Designing an Efficient TCP Stack with P4-enabled SmartNICs

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

School of Electrical Engineering, KAIST
* Intel Labs

Transmission Control Protocol (TCP)

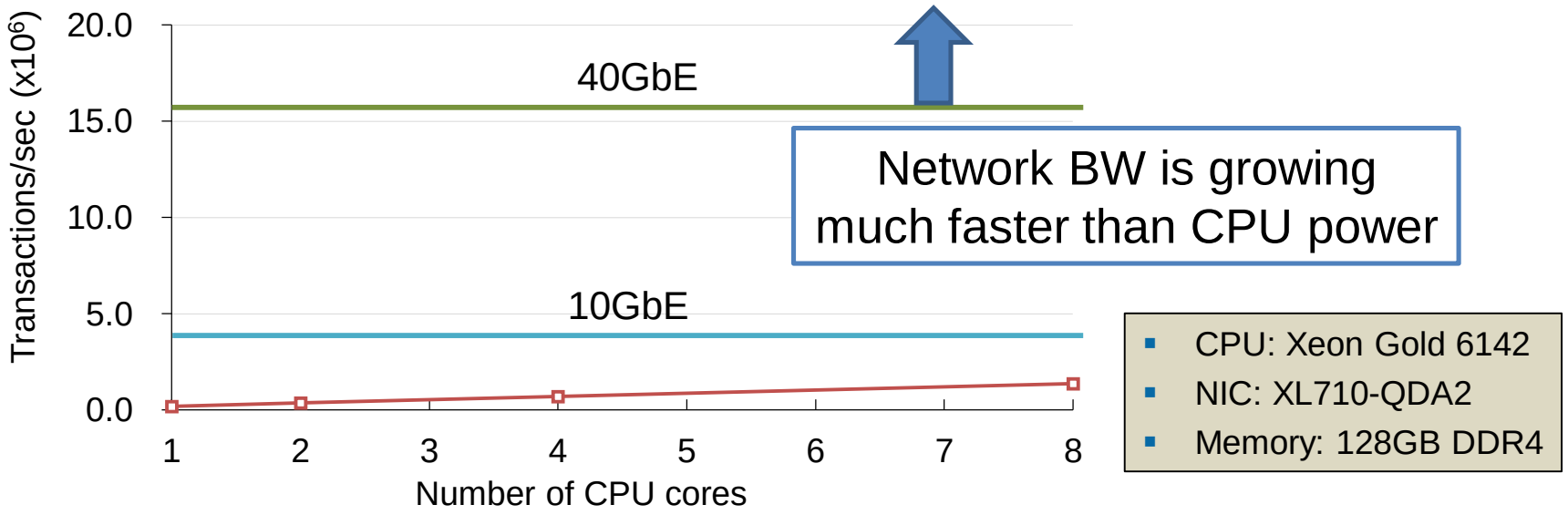
- Reliable data transfer without overwhelming network
- Easy to deploy: no specialized in-network support required

- TCP is widely adopted in modern networks
  - Cellular network: 95+% of traffic is TCP [1]
  - Datacenter network: half of the traffic is TCP [2]

[1] Comparison of Caching Strategies in Modern Cellular Backhaul Networks (MobiSys '13)
[2] RDMA over Commodity Ethernet at Scale (SIGCOMM '16)

