, etc.). To notify a new connection, the NIC stack sets the Ethertype of the ACK packet to 0x090A, and delivers the negotiated options in the TCP Timestamps option. The host stack extracts only the TCP options, and ignores the NIC-generated timestamp value.

Connection teardown: For teardown offload, the host stack creates a TSO packet that holds all remaining data in the send buffer, and sets the EtherType to 0x090B. It also encodes other information such as MSS (2 bytes), current RTO (4 bytes), and current TCP state (2 bytes) in the special header area. The NIC stack notifies the host stack of the number of connections being closed on NIC by either sending a control packet or tagging at any packet delivered to host.

Connection splicing: For splicing offload, the host stack uses 0x090C as EtherType, and writes the sequence and ACK number offsets (4 bytes each), and a 4-tuple of a connection in the special header. When the splicing offload packet is passed to the NIC stack, a race condition may arise if some packets in the flows are passed up to the host stack at the same time. To ensure correct forwarding, the host stack keeps the connection entries until it is notified that the splicing rules are installed at NIC. For reporting a closure of spliced connections, NIC creates a special control packet holding the connection information and traffic statistics with the EtherType, 0x090D, and sends it up to the host stack. By monitoring those control packets, the host stack can keep track of the number of active spliced connections on NIC.

References

[1] Agilio® LX 2x40GbE SmartNIC. https://www.netronome.com/m/documents/PB_Agilio_LX_2x40GbE.pdf. Accessed: 2019-08-27.

[2] Amazon API Gateway. https://aws.amazon.com/api-gateway/. Accessed: 2019-08-27.

[3] Cavium LiquidIO® II Network Appliance Smart NIC. https://www.marvell.com/documents/konmn48108xfxalr96jk/. Accessed: 2019-08-27.

[4] Colfax Direct. https://colfaxdirect.com. Accessed: 2019-08-27.

[5] F-Stack | High Performance Network Framework Based on DPDK. http://www.f-stack.org/. Accessed: 2019-08-27.

[6] HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer. http://www.haproxy.org/. Accessed: 2019-08-27.

[7] IEEE 802.1Qau – Congestion Notification. https://1.ieee802.org/dcb/802-1qau/. Accessed: 2019-08-27.

[8] IEEE 802.1Qbb - Priority-based Flow Control. https://1.ieee802.org/dcb/802-1qbb/. Accessed: 2019-08-27.

[9] Improving Syncookies. https://lwn.net/Articles/277146/. Accessed: 2019-08-27.

[10] Intel DPDK: Data Plane Development Kit. http://dpdk.org/. Accessed: 2019-08-27.

[11] Intel Product Specification. https://ark.intel.com. Accessed: 2019-08-27.

[12] Linux and TCP Offload Engines. https://lwn.net/Articles/148697/. Accessed: 2019-08-27.

[13] lthread: Multicore / Multithread Coroutine Library. https://lthread.readthedocs.io. Accessed: 2019-08-27.

[14] Mellanox BlueField™ SmartNIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf. Accessed: 2019-08-27.

[15] Microsoft Windows Scalable Networking Initiative. http://download.microsoft.com/download/5/b/5/5b5bec17-ea71-4653-9539-204a672f11cf/scale.doc. Accessed: 2019-08-27.

[16] Open vSwitch Offload and Acceleration with Agilio SmartNICs. https://www.netronome.com/m/redactor_files/WP_OVS_Benchmarking.pdf. Accessed: 2019-08-27.

[17] Redis. https://redis.io/. Accessed: 2019-08-27.

[18] RedisLabs/memtier_benchmark: NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark. Accessed: 2019-08-27.

[19] RFC 2663. https://tools.ietf.org/html/rfc2663. Accessed: 2019-08-27.

[20] RFC 4987. https://tools.ietf.org/html/rfc4987. Accessed: 2019-08-27.

[21] RFC 6928. https://tools.ietf.org/html/rfc6928. Accessed: 2019-08-27.

[22] RFC 7323. https://tools.ietf.org/html/rfc7323. Accessed: 2019-08-27.

[23] RFC 791. https://tools.ietf.org/html/rfc791. Accessed: 2019-08-27.

[24] RFC 793. https://tools.ietf.org/html/rfc793. Accessed: 2019-08-27.

[25] Solarflare SFA7942Q with Stratix V A7 FPGA. https://solarflare.com/wp-content/uploads/2018/11/SF-114649-CD-LATEST_Solarflare_AOE_SFA7942Q_Product_Brief.pdf. Accessed: 2019-08-27.

[26] SpecWeb2009 Benchmark. https://www.spec.org/web2009/. Accessed: 2019-08-27.

[27] Why Are We Deprecating Network Performance Features (KB4014193)? https://blogs.technet.microsoft.com/askpfeplat/2017/06/13/. Accessed: 2019-08-27.

[28] Why persistent connections are bad. https://meta.wikimedia.org/wiki/Why_persistent_connections_are_bad#Why_persistent_connections_are_bad. Accessed: 2019-08-27.

[29] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[30] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Transactions on Computer Systems (TOCS)*, 34(4):11, 2017.

[31] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 2010 ACM Internet Measurement Conference (IMC '10)*, 2010.

