Building Flexible Data Center Network Stacks for the Terabit Era

Rajath Shashidhara

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2025

Reading Committee:

Simon Peter, Chair

Arvind Krishnamurthy

Antoine Kaufmann

Program Authorized to Offer Degree:
Computer Science & Engineering

© Copyright 2025

Rajath Shashidhara

University of Washington

Abstract

Building Flexible Data Center Network Stacks for the Terabit Era

Rajath Shashidhara

Chair of the Supervisory Committee:
Simon Peter
Paul G. Allen School of Computer Science & Engineering

Modern data center workloads demand end-host network stacks that sustain terabit-scale bandwidth alongside μs-scale latency, overwhelming traditional software TCP stacks with high CPU overheads. ASIC-based transport offloads deliver high performance and energy efficiency but sacrifice flexibility, hindering customization to diverse application and deployment needs. This thesis explores flexible stateful TCP offload using emerging programmable in-network accelerators. It tackles the core challenge of mapping TCP's complex, stateful processing onto the restrictive programming models of resource-constrained hardware, enabling fine-grained data-path parallelization.

We present FlexTOE and Laminar, two novel TCP stack offloads built on Network Processing Unit (NPU) and Reconfigurable Match-Action Table (RMT) architectures. Both eliminate all host TCP data-path CPU overheads, integrate transparently with existing applications, remain robust under realistic network dynamics, and crucially, retain software programmability. The design principles developed generalize beyond TCP and extend naturally to other accelerator architectures.

Through extensive evaluation, we demonstrate that these practical designs achieve a meaningful balance of high-performance, energy efficiency, and flexibility, surpassing state-of-the-art software stacks and offering a viable, adaptable alternative to rigid hardware transports.

Contents

1 Introduction **1**

1.1 Challenges for Data Center TCP Stacks . . . 2

1.2 Objectives . . . 4

1.3 Opportunity: Programmable In-Network Accelerators . . . 5

1.4 Thesis Statement . . . 6

1.5 Summary of Contributions . . . 8

1.6 Published Works . . . 9

1.7 Outline . . . 10

2 Background **11**

2.1 TCP Impact on Host CPU Performance . . . 11

2.2 Programmable In-Network Accelerators . . . 15

2.2.1 High-Efficiency CPU Cores . . . 15

2.2.2 Field-Programmable Gate Arrays (FPGAs) . . . 16

2.2.3 Network Processing Units (NPUs) . . . 17

2.2.4 Reconfigurable Match-Action Table (RMT) Pipeline . . . 18

2.3 Existing TCP Stack Architectures . . . 18

2.4 Discussion . . . 20

3 High-Level Design **21**

3.1 Offload Architecture . . . 21

3.1.1 Components . . . 22

3.1.2 Interfaces . . . 23

3.2 Fine-Grained Data-path Parallelization 24
3.2.1 Data-Parallel Pipeline 24
3.2.2 Match-Action Architecture 25

4 FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism 26
4.1 Background 28
4.1.1 NPU SmartNIC Architecture 28
4.1.2 Implications for Flexible Offload 30
4.2 Overview 30
4.3 TCP Data-path Parallelization 32
4.3.1 Data-path Workflows 34
4.3.2 Sequencing and Reordering 39
4.3.3 Flexibility 41
4.3.4 Flow Scheduling 43
4.4 Implementation 43
4.4.1 Agilio-CX40 Implementation 44
4.4.2 FlexTOE x86 and BlueField Ports 47
4.5 Evaluation 49
4.5.1 Benefit of Flexible Offload 50
4.5.2 Remote Procedure Calls (RPCs) 54
4.5.3 Robustness 60
4.6 Conclusion 63

5 Laminar: A Match-Action TCP Stack for the Terabit Era 65
5.1 Background 67
5.1.1 RMT Architecture 68
5.1.2 Challenges for TCP on RMT 71
5.2 Overview 74
5.3 RMT TCP Transport Logic Design 76
5.3.1 Segment Reassembly 77
5.3.2 (Re-)transmission 82
5.3.3 Sequence-Address Translation 84
5.4 Implementation 85

5.5 Evaluation 87
5.5.1 Remote Procedure Calls (RPCs) 88
5.5.2 Byte Streaming 95
5.5.3 Flexibility 98
5.5.4 Robustness 100
5.5.5 Generalizability 105
5.6 Conclusion 106

6 Related Work **107**
6.1 Software Stack Improvements 107
6.1.1 Efficient Application Interfaces 108
6.1.2 Kernel-bypass 108
6.1.3 Busy Polling 109
6.1.4 Hardware Assistance 109
6.1.5 Driver Optimizations 110
6.1.6 Scalable Architectures 111
6.2 Transport Offload 112
6.2.1 Hardware Transports 112
6.2.2 FPGA Offloads 113
6.2.3 CPU-based SmartNIC Offloads 115
6.2.4 NPU-based SmartNIC Offloads 115
6.2.5 RMT-pipeline Offloads 116
6.2.6 GPUs as Network Accelerators 116
6.3 Specialized Transport Co-designs 117
6.3.1 Software Stacks 118
6.3.2 RDMA Co-design 118
6.3.3 Co-design with In-network Accelerators 119

7 Generalizing Within and Beyond TCP **120**
7.1 Limitations 121
7.2 Generalizability Beyond TCP 123
7.2.1 Message Orientation 125
7.2.2 Weaker Reliability and Ordering Semantics 126

7.2.3 Multi-pathing 127
7.2.4 Connection-less Operation 128
7.2.5 Receiver-driven Congestion Control 128
7.2.6 Packet Prioritization and Trimming 129
7.2.7 Summary 129

8 Conclusion **130**
8.1 Future Directions 131
8.1.1 Programmable Architecture Search 131
8.1.2 Towards Disaggregated Deployments 132
8.1.3 Towards Higher-level Application Interfaces 134
8.1.4 Beyond Transport: Offloading Broader RPC Operations 135
8.1.5 Networking for ML Workloads 135
8.2 Concluding Remarks 137

Bibliography **139**

List of Tables

2.1 CPU overhead of TCP processing per Memcached request. 12
2.2 Breakdown of TCP/IP stack overheads in TAS. 14

4.1 FlexTOE connection state partitions. 35
4.2 FlexTOE: CPU profile of TCP processing per Memcached request. 51
4.3 FlexTOE performance with flexible extensions. 54
4.4 FlexTOE data-path parallelism breakdown. 59
4.5 FlexTOE congestion control under incast. 62

5.1 Laminar TCP out-of-order (OOO) merge using pseudo segments. 81
5.2 Tofino2 RMT pipeline resource usage for Laminar. 99
5.3 Simulated Flow Completion Times (FCTs) across Out-of-Order (OOO) tracking fidelity levels. 104

7.1 Comparison of modern data center transport protocols. 124

List of Figures

1.1 Mellanox/NVIDIA ConnectX Network Interface Card (NIC) bandwidth over the years. . . . 2

3.1 Fine-Grained TCP Data-Path Parallelization . . . 24

4.1 Netronome NFP-4000 NPU SmartNIC architecture. . . . 29
4.2 FlexTOE offload architecture. . . . 32
4.3 FlexTOE per-connection data-path workflows. . . . 33
4.4 FlexTOE HC pipeline: Transmit, FIN, and retransmit. . . . 36
4.5 FlexTOE TX pipeline sending 3 segments. . . . 37
4.6 FlexTOE RX pipeline receiving 3 segments, 1 out of order. . . . 39
4.7 Undesirable pipeline reordering in FlexTOE. . . . 40
4.8 FlexTOE data-path to FPC and island assignment on Agilio CX40. . . . 45
4.9 FlexTOE Memcached throughput scalability. . . . 52
4.10 Latency of different server-client stack combinations incl. FlexTOE. . . . 52
4.11 FlexTOE RPC throughput for saturated server. . . . 55
4.12 FlexTOE median, 99p and 99.99p RPC RTT. . . . 55
4.13 FlexTOE large RPC throughput with varying RPC size. . . . 57
4.14 FlexTOE connection scalability benchmark. . . . 58
4.15 FlexTOE benefits on BlueField SmartNIC. . . . 59
4.16 FlexTOE throughput, varying packet loss rate. . . . 61
4.17 FlexTOE connection throughput distribution at line rate. . . . 62