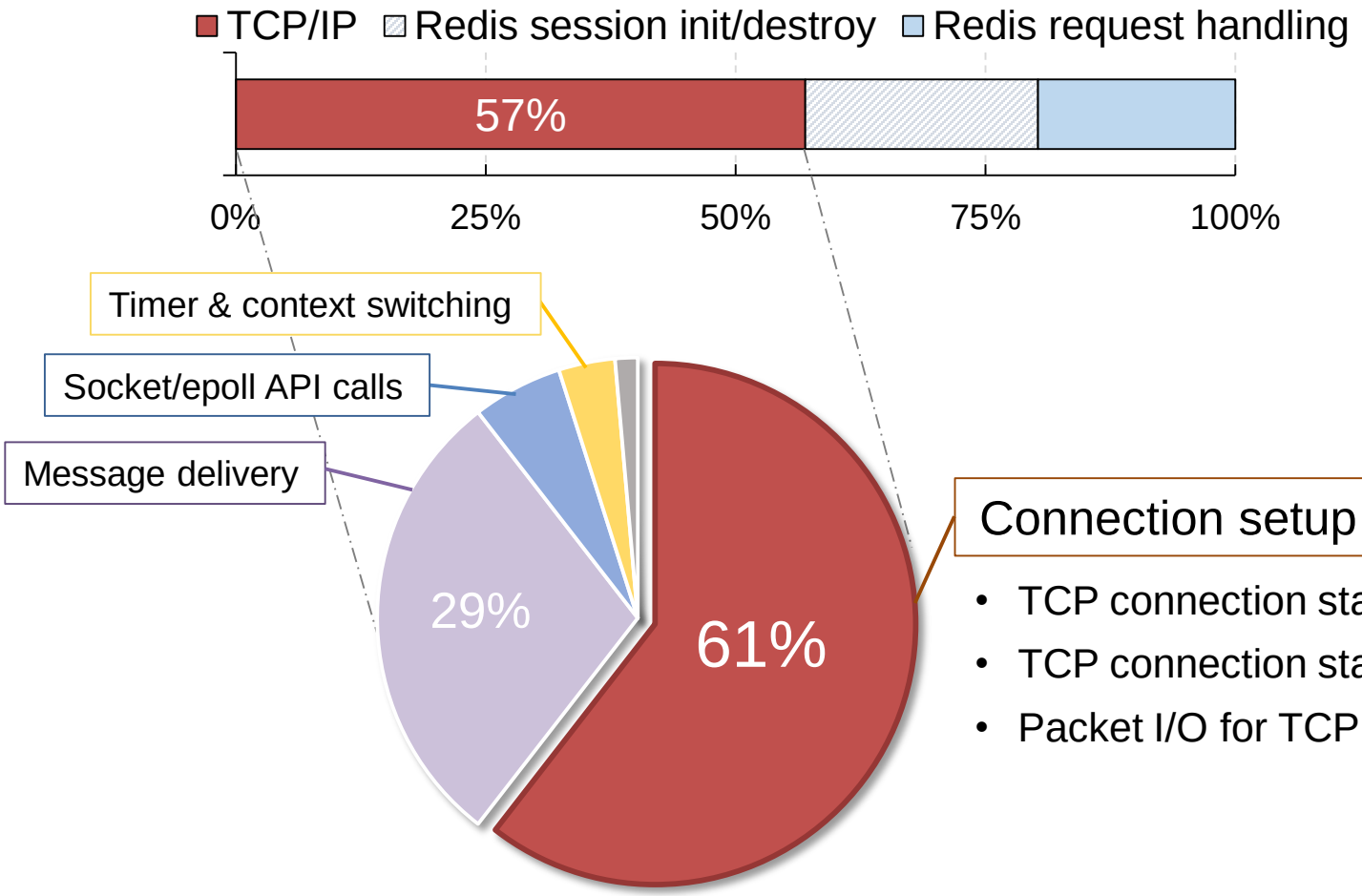
Kernel-Bypass TCP Stacks

- Kernel TCP stack: inefficient packet I/O and limited core scalability
- Several user-level TCP stacks are proposed to avoid the inefficiencies
  - e.g., mTCP [NSDI '14], IX [OSDI '14], Sandstorm [SIGCOMM '14]
  - Key-value store performance with short-lived TCP connections
    - Redis (v4.0.8) patched to mTCP with DPDK 17.08
    - Facebook USR workload with short (< 20B) keys and 2B values



TCP Overhead in Short-lived Connections

- CPU breakdown of Redis with mTCP



Connection setup & teardown

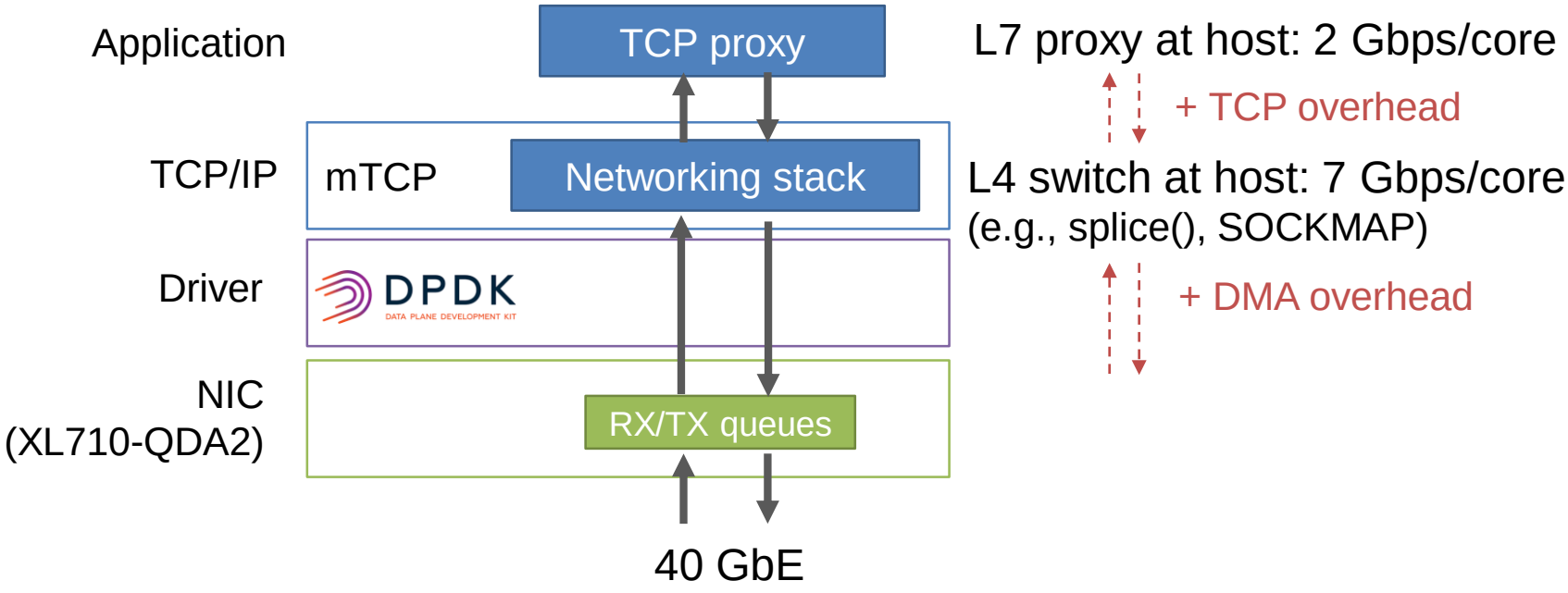
- TCP connection state init & destroy
- TCP connection state update
- Packet I/O for TCP control packets

KAIST

TCP Overhead in Application-level Proxying

- Typical operations of a transparent Layer-7 (L7) proxy
  - Accept a client connection, read a request, and decide a backend server
  - Relay payload between the client and the backend server

- Cost of relaying TCP bytestream in L7 proxy (64B packets)