[32] Theophilus Benson, Aditya Akella, and David A Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement (IMC '10)*, 2010.

[33] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[34] Hsin-Chieh Chiang, Yuan-Pang Dai, and Chuei-Yu Wang. Full Hardware Based TCP/IP Traffic Offload Engine (TOE) Device and the Method Thereof, January 12 2010. US Patent 7,647,416.

[35] Douglas Freimuth, Elbert C Hu, Jason D LaVoie, Ronald Mraz, Erich M Nahum, Prashant Pradhan, and John M Tracey. Server Network Scalability and TCP Offload. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC '05)*, 2005.

[36] Rohan Gandhi, Y. Charlie Hu, and Ming Zhang. Yoda: A Highly Available Layer-7 Load Balancer. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys '16)*, 2016.

[37] 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 ACM SIGCOMM Conference (SIGCOMM '16)*, 2016.

[38] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: A GPU-Accelerated Software Router. In *Proceedings of the 2010 ACM SIGCOMM Conference (SIGCOMM '10)*, 2010.

[39] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, 2012.

[40] David Hancock and Jacobus Van der Merwe. Hyper4: Using P4 to Virtualize the Programmable Data Plane. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies (CoNEXT '16)*, 2016.

[41] EunYoung Jeong, Shinae Woo, Muhammad Asim 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 Symposium on Networked Systems Design and Implementation (NSDI '14)*, 2014.

[42] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, 2019.

[43] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided RDMA Datagram RPCs. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, 2016.

[44] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, 2019.

[45] Georgios P Katsikas, Tom Barbette, Dejan Kostic, Rebecca Steinert, and Gerald Q Maguire Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI '18)*, 2018.

[46] Antoine Kaufmann. Efficient, Secure, and Flexible High Speed Packet Processing for Data Centers. In *PhD Thesis, University of Washington*, 2018.

[47] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas E. Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, 2016.

[48] 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 14th European Conference on Computer Systems (EuroSys '19)*, 2019.

[49] Jakub Kicinski and Nicolaas Viljoen. eBPF Hardware Offload to SmartNICs: cls bpf and XDP. *Proceedings of Netdev 1.2, The Technical Conference on Linux Networking*, 1, 2016.

[50] Hyong-youb Kim and Scott Rixner. Connection Handoff Policies for TCP Offload Network Interfaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, 2006.

[51] Mirja Kühlewind, Sebastian Neuner, and Brian Trammell. On the state of ECN and TCP options on the Internet. In *Proceedings of the 14th Passive and Active Measurement Conference (PAM '13)*, 2013.

[52] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M Swift, and TV Lakshman. UNO: Uniflying Host and Smart NIC Offload for Flexible Packet Processing. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC '17)*, 2017.

[53] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, 2017.

[54] Gregor Maier, Anja Feldmann, Vern Paxson, and Mark Allman. On Dominant Characteristics of Residential Broadband Internet Traffic. In *Proceedings of the 2009 ACM Internet Measurement Conference (IMC '09)*, 2009.

[55] Ilias Marinos, Robert NM Watson, and Mark Handley. Network Stack Specialization for Performance. *ACM SIGCOMM Computer Communication Review*, 44(4):175–186, 2014.

[56] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the 2017 ACM SIGCOMM Conference (SIGCOMM '17)*, 2017.

[57] Jeffrey C Mogul. TCP Offload Is a Dumb Idea Whose Time Has Come. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS '03)*, 2003.

[58] 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 ACM SIGCOMM Conference (SIGCOMM '18)*, 2018.

[59] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, 2019.

[60] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving Network Connection Locality on Multicore Systems. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys '12)*, 2012.

[61] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System Is the Control Plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):11, 2016.

[62] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The Design and Implementation of Open vSwitch. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, 2015.

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

[64] Lin Quan and John Heidemann. On the Characteristics and Reasons of Long-lived Internet Flows. In *Proceedings of the 2010 ACM Internet Measurement Conference (IMC '10)*, 2010.

[65] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM SIGCOMM Conference (SIGCOMM '15)*, 2015.

[66] Shinae Woo, Eunyoung Jeong, Shinjo Park, Jongmin Lee, Sunghwan Ihm, and KyoungSoo Park. Comparison of Caching Strategies in Modern Cellular Backhaul Networks. In *Proceedings of the 11th International Conference on Mobile Systems, Applications, and Services (MobiSys'13)*, 2013.

[67] Zhong-Zhen Wu and Han-Chiang Chen. Design and Implementation of TCP/IP Offload Engine System over Gigabit Ethernet. In *Proceedings of the 15th International Conference on Computer Communications and Networks*. IEEE, 2006.

[68] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC '16)*, 2016.

[69] Tao Zhang, Jianxin Wang, Jiawei Huang, Jianer Chen, Yi Pan, and Geyong Min. Tuning the Aggressive TCP Behavior for Highly Concurrent HTTP Connections in Intra-datacenter. *IEEE/ACM Transactions on Networking (TON)*, 25(6):3808–3822, 2017.

[70] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. *ACM SIGCOMM Computer Communication Review*, 45(4):523–536, 2015.