5.1 Typical RMT pipeline in switches and SmartNICs. . . . 68

5.2 TAS TCP connection state and dependencies. 72
5.3 Laminar overview. 75
5.4 RMT TCP segment reassembly paths. 78
5.5 RMT TCP processing for TX workflow. 83
5.6 RMT TCP processing path for RX workflow. 84
5.7 Sequence-address translation with double buffer in Laminar. 85
5.8 Laminar unloaded RPC round-trip time. 89
5.9 Load-latency across network stacks incl. Laminar. 90
5.10 Laminar core scalability. 91
5.11 Laminar connection scalability. 92
5.12 Laminar pipeline packet processing scalability. 93
5.13 Throughput-per-watt and tail latency for FlexKVS key-value store workload. . . 94
5.14 Pipeline dynamic power consumption with utilization. 95
5.15 Laminar streaming performance with different segment sizes. 96
5.16 SPDK NVMe-oF performance: 4K Random Writes with different network stacks. 98
5.17 Laminar streaming performance under random packet drops. 101
5.18 Laminar streaming performance under incast. 102
5.19 Laminar performance isolation for latency-sensitive and bandwidth-intensive connections. 105

Glossary

Networking

Ethernet is a wiring, signaling, addressing, and data exchange standard for local networks between computers.

MAC *Media Access Control* address is a protocol for communication within a local network.

VLAN *Virtual Local Area Network* is a logical grouping of devices on a network that allows them to communicate as if they were on the same physical LAN, regardless of their actual physical location.

Terabit Network refers to networks capable of data transfer rates beyond hundred gigabits per second (Gbps), scaling into the terabit per second (Tbps) range.

Transport

TCP *Transmission Control Protocol* is a transport layer protocol that provides reliable, ordered, and error-checked delivery of byte streams between applications running on hosts in a network.

ACK *Acknowledgment* is a signal sent from the receiver to the sender indicating that data has been successfully received.

SYN *Synchronize* is a control bit in the TCP header used to initiate a TCP connection between two hosts.

FIN *Finish* is a control bit in the TCP header used to terminate a TCP connection between two hosts.

RST *Reset* is a control bit in the TCP header used to abruptly terminate a TCP connection.

SACK *Selective Acknowledgment* is a TCP option that allows the receiver to inform the sender about all segments that have been received successfully, enabling more efficient retransmission of lost segments.

ECN *Explicit Congestion Notification* is a mechanism for indicating network congestion without dropping packets.

RTT *Round-Trip Time* is the total time it takes for a signal to be sent plus the time it takes for an acknowledgment of that signal to be received.

BDP *Bandwidth-Delay Product* is the product of a data link's capacity (in bits per second) and its round-trip delay time (in seconds). It represents the amount of data that can be in transit in the network.

MSS *Maximum Segment Size* is the largest amount of data, in bytes, that a computer or communications device can receive in a single TCP segment.

MTU *Maximum Transmission Unit* is the size of the largest protocol data unit that can be communicated in a single network layer transaction.

Handshake is a process of negotiation between two communicating parties to establish a connection and agree on communication parameters.

Flow is a sequence of packets sent from a source to a destination that are treated as a single logical unit for the purposes of network management and analysis.

FCT *Flow Completion Time* is the total time taken for a network flow to be transmitted from the source to the destination, including all delays and processing times.

RDMA *Remote Direct Memory Access* is a protocol that allows direct memory access from the memory of one computer into that of another.

RoCE *RDMA over Converged Ethernet* is a network protocol that allows RDMA over an Ethernet network.

iWARP *Internet Wide Area RDMA Protocol* is a protocol that enables RDMA over standard TCP/IP networks.

InfiniBand A high-performance, low-latency networking technology commonly used in high-performance computing environments.

WRED *Weighted Random Early Detection* is a queue management algorithm used in networking to manage congestion by selectively dropping packets based on their priority and the average queue size.

PFC *Priority Flow Control* is a link-layer flow control mechanism that allows for the pausing of specific classes of traffic on a network link to prevent packet loss during periods of congestion.

DCQCN *Data Center Quantized Congestion Notification* is a congestion control algorithm designed for data center networks that utilizes ECN markings to adjust the sending rate of RDMA flows in order to mitigate congestion.

SerDes *Serializer/Deserializer* is a pair of functional blocks used in high-speed communications to convert data between serial and parallel interfaces.

Compute

CPU *Central Processing Unit*.

ALU *Arithmetic Logic Unit* is a digital circuit that performs arithmetic and logical operations on binary data.

ISA *Instruction Set Architecture* is the set of instructions that a processor can execute, defining the interface between software and hardware.

x86 is a family of instruction set architectures based on the Intel 8086 CPU and its successors.

RISC-V is an open standard instruction set architecture based on the principles of reduced instruction set computing (RISC).

ARM is a family of reduced instruction set computing (RISC) architectures for computer processors, widely used in mobile devices and embedded systems.

MIPS *Microprocessor without Interlocked Pipeline Stages* is a RISC architecture known for its simplicity and efficiency.

IPC *Instructions-Per-Cycle* is a measure of a processor's efficiency, indicating how many instructions it can execute in a single clock cycle.

CPI *Cycles-Per-Instruction* or the reciprocal of IPC.

RMW *Read-Modify-Write* is a type of operation that involves reading a value from memory, modifying it, and then writing the updated value back to memory as a single atomic operation.

VLIW *Very Long Instruction Word* is a CPU architecture that allows multiple operations to be encoded in a single long instruction word, enabling parallel execution of instructions.

Memory

RAM *Random-Access Memory* is a type of computer memory that can be accessed randomly, allowing data to be read or written in any order.

DRAM *Dynamic Random-Access Memory* is a type of volatile memory used in computers and other devices to store data that is being actively used or processed.

SRAM *Static Random-Access Memory* is a type of volatile memory known for its fast access times compared to DRAM.

Cache is a smaller, faster memory component that stores frequently accessed data and instructions to improve overall system performance.

iCache *Instruction Cache* is a specialized cache memory that stores instructions for the CPU to

reduce the time it takes to fetch them from main memory.

dCache *Data Cache* is a specialized cache memory that stores data for the CPU to reduce the time it takes to fetch it from main memory.

TLB *Translation Look-aside Buffer* is a memory cache that stores recent translations of virtual memory to physical memory addresses to improve memory access speed.

CAM *Content Addressable Memory* is a type of memory that allows data to be accessed based on its content rather than its address, enabling fast lookups.

LUT *Lookup Table* is a data structure used to map input values to corresponding output values — typically a resource on FPGAs to store precomputed logic truth tables.

BRAM *Block RAM* is a type of memory resource available on FPGAs that provides high-speed, on-chip storage for data and instructions.

TCAM *Ternary Content Addressable Memory* is a type of CAM that allows for three possible states (0, 1, and "don't care") for each bit in the stored data, enabling more flexible and efficient searching capabilities.

Interconnects

PCIe *Peripheral Component Interconnect Express* is a high-speed serial computer expansion bus standard that connects peripheral devices such as graphics cards, storage devices, and NICs to the CPU.

CXL *Compute Express Link* is an open standard interconnect for high-speed CPU-to-device and CPU-to-memory connections, including CXL.cache a protocol for cache-coherent memory access.

UCIe *Universal Chiplet Interconnect Express* is an open standard for high-speed chiplet-to-chiplet communication within a single package, enabling efficient integration of heterogeneous components.

NVMe *Non-Volatile Memory Express* is a high-performance, scalable host controller interface designed to support PCIe-based storage devices.

DMA *Direct Memory Access* is a feature that allows hardware subsystems to access system memory independently of the CPU.

MMIO *Memory-Mapped Input/Output* is a method of performing input/output operations by mapping device registers into the system's address space.

Interrupt is a signal sent to the CPU that temporarily halts the current execution flow to handle a specific event or condition, such as I/O operations or hardware requests.

MSI-X *Message Signaled Interrupts eXtended* is a method for devices to generate interrupts using in-band messages rather than dedicated interrupt lines, allowing for more efficient and scalable interrupt handling.