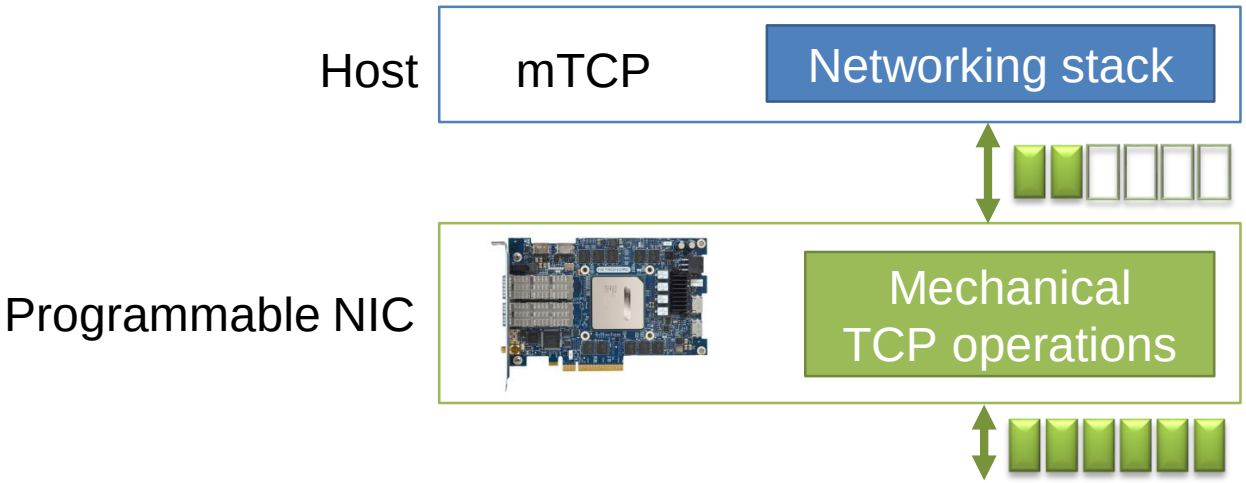


KAIST

Our Goal & Approach

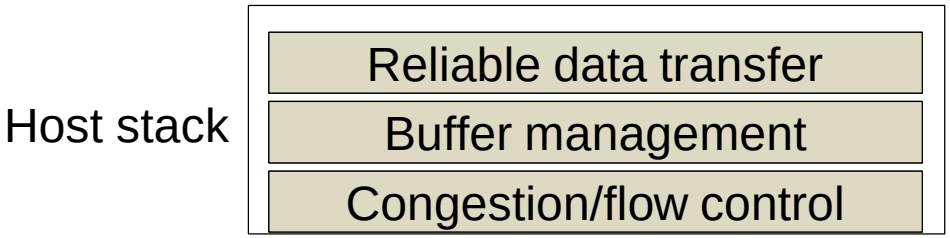
- Goal: To overcome *protocol conformance overhead* in TCP stacks
  - Connection management and control packet processing
  - Payload relaying between two connections

- Our approach: Offload mechanical TCP operations to NICs



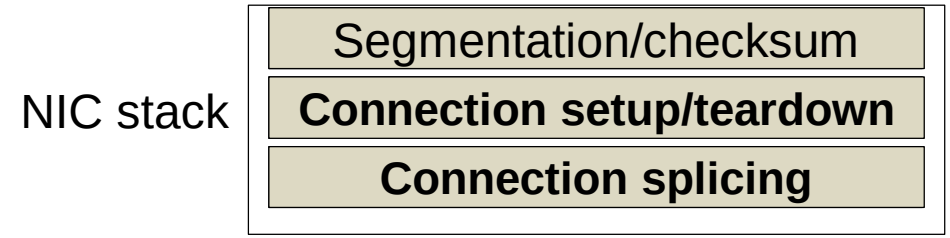
AccelTCP

- A dual-stack TCP architecture with stateful offloading



Central TCP operations

– Complex operations for application data transfer



Peripheral TCP operations

– Mechanical operations for protocol conformance

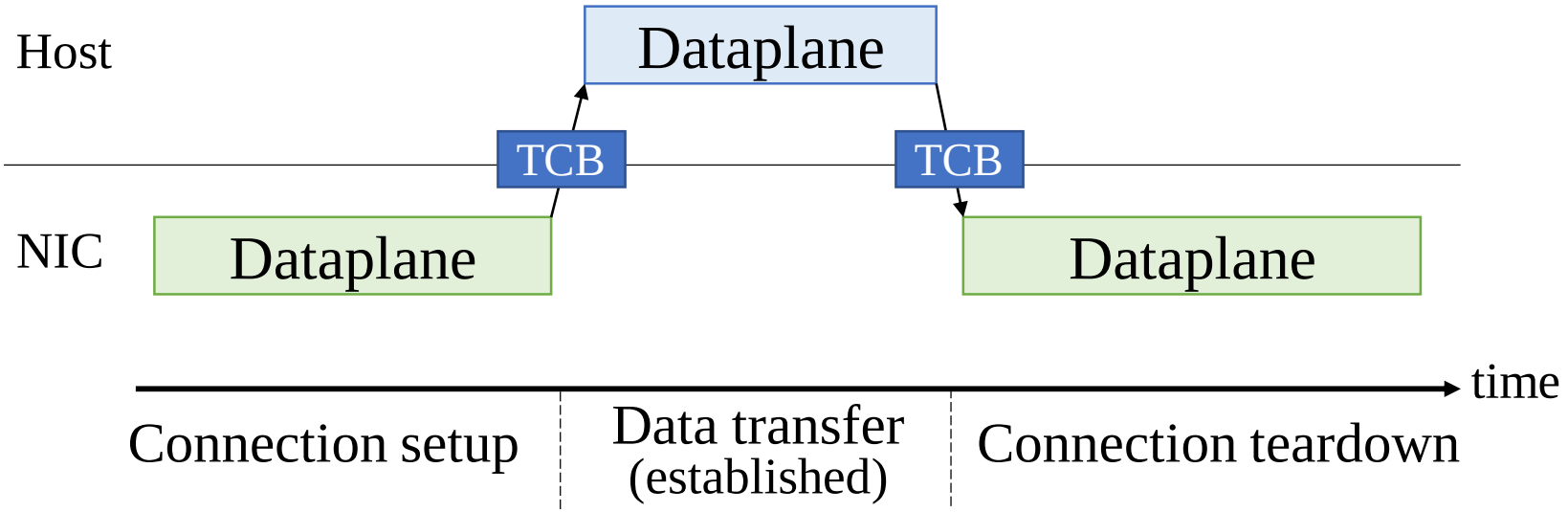
AccelTCP Design Approach

Challenge #1. Synchronize transmission control block (TCB)

