AccelTCP: Accelerating Network Applications with Stateful TCP Offloading

YoungGyoun Moon, Seungeon Lee, Muhammad Asim Jamshed*, KyoungSoo Park

School of Electrical Engineering, KAIST
* Intel Labs
TCP is widely adopted in modern networks

- Used by 95+% of WAN traffic and 50+% of datacenter traffic [1][2]
- The gap between network bandwidth and CPU capacity widens

<table>
<thead>
<tr>
<th>2009</th>
<th>2019</th>
</tr>
</thead>
<tbody>
<tr>
<td>1GbE</td>
<td>100GbE</td>
</tr>
</tbody>
</table>

100x

CPU efficiency of TCP stack becoming increasingly important

[1] Comparison of Caching Strategies in Modern Cellular Backhaul Networks (MobiSys ’13)
[2] RDMA over Commodity Ethernet at Scale (SIGCOMM ’16)
## Suboptimal CPU efficiency in TCP stacks

- Recent TCP stacks adopt numerous optimization techniques
  - e.g., optimized packet I/O, kernel-bypassing, zero-copying

- Unfortunately, fundamentally limited by TCP conformance overhead

<table>
<thead>
<tr>
<th>Host CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection management</td>
</tr>
<tr>
<td>Reliable data transfer</td>
</tr>
<tr>
<td>Buffer management</td>
</tr>
<tr>
<td>Congestion/flow control</td>
</tr>
</tbody>
</table>
TCP overhead in short-lived connections

- Short TCP flows dominates the Internet
  - 80% of cellular network traffic is smaller than 8KB \(^1\)

- Connection management overhead in short TCP flows

**CPU breakdown of mTCP + Redis**
- A single key-value lookup per connection

\(^1\) Comparison of Caching Strategies in Modern Cellular Backhaul Networks (MobiSys '13)
TCP overhead in Layer-7 (L7) proxying

- L7 proxies are widely adopted (e.g., load balancer, API gateway)

- Payload relaying overhead in L7 proxies

```
App

TCP

NIC

Connection 1

splice()

Connection 2
```

Overheads
- Memcpy from/to app
- TCP processing
- DMA overhead

L7 load balancer

100GbE = 44 cores
Our work: AccelTCP

NIC offload of mechanical operations for TCP conformance

Connection management

Connection splicing
Existing TCP NIC offloads

- Full-stack TCP offload engine (TOE)
  - Poor connection scalability
  - Difficult to extend (e.g., adding a new congestion control algorithm)

- TCP Segmentation Offload (TSO) and Large Receive Offload (LRO)
  - Saves significant CPU cycles for processing large messages
Our work: AccelTCP

Extend the benefit of NIC offload to general TCP applications

Small-message connections

Large-message connections

Server/clients

Proxies

AccelTCP

Typical TCP offloads (e.g., TSO, LRO)
A dual-stack TCP architecture with stateful TCP offloading
• Selectively offloads peripheral TCP operations to NICs

Host stack
- Reliable data transfer
- Buffer management
- Congestion/flow control

Central TCP operations
→ Required for data transfer

NIC stack
- Segmentation/checksum
- Connection setup/teardown
- Connection splicing

Peripheral TCP operations
→ Required for protocol conformance
AccelTCP design overview

- A dual-stack TCP architecture with stateful TCP offloading
  - Selectively offloads peripheral TCP operations to NICs

**Host stack**
- Reliable data transfer
- Buffer management
- Congestion/flow control

**NIC stack**
- Segmentation/checksum
- Connection setup/teardown
- Connection splicing

**Challenges**
- Synchronizing flow states
- Limited NIC resources
Challenge #1. Synchronizing flow states

- Connection management and splicing are stateful TCP operations
  - Transmission control block (TCB) needs to be updated

- Challenging to maintain flow state consistency across two stacks
  - Huge DMA cost to deliver sync messages
Challenge #1. Synchronizing flow states

- Our approach: Single ownership of a TCP flow and its TCB
- Key ideas:
  - TCB sync occurs only in between the different phases
  - TCB sync messages are piggybacked with payload packets
Challenge #2. Limited NIC resources

- Limited fast memory size
  - For holding program instructions and connection states
  - e.g., 8MB SRAM in Netronome Agilio LX