Fabric is a general term for high-speed interconnects that connect various components within a computer system or data center, enabling efficient data transfer and communication.

NoC *Network on Chip* is a communication subsystem within a system-on-chip (SoC) that enables efficient data transfer between different components or cores on the chip.

Network Hardware

NIC *Network Interface Controller* is a hardware component that connects a computer to a network.

SmartNIC A type of NIC that includes additional processing capabilities tightly integrated with the network interface, allowing it to offload certain tasks from the host CPU.

Data-path is the path that data packets take as they traverse a network from the source to the destination.

Control plane refers to network components that handle signaling responsible for orchestration, configuration, and management of the data-path.

On-path SmartNIC A SmartNIC that is positioned in the data-path between two communicating endpoints, allowing it to intercept and process network traffic in-motion.

Off-path SmartNIC A SmartNIC that is not directly in the data-path between two communicating endpoints, typically connected via a separate interface.

In-network accelerator A programmable or fixed-function hardware device integrated within the network infrastructure that can process data packets as they traverse the network.

TOE TCP Offload Engine — a specialized hardware component that offloads TCP/IP processing from the host CPU.

SoC *System on Chip* is an integrated circuit that consolidates all components of a computer or other electronic system onto a single chip.

Chiplet is a small integrated circuit that contains a specific functionality and is designed to be combined with other chiplets to create a larger SoC.

ASIC *Application-Specific Integrated Circuit* is a type of integrated circuit designed for a specific application, rather than for general-purpose use.

FPGA *Field-Programmable Gate Array* is a versatile integrated circuit that can be reconfigured by the user to suit specific applications or functions.

NPU *Network Processing Unit* is a specialized microprocessor designed to handle network traffic efficiently, typically found in on-path SmartNICs.

FPC *Flow Processing Core* is a type of programmable processing unit found in some SmartNICs — an alternative term for NPU.

DPU *Data Processing Unit* is also an alternative term for NPU.

GPU *Graphics Processing Unit* is a specialized processor designed to accelerate graphics rendering and parallel processing tasks.

MAU *Match Action Unit* is a processing unit in network hardware that performs packet process-

ing based on matching packet headers to predefined rules and executing corresponding actions.

RMT *Reconfigurable Match-Action Tables* is a programmable architecture for high-speed packet processing in network devices, typically consisting of a chain of match-action units (MAUs).

Software Frameworks

API *Application Programming Interface* is a defined interface that allows different software components to communicate with each other.

RPC *Remote Procedure Call* is a protocol that allows a program to execute a function on another computer over a network as if it were a local function call.

POSIX *Portable Operating System Interface* is a family of standards specified by the IEEE for maintaining compatibility between operating systems.

P4 *Programming Protocol-Independent Packet Processors* is a high-level programming language designed for programming network devices.

eBPF *extended Berkeley Packet Filter* is a programming framework that allows sandboxed and verified execution of untrusted programs within a privileged context.

XDP *eXpress Data Path* is a high-performance packet processing framework based on eBPF.

DPDK *Data Plane Development Kit* is a set of libraries and drivers for fast packet processing.

SPDK *Storage Performance Development Kit* is a set of tools and libraries for high-performance storage applications.

NVMe-oF *NVMe over Fabrics* is a network protocol that allows NVMe commands to be sent over a network fabric, enabling remote access to NVMe storage devices.

Units and Performance Metrics

BW *Bandwidth*.

Pkts/sec *Packets Per Second*.

Mpps *Million Packets Per Second* is a unit of measurement.

Bpps *Billion Packets Per Second*.

Gbps *Gigabits Per Second* or 10^9 bits per second.

Tbps *Terabits Per Second* or 10^{12} bits per second.

IOPS *Input/Output Operations Per Second*.

QPS *Queries Per Second*.

MOps/sec *Million Operations Per Second*.

μs *Microsecond* or 10^{-6} seconds.

W *Watt* is a unit of power measurement.

JFI *Jain Fairness Index* is a metric used to evaluate the fairness of resource allocation among multiple competing entities in a system.

Xp *Xth Percentile*. Typically, used for reporting tail metrics such as 99p, 99.9p, and 99.99p.

Miscellaneous

AI *Artificial Intelligence*.

ML *Machine Learning*.

RFC *Request for Comments* is a type of publication from the technology community that describes methods, behaviors, research, or innovations applicable to the working of the Internet and Internet- connected systems.

VM *Virtual Machine* is a software emulation of a physical computer that runs an operating system and applications as if they were running on a physical machine.

LRU *Least Recently Used* is a cache replacement policy that evicts the least recently accessed items when new data needs to be loaded into the cache.

FIFO *First In, First Out* is a method for organizing and manipulating a data buffer, where the oldest (first) entry is processed first.

OOO *Out-Of-Order*.

QoS *Quality of Service* refers to the overall performance of a network or service, particularly in terms of its ability to guarantee certain performance levels, such as bandwidth, latency, and error rates.

CRC *Cyclic Redundancy Check* is an error-detecting code used to detect accidental changes to raw data in digital networks and storage devices.

ZC *Zero Copy* is a method of data transfer that allows data to be moved between two locations without the need for intermediate copying by the CPU, reducing CPU overhead and improving performance.

Acknowledgments

I would like to express my deepest gratitude to my advisor, Simon Peter, for his unwavering guidance, support, and patience throughout my research journey. I might have never pursued a Ph.D. at all if not for taking his “Datacenters” course at UT Austin, which first exposed me to quality systems research and sparked my interest in the field. Simon has been an exceptional mentor — generous with his time, thoughtful in his feedback, and always encouraging me to think critically and independently. He never imposed his own views or added undue pressure, yet constantly pushed me to sharpen my ideas and elevate my work. I am especially grateful for his trust and flexibility, including his encouragement to pursue a long-term collaboration with Google, even if it meant diverting time away from projects under his direct supervision. There were many moments of doubt when I felt lost — when projects did not pan out, progress was slow, or demands of research felt overwhelming. Through it all, Simon remained patient and supportive. From him, I learned how to write and present research clearly and effectively — skills that did not come naturally to me. I especially struggled with public speaking, but Simon provided steady encouragement, constructive feedback, and created countless opportunities for me to practice and improve, all of which helped me grow immensely. He also ensured I was always funded and had the resources I needed to succeed. I feel incredibly fortunate to have had the opportunity to work with him, and I will carry his mentorship, generosity, and the lessons I learned from him throughout my career.

I would also like to thank Antoine Kaufmann, who has been an exceptional collaborator throughout my Ph.D. journey. His deep expertise, technical insight, and feedback have been invaluable in shaping this work. I am equally grateful to Tom Anderson and Arvind Krishnamurthy, members of my supervisory committee, for their time and constructive suggestions, which greatly

improved the quality of this work. I am fortunate to have had a committee of such distinguished researchers, whose foundational work in this field is the basis and inspiration for the research presented here.

I am deeply grateful to my collaborators in the Systems Research Group at Google. Working with Google gave me invaluable real-world experience and a rare perspective on the intersection of academia and industry. Kimberly Keeton has been a wonderful mentor and a second advisor during my formative years, continually inspiring me to grow as a researcher. She trusted me with ambitious, large-scale production projects early on, encouraged me to take ownership of complex real-world challenges, and gave me the freedom to pursue and form my own open-ended research questions. The extended Google team — including Scott Hare, Stanko Novakovic, Wei Xu, Suli Yang, Kan Wu, David Culler, Hank Levy, and many others — as well as fellow interns including Zhiyuan Guo, Anil Yelam, Vinay Banakar, and Shaurya Agrawal — have all provided critical feedback, discussions, sounding boards, and collaboration that have significantly enriched my research experience.

I would like to thank my parents for their unconditional love, support, and sacrifices throughout my life. My father, Shashidhara, inspired my intellectual pursuits and instilled in me a deep curiosity about the world. My mother, Sujatha, is the strongest person I know — her resilience, work ethic, and perseverance have taught me invaluable life lessons.