- Difficult to maintain consistency across two stacks

Approach #1. Single ownership of a TCP flow at any given time

- No shared connection state between NIC and host stack



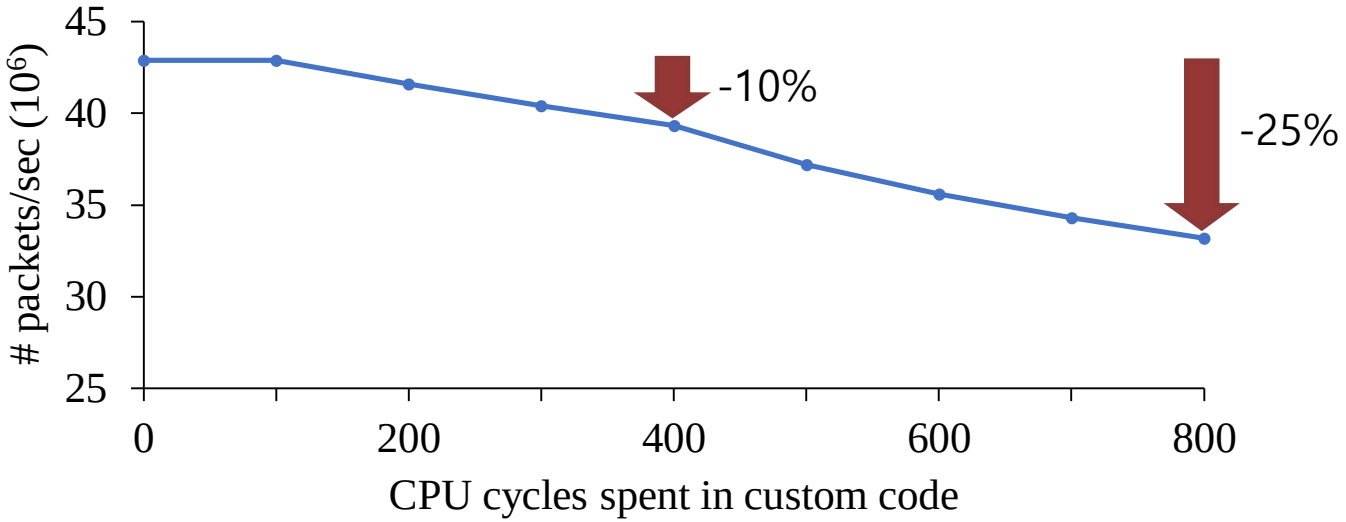
AccelTCP Design Approach

Challenge #2. Limited On-NIC Resources

Limited SRAM on NICs

- For holding program instructions and connection states
- e.g., Netronome Agilio LX (40GbE NIC): 8MB of on-chip SRAM

Limited packet processing headroom on NICs



KAIST

AccelTCP Design Approach

Approach #2. Minimize complexity on NIC dataplane

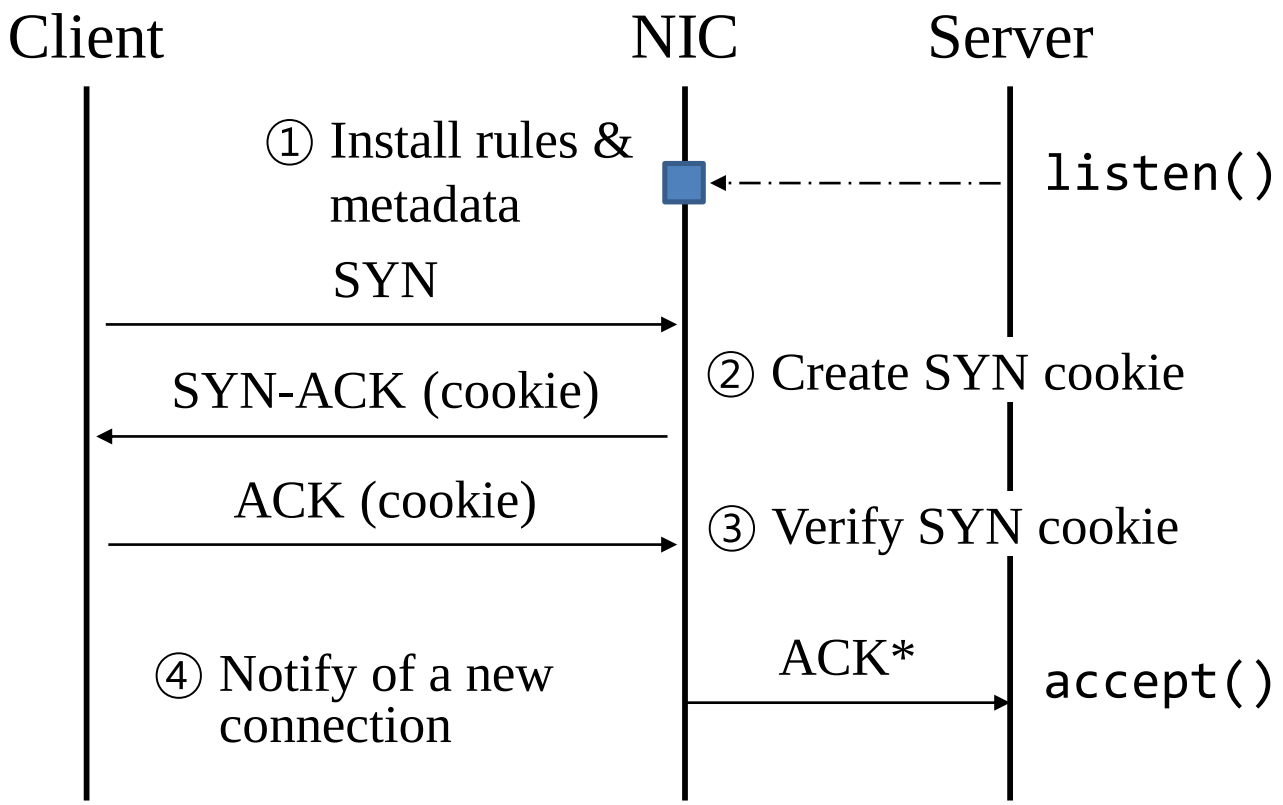
- NIC dataplane handles common-case fast path of TCP operations
  - e.g., When L7 proxy becomes transparent mode, fall back to L4 switch on NIC

Approach #3. The host side should enforce full control of offloading

- Host stack keeps monitoring the resource consumption on NIC
- The host stack should be able to operate standalone