- Limited compute capacity
  - Typical TCP stacks: 1000 - 3000 cycles/packet
    → Performance drop by 30 - 80% in Agilio LX
**Challenge #2. Limited NIC resources**

Our approach: Minimize NIC dataplane complexity

<table>
<thead>
<tr>
<th></th>
<th>Limited memory</th>
<th>Limited CPU capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Connection setup</strong></td>
<td>Use SYN cookie (\rightarrow) stateless operation</td>
<td>Use fast hashing (in hardware)</td>
</tr>
<tr>
<td><strong>Connection splicing</strong></td>
<td>Minimize TCB on NIC</td>
<td>Differential checksum update</td>
</tr>
<tr>
<td><strong>Connection teardown</strong></td>
<td># of concurrent flows: (10k \rightarrow 256k)</td>
<td><strong>Timer bitmap wheel</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>this talk</strong></td>
</tr>
</tbody>
</table>
Tracking timeouts on NIC

- Required for TCP retransmission or last ACK timeout, TIME_WAIT

- No flow-to-core affinity → A global data structure for tracking timeout
  - Frequent timer registration incurs a huge lock contention
Timer bitmap wheel

- Efficient timer registration & invocation in NIC dataplane

3 flows added (RTO for FIN = 8ms)

flow bitmap (indexed by flow ID)

scanned by
- core 1
- core 2
- core 3
- core 4
- core N

Efficient timer registration & invocation in NIC dataplane
Host stack optimizations

1. User-level threading
   • Avoid heavy context switching overhead between TCP stack and app

2. Opportunistic zero-copy
   • Avoid socket buffer copy if packets can be delivered directly from/to app

3. Lazy TCB Creation
   • Many fields of TCB (up to 700 bytes) are unused in single transaction case
     ➢ Our approach: Create a quasi-TCB (40 bytes) for a new connection

Check out our paper for more details 😊
Implementation and experiment setup

- NIC stack: running on Netronome Agilio NICs
  - 1,501 lines of C code and 195 lines of P4 code

- Host stack: extended mTCP to support NIC offloads
  - Easy to port existing apps (connect() → mtcp_connect())

- Experiment setup
  - CPU: Xeon Gold 6142 (16-cores @ 2.6GHz)
  - NIC: Netronome Agilio LX 40GbE x2
  - Memory: 128GB DDR4 RAM
  - Use up to 8 client machines (Xeon E5-2640 v3) to generate workload
Does AccelTCP support high connection rate?

- Throughput performance of a TCP server
  - A single 64B packet transaction per connection

![Graph showing throughput performance comparison between mTCP and AccelTCP. The x-axis represents the number of CPU cores, and the y-axis represents transactions per second (x10⁶). The graph indicates that AccelTCP outperforms mTCP with increasing CPU cores. At 8 cores, AccelTCP provides 2.2x more transactions than mTCP. The NIC bottleneck is noted at 8 cores.]
Do applications benefit from AccelTCP?

Redis under Facebook USR workload (flow size: < 20B)

Throughput

- mTCP
- AccelTCP

![Graph showing throughput comparison between mTCP and AccelTCP.](image)

CPU utilization

- mTCP: TCP/IP 1/4 Redis app
- AccelTCP: Redis app

![Graph showing CPU utilization.](image)
Do applications benefit from AccelTCP?

HAProxy under SpecWeb2009-like workload

<table>
<thead>
<tr>
<th></th>
<th>mTCP (8-core)</th>
<th>AccelTCP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6.2 Gbps</td>
<td>73.1 Gbps</td>
</tr>
<tr>
<td>Speed Gain</td>
<td></td>
<td>11.8x</td>
</tr>
</tbody>
</table>
Summary

- TCP performance limited by protocol conformance overhead
  - Short-lived flows and L7 proxies cannot benefit from existing TCP offloads

- AccelTCP explores a new design space of NIC-assisted TCP stack
  - Connection management and splicing can be offloaded to NIC

- AccelTCP significantly improves CPU efficiency of real-world apps
  - 2.3x improvement with Redis, 12x improvement with HAproxy

Source code available: 
- shader.kaist.edu/accltcp
- github.com/accltcp