The proverb “it takes a village to raise a child” could not be truer in my case. Towards the end of high school, my life was upended — my father passed away suddenly, and my mother battled two life-threatening illnesses at once. During this difficult time, my extended family — my grandparents, Narasimha Murthy and Lakshmi Devi, aunts and uncles Suma, Rama, Anuradha, Krishna, Shanti, Ramesh, and many others — stepped in with immense care, support, and stability. Their collective strength and kindness over the past decade allowed me to focus on my education and personal growth without the constant worry of my mother’s health, ultimately making it possible for me to pursue a Ph.D. abroad. I am forever indebted to them for their love, faith, and unwavering support.

A Ph.D. can be a lonely journey, and my wife, Manaswini, has been my best friend, companion, and a source of strength and comfort — keeping me company through the best of times and grounded during the most challenging ones. I would also like to thank Manaswini’s parents,

Mamatha and Nagaraj, whose warmth has made this journey even more meaningful.

Finally, I would also like to thank my friends — Bhargavi, Shibalik, Ganesh, Sharmila, Tushar, Deven, Vipul, Prannoy, Biplab, Ajinkya, Sujay, Gabi, Seba, Huzaifa, Ajith, Shreyansh, and many others — for making life outside of research enjoyable and fulfilling. I am also thankful to my peers at UW and UT Austin — Timothy Stamler, Deukyeon Hwang, Aditya Kamath — who have all collaborated with me on various projects.

I conclude by expressing my heartfelt appreciation to everyone who has made this achievement possible.

Chapter 1

Introduction

Cheap, reliable, versatile, and high-performance network communication has been instrumental in driving the exponential growth of data and computing demands over the past decade [260, 237]. Today, nearly all cloud-scale applications are designed as microservices, relying on low-latency Remote Procedure Calls (RPCs) to achieve scalability [252, 108, 73, 34, 169, 295, 133]. Emerging paradigms like serverless computing push these boundaries further by deconstructing applications into granular, microsecond-scale functions that externalize all state to remote storage [126, 148, 254, 294, 280]. Meanwhile, the rapid growth of Machine Learning (ML) workloads has significantly increased the demand for terabit-scale network bandwidth to keep GPUs fully utilized [299], driven by the growing complexity and size of deep learning models and datasets [30, 234, 86, 104, 164, 302, 208, 139, 134]. Additionally, infrastructure providers are increasingly adopting disaggregated architectures that decouple memory, compute, and storage resources, enabling independent scaling and provisioning. This approach relies heavily on network infrastructure to transfer vast amounts of data between resource pools [179, 300, 88,

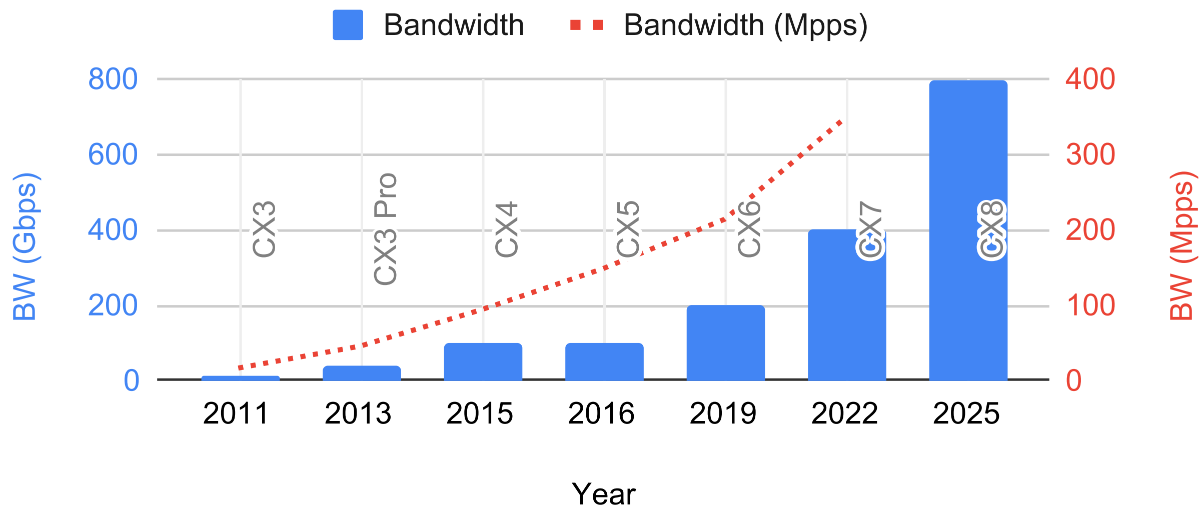

Figure 1.1: Mellanox/NVIDIA ConnectX Network Interface Card (NIC) bandwidth over the years. Source: (1) NVIDIA specifications [173, 174, 203, 204, 205, 206, 209]; (2) DPDK performance reports [221, 220, 217, 216, 215, 214, 213, 212, 211].

26, 14, 94, 15]. Consequently, modern data center workloads impose unprecedented demands on end-host network stacks.

1.1 Challenges for Data Center TCP Stacks

Today, TCP remains the de-facto transport protocol for data center applications due to its robust delivery semantics, mature implementation and tooling, and broad compatibility with applications and commodity network hardware [252, 67, 179, 4, 86]. However, as modern network hardware scales — roughly doubling peak bandwidth and packet rates every 2–3 years (Figure 1.1), software-based end-host TCP stacks struggle to keep pace. Despite numerous hardware and software innovations [227, 160, 120, 32, 228, 138, 120, 161, 162, 45], these stacks fail to effectively translate hardware gains to applications, a gap exacerbated by stagnating CPU speeds [243]. For instance, even highly-optimized kernel-bypass TCP stacks consume up to

48% of per-request CPU cycles for communication-intensive short RPC workloads (§2.1), while the Linux kernel network stack scales to only ∼42Gbps per-core for large flows [44]. Fully utilizing a 100Gbps NIC can demand as many as 20 cores solely for networking tasks [224, 223]. As network speeds outpace CPU capacity, these inefficiencies increasingly tax precious CPU resources, reducing productivity and raising power consumption, a challenge further amplified by emerging server designs that rely on compute-constrained, low-power CPUs to orchestrate data movement across high-bandwidth accelerators and disaggregated resource pools [226, 261, 182, 156].

Offloading transport processing is a promising avenue to reduce CPU overhead. Limited offloads of stateless functions into the NIC, such as checksum computation and segmentation offload, have been commonplace for some time [262]. In recent years, however, the unsustainable CPU cost of software network stacks has driven data center operators to explore large-scale deployments of more comprehensive ASIC-based transport offloads, including full TCP offload engines (TOEs) and RDMA-enabled NICs (e.g., iWARP and RoCEv2) [63, 51, 181]. These hardware transports deliver high performance with near-zero CPU overhead and low power consumption, achieving substantial improvements in performance-per-watt compared to CPU-based stacks [206, 219]. Despite these benefits, they sacrifice flexibility, constraining protocol evolution to meet emerging application or deployment needs, which in turn leads to operational issues and deployment challenges at scale [186, 80, 172]. Notably, RDMA-based transports were originally designed for tightly controlled, lossless fabrics at smaller scales, and retrofitting them to operate efficiently and reliably in modern large-scale, congestion-prone data center networks has proven difficult and error-prone, requiring years of slow hardware cycles to mature [102]. Data center networks, application characteristics, and protocols, including TCP [16, 155, 121, 172, 67], all evolve rapidly and impose diverse requirements; operators

therefore prioritize flexibility for debugging, tracing, managing, and adapting the network stack with high velocity [172, 270]. For instance, data center operators like Google emphasize manageability and development agility over raw performance [84]. Consequently, hardware transports are still typically deployed only in specialized, siloed environments such as storage and AI clusters [86, 26, 88, 234, 267], while software-based stacks remain the default option.

1.2 Objectives

Building upon the challenges presented, any practical data center transport stack for the terabit era must simultaneously achieve the following objectives:

- **Performance:** The transport stack must deliver hardware-competitive performance, minimizing the gap between application-level experience and underlying network hardware capabilities. It should sustain *terabit-scale throughput* for streaming workloads and achieve *high message rates* at *microsecond-scale latencies* for short RPC workloads. It must also *scale* gracefully to tens of thousands of concurrent connections and applications, while ensuring *performance isolation* among workloads. Moreover, since rare performance outliers can cascade to determine overall application behavior at data center scale [66], the stack must ensure *predictable performance* even under load. Finally, *resilience* to packet loss, reordering, and congestion is an essential requirement for maintaining reliable and consistent performance.