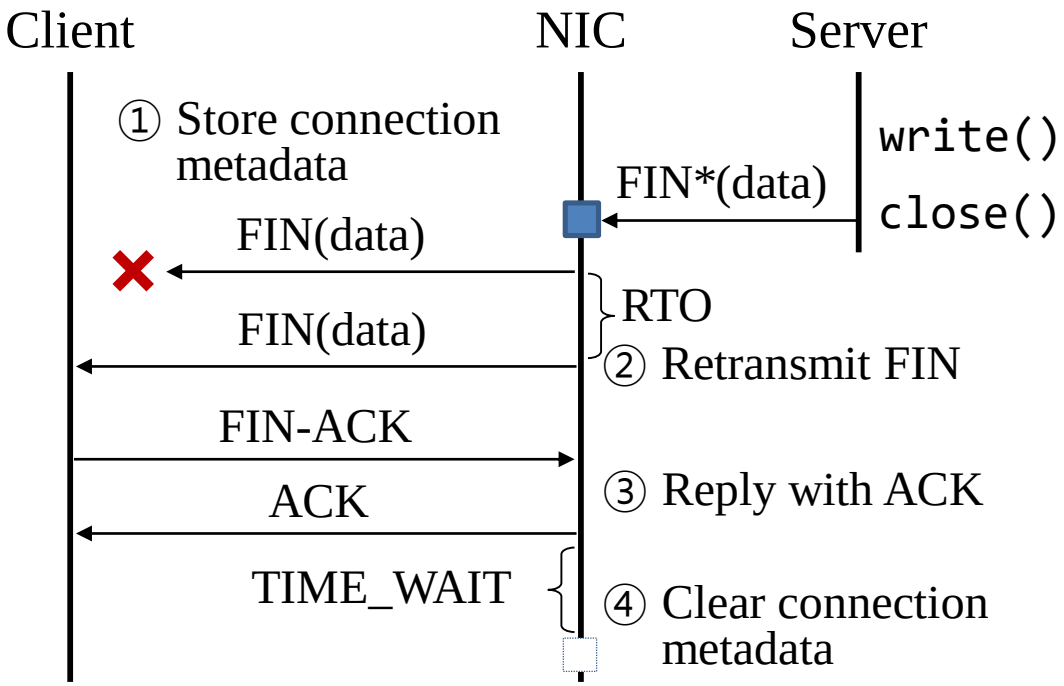
Connection Setup Offload

- Use SYN cookie to handle incoming connections
  - Can handle connection setup in a stateless manner



Connection Teardown Offload

- Common case of connection teardown is a simple state transition
  - Host stack hands over ownership of TCB and remaining send buffer data
    - Offload only if *(data size) < (initial congestion window size)*
  - Perform TCP segmentation at NIC if required (e.g., TSO)

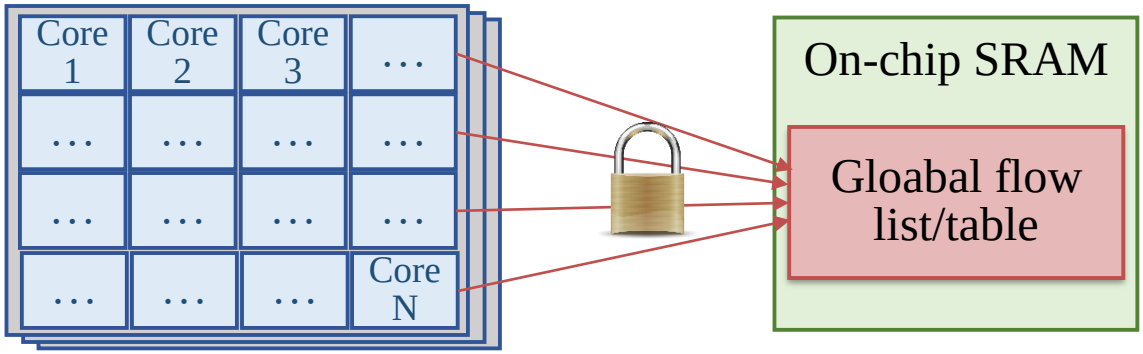


- Flow states on SRAM
  - 4-tuple
  - TCP state
  - RTO
  - Expected SEQ/ACK #
- Entire packets on DRAM (after TCP segmentation)

Handling Timeouts on NIC

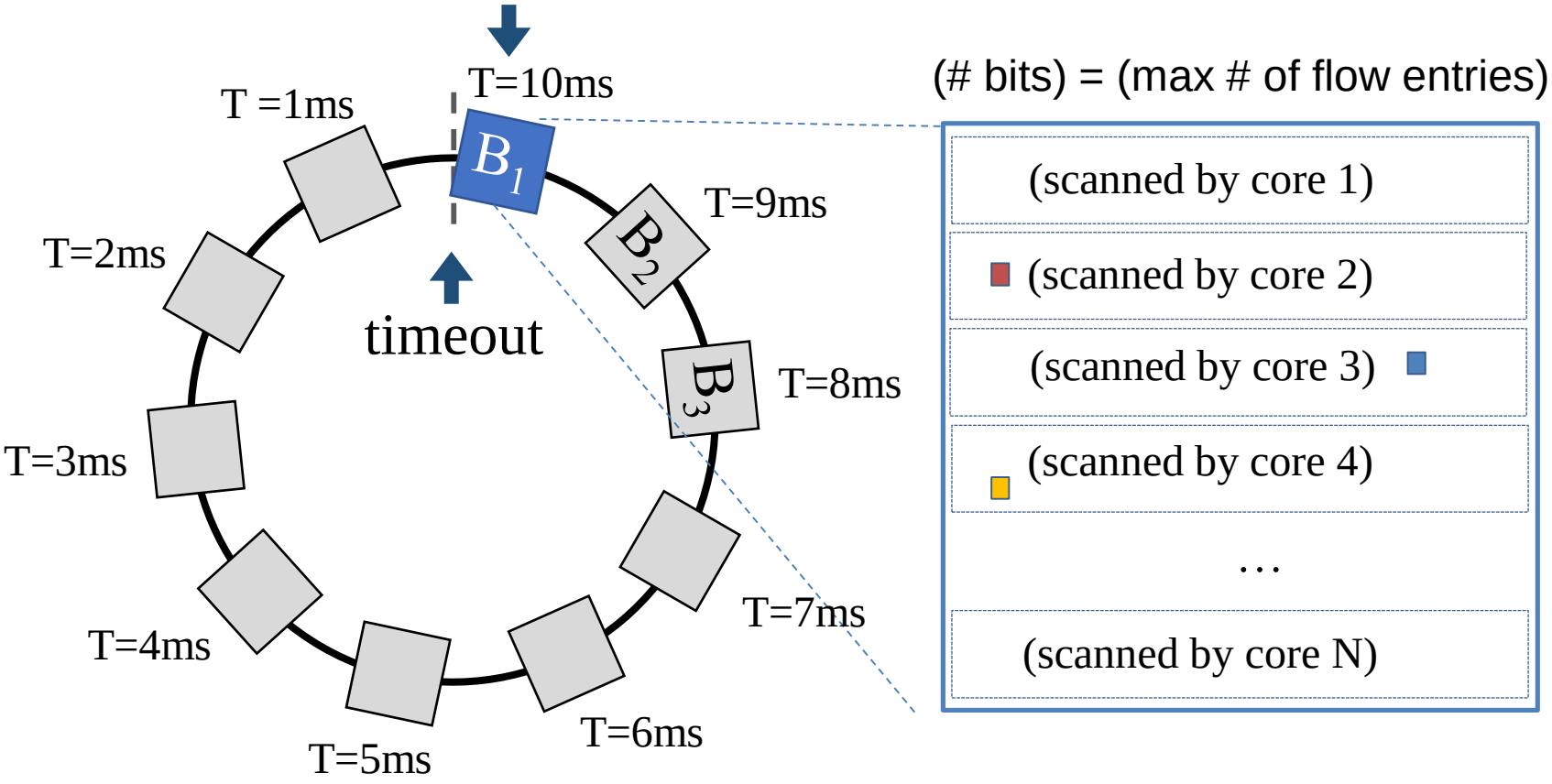
- Required for TCP retransmission or idle timeout, TIME_WAIT

- Naïve approach: a global list or hash table for tracking timeout
  - Tens or hundreds of processing cores in recent SmartNICs
    - e.g., Netronome Agilio LX have 120 cores for flow processing
    - No flow-core affinity is guaranteed
  - Concurrent access to a global structure incurs a huge lock contention



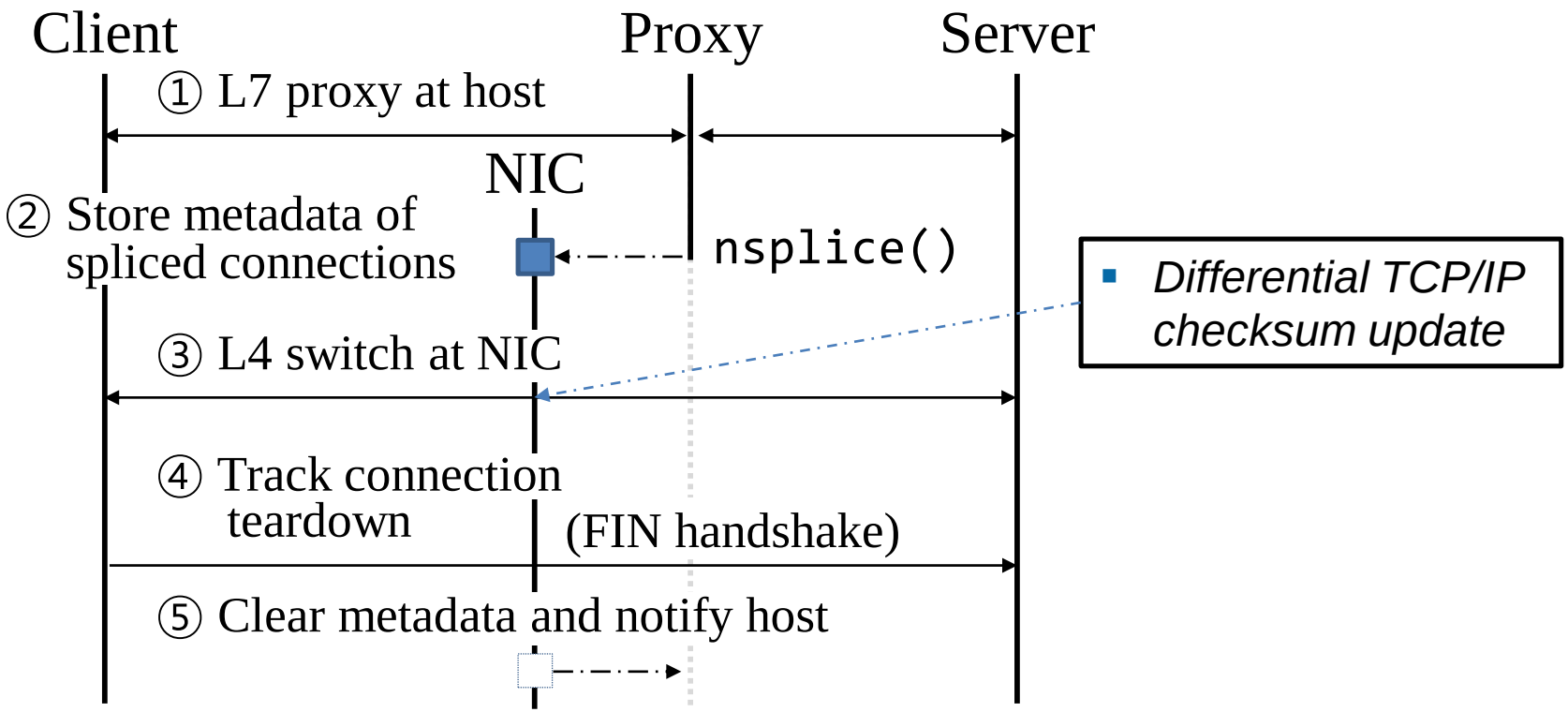
Handling Timeouts on NIC

- Our approach: timer bitmap wheel approach
  - Suitable for concurrent write-heavy workloads
  - Bitmap scan is parallelized with many processor cores



Connection Splicing Offload

- L7 proxy can ask for connection splicing offload on transparent mode
  - No more payload modification is required



AccelTCP Host Stack

- Host stack can selectively control NIC offload
  - It also handles any corner cases that cannot benefit from offload
    - e.g., SYN cookie cannot be enabled for SYNs without TCP timestamp
    - e.g., On-NIC SRAM is overloaded

- Host TCP stack optimizations
  - Opportunistic zero-copy
  - User-level threading
  - **Lazy TCB Creation**

Lazy TCB Creation

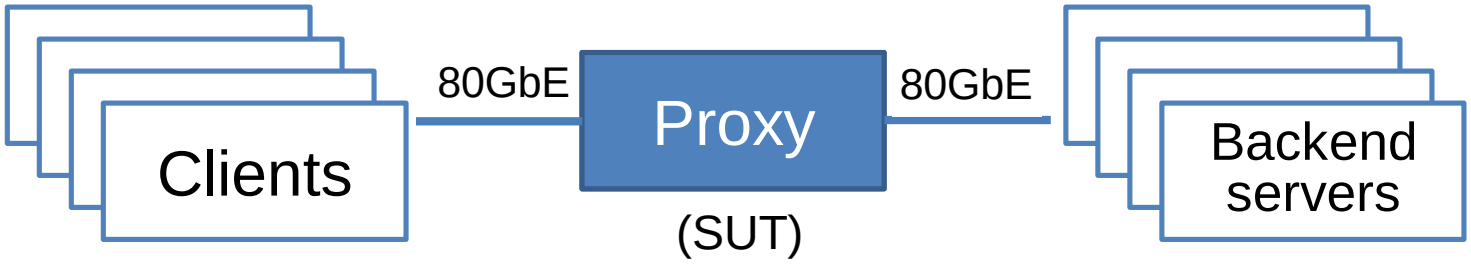
- After NIC offload, TCB init and destroy takes **30% of total CPU cycles**
  - A full TCB of a connection ranges from 400 to 700 bytes
- Our observation
  - Many of the fields are unnecessary for single-transaction case
    - e.g., metadata related to TCP send and receive buffers
- Our approach
  - Create a quasi-TCB (40 bytes) for a new connection
  - Convert it to full TCB only when multiple transactions are observed