- **Energy Efficiency:** The transport stack must optimize *performance-per-watt*, approaching the efficiency of fixed-function ASICs, even as the network speeds scale beyond the terabit regime. To do so, it must *reduce CPU overhead*, conserving general-purpose compute resources for higher-level application logic.

- **Flexibility:** The transport stack must provide *agile software programmability* and *expressiveness* to adapt to the continuously evolving data center network requirements. Equally important are *observability* and *debuggability*, essential for maintaining operational efficiency at scale. The stack should embrace high-level programming models that accelerate iteration and development velocity. Flexibility is also vital for security and policy enforcement in multi-tenant environments [138, 246, 31]. Finally, an important dimension of flexibility is *generality* — the ability to support diverse application workloads without constraining design choices. Tight application-protocol-hardware couplings can boost performance but sacrifice generality, as developers overfit to specific hardware and transport features, complicating software reuse, management, and development agility.

1.3 Opportunity: Programmable In-Network Accelerators

Emerging programmable in-network hardware accelerators, such as those found in SmartNICs and switches [195, 59, 85, 175, 48, 40, 21, 58, 41, 42, 20, 210, 180], offer a path forward combining high performance and efficiency with programmability. These devices integrate specialized compute units and hardware accelerators closely with the network interface, that better exploit the inherent parallelism in network processing tasks, achieving significantly higher performance and efficiency relative to server-class CPUs.

These hardware platforms span a diverse design space, including general-purpose efficiency cores, Field-Programmable Gate Arrays (FPGAs), Network Processing Units (NPUs), and Reconfigurable Match-Action Table (RMT) pipelines, each presenting distinct trade-offs among programmability, performance, and efficiency. Low-power cores offer general-purpose flexibility but lack specialization for efficient packet processing; FPGAs excel at fine-grained parallelism but

demand low-level hardware design expertise; NPUs are optimized for highly-parallel packet processing tasks requiring limited co-ordination while retaining high-level programming toolchains; and RMT pipelines deliver ASIC-class line-rate performance and efficiency under a constrained programming model.

These architectures have proven highly effective for a wide range of network processing tasks, particularly those that are embarrassingly parallel or stateless, such as routing, filtering, and telemetry [180]. However, their efficiency gains often comes at the expense — of varying degrees depending on the design — of poor single-thread performance, increased programming and resource management complexity, and limited expressiveness for stateful computations and sophisticated program flows. Consequently, implementing inherently sequential, complex stateful code-paths of protocols like TCP — requiring precise tracking of in-flight segments sensitive to packet reordering — remains a significant challenge. There are no clear winners in this trade-off space, and evaluating these architectures for stateful TCP offload — using high-level programming frameworks and with end-to-end integration with real applications running on the host — thus far remains largely unexplored.

1.4 Thesis Statement

> By introducing fine-grained data-path parallelization design principles for mapping stateful TCP logic onto programmable network hardware, this thesis demonstrates that it is possible to design and implement data center TCP stacks that surpass CPU-based implementations in performance, rival ASIC-based transports in energy efficiency, and retain a high degree of flexibility using agile software programmability, and generality across diverse application classes. More broadly, this thesis shows

that *flexibility*, *performance*, and *efficiency* — often viewed as competing objectives — can be meaningfully balanced in practical TCP stack designs for modern data center environments.

In this work, we investigate the offload and acceleration of TCP stacks on leading programmable network hardware architectures. Specifically, we present the design and implementation of two systems: **FlexTOE** (§4) and **Laminar** (§5), targeting Network Processing Unit (NPU) and Reconfigurable Match-Action Table (RMT)-pipeline based architectures, respectively. Both stacks exceed CPU-based software stacks in performance and efficiency — nearing hardware-competitive levels — while crucially retaining flexibility for full customization of transport logic. Unmodified applications interface transparently with these stacks, by simply linking with *libTOE* library, which implements POSIX sockets. Importantly, neither system requires specialized data center network features (such as lossless fabric), and both interoperate seamlessly with other TCP stacks, and remain robust to packet loss and congestion. The two designs explore distinct points in the flexibility-performance-efficiency trade-off space, reflecting the characteristics of their underlying hardware architectures: FlexTOE prioritizes flexibility through programmability in high-level languages, while Laminar achieves exceptional ASIC-like efficiency under a more constrained programming model. Finally, we show that the design principles developed in this work generalize across other programmable network hardware platforms, providing benefits on low-power CPU cores, FPGAs, and even on host CPUs themselves.

While rooted in TCP, the insights gained are broadly applicable beyond any single protocol or architecture. The challenges of mapping stateful protocols to programmable hardware, managing limited resources, and optimizing performance are universal to programmable data planes. By addressing these challenges, we inform the debate on programmability versus efficiency, influencing the design of next-generation hardware, including SmartNICs, FPGA-

based accelerators, chiplet-based heterogenous processors, and hybrid systems that integrate programmable and fixed-function components, and the evolution of hardware-efficient transport protocols for data centers.

1.5 Summary of Contributions

Concretely, we make the following contributions:

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

- Stateful TCP data-path offload is challenging. Programmable dataplane devices are resource-constrained platforms geared towards massively parallel processing that operate under strict timing constraints, lacking support for complex stateful computation and sophisticated program flow. On the other hand, TCP has complex, stateful code paths to track in-flight segments, perform congestion control, and is extremely sensitive to packet reordering. Resolving the gap between TCP's requirements and hardware capabilities requires careful placement of state and logic. Conventional TCP stack designs are either inefficient or impractical on these architectures, necessitating substantial adaptations and novel design principles for viable high-performance TCP processing. We present the design of FlexTOE and Laminar, detailing *fine-grained data-path parallelization* techniques used to adapt TCP to the restrictive hardware architectures.

- We implement FlexTOE and Laminar on commercially available instantiations of their respective hardware architectures, specifically the Netronome Agilio-CX40 [195] and

Intel Tofino2 switch [59]. Using the design principles, we are the first to demonstrate that these hardware platforms can support high-performance, yet flexible TCP data-path offload. To enable researchers and industry to build upon our findings, the entire code will be made open-source at: https://github.com/tcp-acceleration-service/.

- Through extensive benchmarking across diverse workloads, we evaluate both FlexTOE and Laminar, highlighting their ability to strike a practical balance among flexibility, performance, and efficiency compared to traditional TCP stacks. Additionally, we demonstrate the generalizability of our design principles by applying them to other programmable hardware platforms, including low-power CPU cores and FPGAs, achieving notable performance and efficiency improvements.

1.6 Published Works

- Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. 2022. **FlexTOE: Flexible TCP offload with Fine-Grained parallelism**. *In 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*.

- Rajath Shashidhara, Antoine Kaufmann, and Simon Peter. 2025. **Laminar: A Match-Action TCP Stack for the Terabit Era**. *Under review*.

An early version of FlexTOE was included in my Master's thesis:

- Rajath Shashidhara. 2021. **TASNIC: A Flexible TCP offload with Programmable SmartNICs**. *Master's thesis, The University of Texas at Austin*.

This version expands the early TAS [138] to Netronome [195] port into a full-fledged system,

adding general design principles, complete TCP implementation, framework for flexibility, broader evaluation, and generalization to other hardware platforms.

1.7 Outline

The remainder of this thesis is organized as follows. Chapter 2 motivates the need for TCP offload, examines the flexibility and efficiency trade-offs of modern programmable network accelerators, and outlines key challenges for TCP offload to such platforms. Chapter 3 introduces the shared design principles of FlexTOE and Laminar, including their fine-grained data-path parallelization architecture. Chapters 4 and 5, detail and evaluate the FlexTOE and Laminar on NPU and RMT architectures, showcasing their ability to reach exceptional efficiency with flexibility. Chapter 6 reviews related work, while Chapter 7 discusses limitations and generalization beyond TCP. Finally, Chapter 8 concludes the thesis and outlines future research directions.