AccelTCP Implementation

- NIC stack
  - 1,501 lines of C code and 195 lines of P4 code
    - Running on Netronome Agilio NICs (with NFP-4000 or NFP-6480 chipset)
  - Host-side L2-L4 NFs (e.g., firewalling, ACL, or host networking)
    - Must be offloaded to NIC accordingly
    - Such NFs can be written in P4 and easily integrated with AccelTCP (e.g., placing them properly at ingress/egress pipelines of the NIC dataplane)
- Host stack
  - Extended mTCP to implement AccelTCP
    - Easy to port existing apps (connect() → mtcp_connect())
  - With minimum set of additional APIs for controlling NIC offload
    - Only require 1 to 3 LoCs from mTCP to AccelTCP

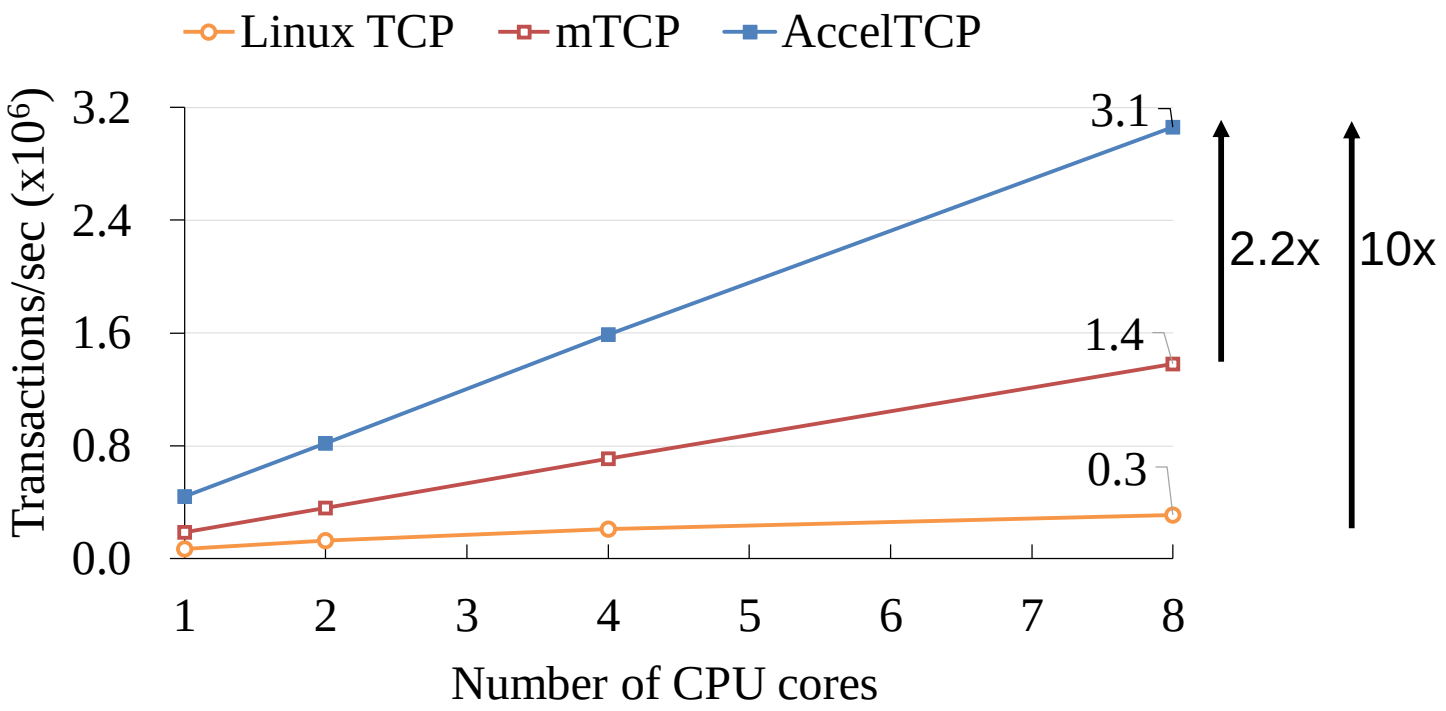
Evaluation

- Hardware configuration
  - **Server** (or proxy):
    - Xeon Gold 6142 (16-cores @ 2.60GHz), 128GB DDR4 DRAM
    - Netronome Agilio-Lx 40GbE dual-port NIC
      (* For IX experiment: two Intel 82599 dual-port 10GbE NICs)
  - **Clients**:
    - Xeon E5-2640v3 (8-cores @ 2.60GHz), 32GB DDR4 DRAM
    - XL710-QDA2 40GbE dual-port NIC (*only the 1st port is used)
  - **Backend servers** (for proxy experiment)
    - CPU: mix of Xeon E5-2699 v4 @ 2.2GHz, Xeon E5-2683 v4 @ 2.1GHz
    - XL710-QDA2 40GbE dual-port NIC (*only the 1st port is used)