Chapter 2

Background

We motivate the need for TCP offload by analyzing host CPU processing overheads of existing approaches (§2.1). Next, we outline the core processing characteristics of leading in-network accelerator hardware architectures for flexible packet processing (§2.2), followed by a discussion of existing TCP stack designs and their incompatibility with these hardware architectures (§2.3), motivating the adaptations necessary to map TCP processing onto them (§2.4).

2.1 TCP Impact on Host CPU Performance

Two dominant traffic motifs characterize data center networks [16, 33, 241, 252, 44], each stressing different aspects of the end-host transport stack:

- **Latency-sensitive RPC workloads** typically maintain thousands of persistent connections, with median packet sizes under 200B, each leaving only a few packets in-flight at any given time, imposing extremely high packet processing rates (packets/sec) on the transport stack.

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

Table 2.1: CPU overhead of TCP processing per Memcached request.

- **Bandwidth-intensive large flows**, prevalent in storage and AI workloads, requires saturating the line rate with few connections, where payload copying overheads dominate.

We quantify the impact of different TCP processing approaches on host CPU performance in terms of CPU overhead, execution efficiency, and cache footprint, when processing common RPC-based workloads. We do so by instrumenting a single-threaded Memcached [178] server application using hardware performance counters (cf. §4.5 for details of our testbed). We use the popular `memtier_benchmark` [240] to generate the client load, consisting of 32 B keys and values, using as many clients as necessary to saturate the server, executing closed-loop KV transactions on persistent connections. Table 2.1 shows a breakdown of our server-side results, for each Memcached request-response pair, into NIC driver, TCP/IP stack, POSIX sockets, Memcached application, and other factors.

Due to their substantial packet processing overhead, existing TCP stacks struggle to deliver optimal performance and energy efficiency for latency-sensitive RPC workloads — the primary focus of this analysis. While bandwidth-intensive large flows are often dominated by payload copy overheads [44], these pose distinct challenges that can be partially mitigated through hardware-assisted copy optimizations [270, 51, 44].

In-kernel. Linux's TCP stack is versatile but bulky, leading to a large cache footprint, inefficient execution, and high CPU overhead [44]. Stateless offloads [262], such as segmentation and generic receive offload [114], reduce overhead for large transfers, but they have minimal impact on RPC workloads dominated by short flows. We find that Linux executes 12.13 kc per Memcached request on average, with only 10% spent in the application. Not only does Linux have a high instruction and instruction cache (Icache) footprint, but privilege mode switches, scattered global state, and coarse-grained locking lead to 62% of all cycles spent in instruction fetch stalls (frontend bound), cache and TLB misses (backend bound), and branch mispredictions (cf. [138]). These inefficiencies result in 1.33 instructions per cycle (IPC), leveraging only 33% of our 4-way issue CPU architecture. Linux is, in principle, easy to modify, but kernel code development is complex and security sensitive. Hence, introducing optimizations and new network functionality to the kernel is often slow [172, 201, 200].

Kernel-bypass. Kernel-bypass, such as in mTCP [120] and Arrakis [228], eliminates kernel overheads by entrusting the TCP stack to the application, but it has security implications [246]. TAS [138] and Snap [172] instead execute a protected user-mode TCP stack on dedicated cores, retaining security and performance. By eliminating kernel calls, TAS spends only 800 cycles in the socket API—31% of Linux's API overhead. TAS also reduces TCP stack overhead to 34% of Linux. TAS reduces Icache footprint, front and back-end CPU stalls, improving IPC by 40% versus

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

Table 2.2: Breakdown of TCP/IP stack overheads in TAS.

Linux, and reducing the total per-request CPU impact to 27% of Linux. However, kernel-bypass still has significant overhead. Only 26% of per-request cycles are spent in Memcached—the remainder is spent in TAS.

Table 2.2 shows a breakdown of the per-packet TCP/IP processing overheads (summarized as *TCP/IP stack* in Table 2.1) in TAS. For each request, TAS performs loss detection (and potentially recovery) that involves processing the incoming request segment, generating an acknowledgement for it, and additionally, processing the acknowledgement for the response segment, consuming 42% of the total per-packet processing cycles. TAS spends 9% of the total cycles to prepare the response TCP segment for transmission and an additional 12% to schedule flows based on the rate configured by the congestion control protocol. TAS spends 26% of per-packet cycles interacting with the application, to notify when a request is received, to admit a response for transmission, and to free the transmission buffer when it is acknowledged. For small request-response pairs (32B in this case), the payload copy overheads are negligible.

Inflexible TCP offload. TCP offload can eliminate host CPU overhead for TCP processing. Indeed, TOEs [63] that offload the TCP data-path to the NIC have existed for a long time. Existing approaches, such as the Chelsio Terminator [51], hardwire the TCP offload. The

resulting inflexibility prevents data center operators from adapting the TOE to their needs and leads to a slow upgrade path due to long hardware development cycles. For example, the Chelsio Terminator line has been slow to adapt to RPC-based data center workloads.

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

2.2 Programmable In-Network Accelerators

Despite extensive optimizations, software TCP stacks on general-purpose CPUs (§6) remain constrained by limited parallelism and high energy consumption [22, 168]. In contrast, specialized hardware platforms, such as SmartNICs and programmable switches, integrate specialized packet processing engines tightly with the network interface, offering a spectrum of trade-offs between programmability, performance, and efficiency. We survey these platforms below.

2.2.1 High-Efficiency CPU Cores

Several commercially available SmartNICs, such as NVIDIA Bluefield [175, 218, 222], Broadcom Stingray [40], Marvell Octeon [49] and Intel IPU [58], integrate high-efficiency, general-purpose CPU cores (typically ARM, RISC-V, or MIPS), directly on the NIC to enable programmability.

These cores may be integrated as either *off-path* or *on-path* processors, depending on whether they reside outside or within the critical path of network packet flow, respectively. For instance, NVIDIA BlueField-I and BlueField-II SmartNICs are off-path designs that route traffic between the Ethernet interface, the host, and the on-board ARM cores via an internal PCIe switch. Both host CPU cores and on-board ARM cores must traverse this PCIe interface to process packets, offering little inherent performance advantage in using the on-board ARM cores [249]. In contrast, Octeon LiquidIO SmartNICs employ on-path integration, giving NIC cores direct access to the Ethernet interface and hardware accelerators for buffer management and packet scheduling, though all traffic must traverse these cores [49].

While these on-board cores are significantly weaker than host CPU cores [168, 166], they are versatile enough to run Linux, with offloads implemented as user-space applications. This preserves software flexibility but limits scalability and efficiency — e.g., a BlueField-II consumes up to 65W for just 8 ARM cores [218], insufficient to sustain the packet rates of RPC workloads (cf. §5.5.1.4). Control-flow and I/O-intensive workloads can still benefit from offloading to these wimpy cores [168, 166, 143, 107].

2.2.2 Field-Programmable Gate Arrays (FPGAs)

FPGAs are a powerful and highly expressive platform for high-performance network offload [233, 152]. By compiling offloads directly into hardware logic, they enable massive parallelism and deterministic low-latency processing, offering strong performance for many network functions [152].

Their near-hardware programming model [157] demands careful design: complex, stateful, and inherently sequential protocols are challenging to implement efficiently [152]. High-level

synthesis (HLS) tools ease programmability, but often produce suboptimal hardware that still requires hardware expertise to refine [146].

FPGAs can also be power-hungry at high clock frequencies, sometimes exceeding the PCIe power envelope (e.g., 100 Gbps FPGA NICs can draw up to 250 W [20]). Power efficiency depends heavily on design style — deeply pipelined implementations can reduce frequency and power, but complicate consistency and state management, and increase chip area and latency [23]. While FPGAs typically consume more power and chip area than ASICs and remain costlier [37], they offer a compelling balance of flexibility, performance, and efficiency when carefully architected.

2.2.3 Network Processing Units (NPUs)

NPU-enabled SmartNICs (e.g., Netronome Agilio [195, 194], Pensando Elba [21], NVIDIA Bluefield-III [222, 52]), integrate large arrays ($\sim$ 100s) of purpose-built network processors (NPUs) designed for massively-parallel *on-path* packet processing. NPUs feature restrictive Instruction Set Architectures (ISAs), limited instruction and data memory, poor single-thread performance, deep memory hierarchies, and often forego cache-coherency for efficiency. They are typically augmented with specialized accelerators — such as Content-Addressable Memory (CAM), parsers, and hash engines — that significantly boost network processing efficiency over general-purpose CPUs. For instance, with hardware-assisted multi-threading, the Agilio-CX40 can process up to 480 packets in parallel, sustaining 40Gbps processing at a modest 20W [195] peak power usage, offering far better efficiency than host CPUs. These NICs remain software programmable through high-level C/eBPF based frameworks, though achieving high performance still demands deep understanding of their parallel execution model and extensive manual tuning [52, 158] (cf. §4).

2.2.4 Reconfigurable Match-Action Table (RMT) Pipeline

RMT-pipeline architecture has emerged as a predominant design for line-rate programmable packet processing. It consists of a sequential pipeline composed of identical pipeline stages with local memory, each executing simple operations — such as modifying packet header fields, or updating stage-local state — under strict timing and resource constraints, in a synchronized, lockstep manner. Each stage is programmable typically via a high-level programming language like P4 [43, 57]. This design in switches, such as Intel Tofino2, delivers over 12.8 Tbps and 6 Billion packets per second at roughly 400 ns of latency and ASIC-class power consumption [59, 13], far surpassing CPU-based processing in both performance and efficiency. Similarly, AMD Pensando SmartNICs integrate an RMT-pipeline achieving 400G line rate packet processing at just 50W [21].

However, this ASIC-like performance and efficiency comes at the cost of programmability: the unidirectional, stage-local execution model with fixed timing limits restricts expressiveness, especially for complex protocols with dependent states. Still, the P4 language simplifies programming for many flexible network functions. For instance, the RMT-pipeline is also a programming abstraction — more powerful NPU-based SmartNICs [195, 222, 21] and FPGAs [19], can emulate RMT-pipeline behavior for programming ease.

2.3 Existing TCP Stack Architectures

The dominant TCP stack architectures on general-purpose CPUs — including the Linux in-kernel TCP stack, and high-performance kernel-bypass stacks — follow a *monolithic, run-to-completion processing* model, where a single CPU core fully executes the TCP processing for each TCP packet or socket operation. Scalability is achieved by distributing connections across threads,

with hardware-assisted flow steering for load balancing [44, 138].

This design worked well up to 40 Gbps link bandwidths, where single-core performance could saturate the network bandwidth with a single flow. However, it no longer suffices for modern high-speed networks [44]. As discussed earlier, different TCP workloads stress different aspects of the transport stack. When the performance bottleneck was the network, such dedicated, static designs were acceptable; but with host CPU performance now critical, this design lacks mechanisms to carefully allocate resources — including CPU cores, NIC queues, packet buffers — among competing network traffic with diverse performance needs (cf. §6).

To alleviate this bottleneck, recent proposals have modularized the network stack into coarse-grained components that scale independently across cores [172, 45]. Even so, the TCP transport logic remains monolithic within each stack, limiting the achievable parallelism and efficiency. FPGA-based TCP offloads [23, 156] also largely follow similar designs, implementing per-connection TCP processing in circuit form that executes within a single clock cycle. Increasing bandwidths then requires reducing clock cycle time, which presents a significant design challenge for any flexible enhancement and increases power consumption.

Such monolithic architectures are either inefficient or impractical on modern in-network accelerators — including NPU, RMT, and FPGA platforms. Efficient offload necessitates fine-grained parallelization and careful placement of TCP state and logic. NPUs have poor single-thread performance and depend on massive multi-threading, RMT-pipelines rely on instruction-level pipeline parallelism, and FPGAs benefit from deep pipelining for scalability and efficiency.

2.4 Discussion

In-network accelerators offer a compelling path forward for high-performance, efficient, and flexible TCP offload. Yet, no single architecture is a panacea. Low-power CPU cores provide versatility, but fall short on performance. NPU and RMT architectures deliver exceptional efficiency, but require careful design to overcome their programmability constraints and extract performance. FPGAs provide both power and expressiveness but demand significant hardware expertise to achieve high efficiency.

Modern SmartNICs are increasingly hybrid, combining CPUs, NPUs, and FPGAs to balance programmability, efficiency, and performance. For instance, NVIDIA BlueField-III integrates ARM cores with NPU-based DPUs [222], while AMD Pensando SmartNICs pair an RMT pipeline with ARM cores [21]. Such hybrid designs enable offloads to exploit the strengths of multiple architectures.

Stateful TCP offload on these platforms presents unique challenges. TCP is inherently sequential and relies on complex, stateful logic to track in-flight segments to perform reassembly, retransmission, and congestion control, while remaining highly sensitive to packet reordering. Conventional TCP stack designs are incompatible with these architectures. Extracting parallelism and mapping TCP's intricate code paths and state efficiently within the constraints of these architectures require substantial rethinking of stack design.

This design space remains largely unexplored — especially for stateful TCP offload. In this thesis, we focus primarily on two distinct architectures, NPU and RMT, to investigate the challenges and design principles for efficient TCP offload on in-network accelerators. The techniques we develop, however, generalize across architectures, including FPGAs, CPUs, and hybrid designs.

Chapter 3

High-Level Design

We now present the high-level architecture (§3.1), along with the overarching parallelization design principles (§3.2) of our TCP offload systems, FlexTOE and Laminar.

3.1 Offload Architecture

We are guided by Amdahl's Law [101] in our design, recognizing that accelerating uncommon paths often outweighs the benefits. Instead, isolating performance-critical paths from less critical or infrequently executed ones simplifies design and optimization. Separating frequently modified or evolving components further improves manageability. Decomposing the TCP stack along performance and flexibility boundaries allows tailoring each to the most appropriate accelerator architecture.

3.1.1 Components

We build on the TAS host stack architecture [138], whose key contribution is the separation of the TCP stack into distinct components: a streamlined fast-path that handles common, performance-sensitive per-packet processing; a slow-path for uncommon or complex policy and control operations; and an efficient application interface layer that enables direct interaction with both paths via kernel-bypass. We preserve this high-level architecture in both FlexTOE and Laminar. The common-case fast-path is accelerated on NPU and RMT architectures, respectively, while the slow-path remains on on-NIC efficiency CPUs or the host. The application interface layer is adapted to enable low-latency interaction over accelerator interconnects such as PCIe.

Below, we present a high-level overview of each component. For clarity, we refer to the fast-path as the data path, the slow-path as the control plane, and the application interface layer as libTOE in the remainder of this work.

Data-path. The data-path is responsible for scalable data transport of established connections: TCP segmentation, loss detection and recovery, rate control, payload transfer between socket buffers and the network, and application notifications.

Control Plane. The control plane oversees connection and context lifecycle operations, including establishment and teardown. It also handles the congestion control policy and manages other uncommon scenarios, such as timeouts, that have non-constant per packet overhead or are too expensive or stateful to process in the data-path. Additionally, it is responsible for accelerator platform management, including initialization, configuration, and resource allocation of the data-path resources and interfaces. It may run either on the host or on a control CPU alongside the acceleration platform, in its own protection domain, isolated from untrusted applications.

Application Interface (libTOE). The application interface is provided by libTOE, a dynamically linkable library. It exposes a POSIX-compatible socket API, enabling seamless integration with existing applications. It may be pre-loaded using `LD_PRELOAD` or linked directly. It also provides a zero-copy API, libTOE-ZC, that allows data-intensive applications to bypass socket buffer copies for improved performance.

3.1.2 Interfaces

The three components interact via well-defined, optimized interfaces over the underlying hardware interconnects, depending on the placement of each component.

Context Queues (CTX-Qs). A libTOE instance is referred to as a *context*. Typically, each application thread uses a separate context for scalability, akin to per-core NIC queues in traditional network stacks[1]. Each context interacts with the control-plane and data-path using dedicated *context queue pairs (CTX-Qs)* — efficient message-passing queues optimized for shared memory, PCIe, or other proprietary interconnects. Context queues also support doorbell mechanisms to notify the arrival of new events. This enables the data-path to interrupt idle contexts when data arrives, or enabling the context to trigger data-path processing and batch multiple requests efficiently [77].