Performance Improvement on Key-value Store

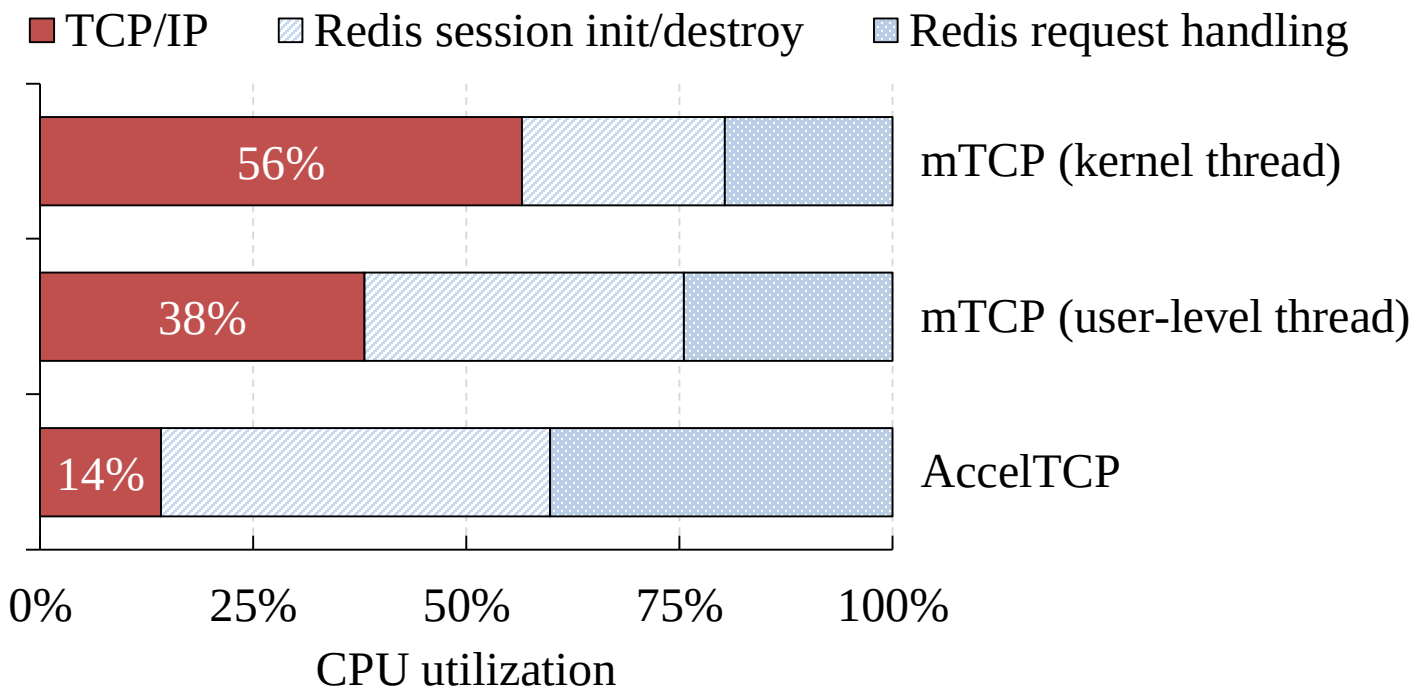
- Key-value store (Redis) performance
  - A single CPU core used
  - Under realistic workload (USR workload from Facebook)



KAIST

Performance Improvement on Key-value Store

- CPU breakdown for key-value store (Redis)



Performance Improvement on L7 Load Balancer

- Layer-7 load balancer (HAProxy) performance
  - Under realistic workload (SpecWeb2009-like)

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

Cost-effectiveness Analysis

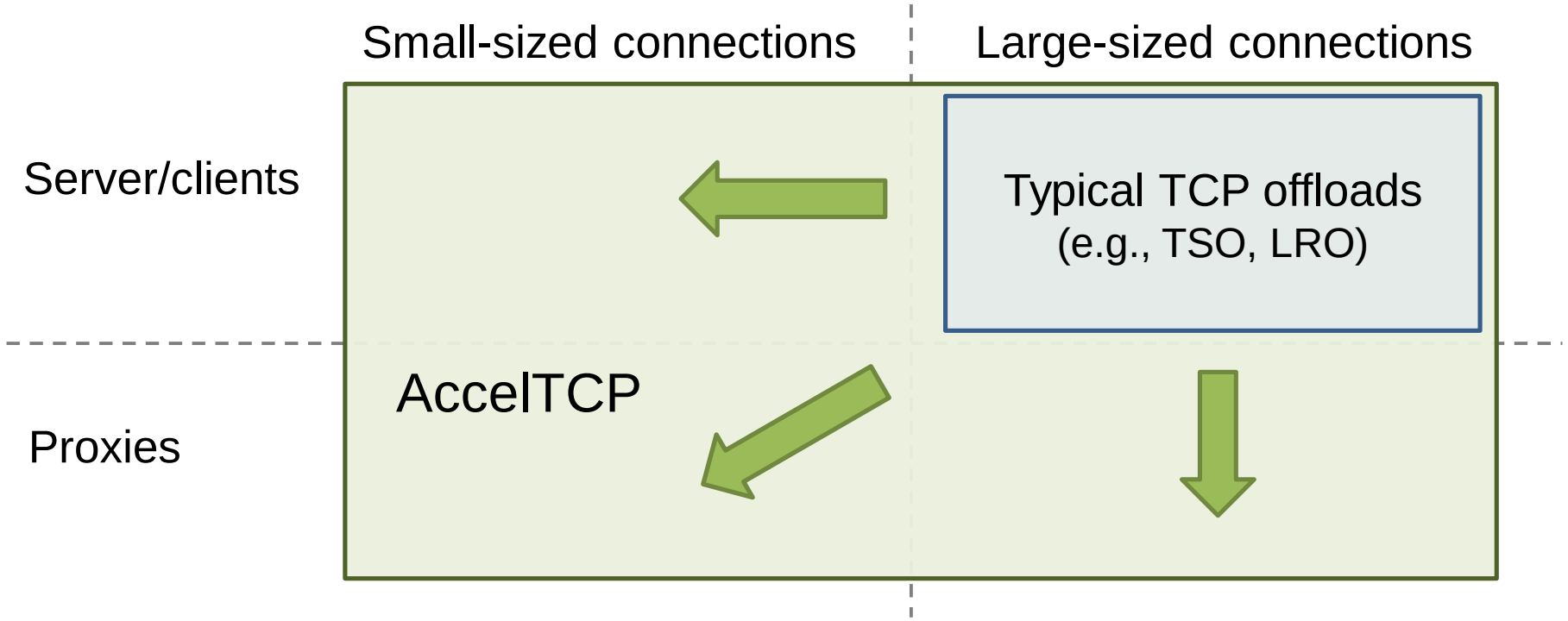
- Normalized performance-per-dollar (64B TCP echo server)

		E5-2650v2 **($1,170)**	Gold 6142 **($2,950)**
mTCP XL710 **($440)**		1	1.25
AccelTCP Agilio LX **($1,750)**		1.93 (93%↑)	1.96 (57%↑)

KAIST

Comparison Against Prior TCP Offloads

- TCP Offload Engine (TOE): offload entire TCP stack to NICs
  - Unpopular in practice: complex interface, limited NIC resources
- TCP Segmentation Offload (TSO) and Large Receive Offload (LRO)
  - Significantly saves CPU cycles for processing large messages



Conclusion

- TCP stack performance is fundamentally limited due to protocol conformance overhead
- We propose AccelTCP, a hardware-assisted TCP stack architecture
  - Harnesses programmable NICs as a TCP protocol accelerator
- AccelTCP saves a significant amount of CPU cycles used for connection management and proxying
  - Improves the performance of key-value store by 2.3x
  - Improves the performance of L7 proxying by up to 11.9x

Thank you