Connection Payload Buffers. For each established connection, the context allocates per-connection *transmit (TX) and receive (RX) socket buffers (PAYLOAD-BUFs)* in host memory. These buffers remain DMA accessible to the data-path for efficient zero-copy data transfer. libTOE appends data for transmission into the per-socket TX PAYLOAD-BUF and notifies the data-path

[1]This is not a strict requirement; contexts can be shared across threads, though this may introduce contention. Ensuring thread safety is the responsibility of the application.

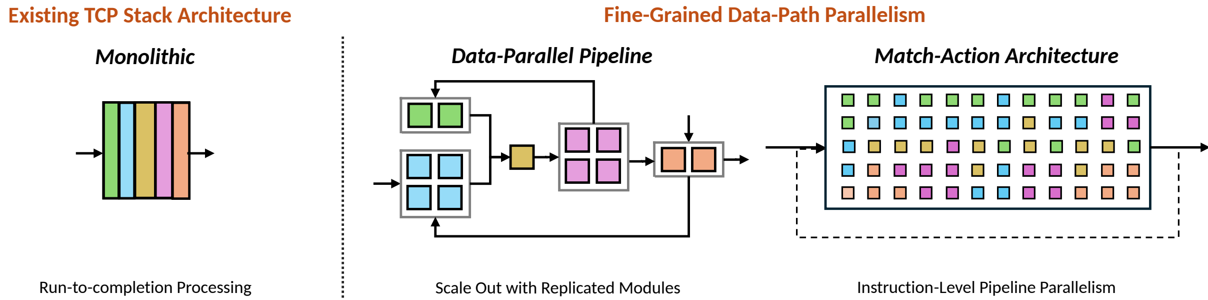

Figure 3.1: Fine-Grained TCP Data-Path Parallelization

using a thread-local CTX-Q. The data-path appends received data stream to the sockets' RX PAYLOAD-BUFs after reassembly and libTOE is notified via the same thread-local CTX-Q. For zero-copy access, libTOE-ZC allows applications direct access to these PAYLOAD-BUFs.

3.2 Fine-Grained Data-path Parallelization

Different in-network accelerator architectures enable diverse parallelism models (§2.2). Effective parallelization strategy and placement of TCP state and logic are necessary conditions for leveraging these accelerators efficiently. A core design contribution of this thesis is the *fine-grained TCP data-path parallelization*. Below, we provide a high-level overview of how parallelization is achieved, depicted in Figure 3.1, deferring architecture-specific details to Chapters §4 and §5.

3.2.1 Data-Parallel Pipeline

Conventional monolithic TCP stack designs exceed the instruction and memory footprint of NPUs, and sequential instruction execution is much slower than on host CPUs (§2.2.3). Efficient NPU offload therefore must exploit all available parallelism to scale to a large pool of NPUs, and limit the per-NPU execution footprint to achieve high performance. In FlexTOE, we decompose

the TCP data-path into a data-parallel execution pipeline: each specialized module keeps private state, communicates explicitly, and may scale independently to maximize parallelism (§4). This modularity eases also flexibility — modules may be inserted, removed, or modified into the pipeline (§4.5). Moreover, plateauing single-threaded performance on CPU cores makes monolithic designs increasingly untenable (§2.3). We find that this model is beneficial even on CPUs (§4.5), particularly to scale single connection throughput beyond the capacity of a single core (cf. [45]).

3.2.2 Match-Action Architecture

Even with data-parallel pipelines, the TCP transport logic remains a single module (cf. *Protocol* module in FlexTOE) that serializes per-connection state updates. This is incompatible with unidirectional execution, limited per-stage resources, and strict processing time budgets of RMT architectures (§2.2.4). Adapting TCP to the RMT match-action model requires breaking down TCP state and its dependencies at the individual instruction level into simple Match-Action operations, and spreading them across the pipeline stages. Laminar is the first system to achieve full pipeline parallelism for TCP state management (§5), enabling line-rate, stateful TCP processing on RMT hardware. This approach also benefits FPGA-based designs, which often largely implement per-connection stateful TCP processing in circuit form to execute within a single clock cycle (§2.3), by loosening timing constraints and improving energy efficiency (§5.5).

Chapter 4

FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism

FlexTOE is a flexible, yet high-performance TCP offload engine (TOE) to network processor (NPU) based SmartNICs. FlexTOE focuses on scenarios that are common in data centers, where connections are long-lived and small transfers are common [172]. FlexTOE offloads the TCP data-path to network processors (NPUs), enabling full customization of transport logic and flexibility to implement data-path features whose requirements change frequently in data centers. Unmodified applications interface directly but transparently with the FlexTOE data-path through the *libTOE* library that implements POSIX sockets, while FlexTOE offloads all TCP data-path processing (§2.1). FlexTOE interoperates well with other TCP stacks, is robust under adverse network conditions.

TCP data-path offload to NPU SmartNIC architecture is challenging (§4.1.1). NPU SmartNICs support only restrictive programming models with stringent per-packet time budgets and are

geared towards massive parallelism with wimpy cores [196]. They often lack timers, as well as floating-point and other computational support, such as division. Finally, offload has to mask high-latency operations that cross PCIe. On the other hand, TCP requires computationally intensive and stateful code paths to track in-flight segments, for reassembly and retransmission, and to perform congestion control [23]. For each connection, the TCP data-path needs to provide low processing tail latency and high throughput and is also extremely sensitive to reordering.

Resolving the gap between TCP's requirements and SmartNIC hardware capabilities requires careful offload design to efficiently utilize SmartNIC capabilities. Targeting FlexTOE at the TCP data-path of established connections avoids complex control logic in the NIC. FlexTOE's offloaded data-path is one-shot for each TCP segment—segments are never buffered in the NIC. Instead, per-socket buffers are kept in per-process host memory where libTOE interacts with them directly. Connection management, retransmission, and congestion control are part of a separate control-plane, which executes in its own protection domain, either on control cores of the SmartNIC or on the host. To provide scalability and flexibility, we decompose the TCP data-path into fine-grained modules that keep private state and communicate explicitly. Like microservices [172], FlexTOE modules leverage a data-parallel execution model that maximizes SmartNIC resource use **and** simplifies customization. We organize FlexTOE modules into a *data-parallel computation pipeline*. We also *reorder* segments on-the-fly to support parallel, out-of-order processing of pipeline stages, while enforcing in-order TCP segment delivery. To our knowledge, no prior work attempting full TCP data-path offload to NPU SmartNICs exists.

We compare FlexTOE implementation on an Agilio-CX40 [195] to host TCP stacks Linux and TAS, and to the Chelsio Terminator TOE. We find that Memcached scales up to 38% better on FlexTOE versus TAS, while saving up to 81% host CPU cycles versus Chelsio. FlexTOE provides competitive

performance for RPCs, even with wimpy SmartNICs. FlexTOE cuts 99.99th-percentile RPC RTT by 3.2× and 50% versus Chelsio and TAS, respectively. FlexTOE's data-path parallelism generalizes across hardware architectures, improving single connection RPC throughput up to 2.4× on x86 and 4× on NVIDIA BlueField SmartNICs [175]. FlexTOE supports C and XDP programs written in eBPF. It allows us to implement popular data center transport features, such as TCP tracing, packet filtering and capture, VLAN stripping, flow classification, firewalling, and connection splicing.

First, we survey the relevant *on-path* SmartNIC architecture (§4.1), followed by design (§4.2) and implementation (§4.4) for efficient offload. Finally, we evaluate FlexTOE on diverse workloads and implement flexible data center networking extensions (§4.5).

4.1 Background

We introduce NPU SmartNIC architecture, using the Netronome NFP-4000 as a representative example (§4.1.1), and discuss implications for flexible TCP offload (§4.1.2).

4.1.1 NPU SmartNIC Architecture

NPU SmartNICs[1], such as Marvell Octeon [49], Pensando Capri [85, 277], and Netronome Agilio [193, 194], support massively parallel packet processing with a large pool of flow processing cores (FPCs), but they lack efficient support for sophisticated program control flow and complex computation [166].

We explore offload to the NFP-4000 NPU, used in Netronome Agilio CX SmartNICs [193]. We

[1] Mellanox BlueField [175] and Broadcom Stingray [40] are off-path SmartNICs that are not optimized for packet processing [166].

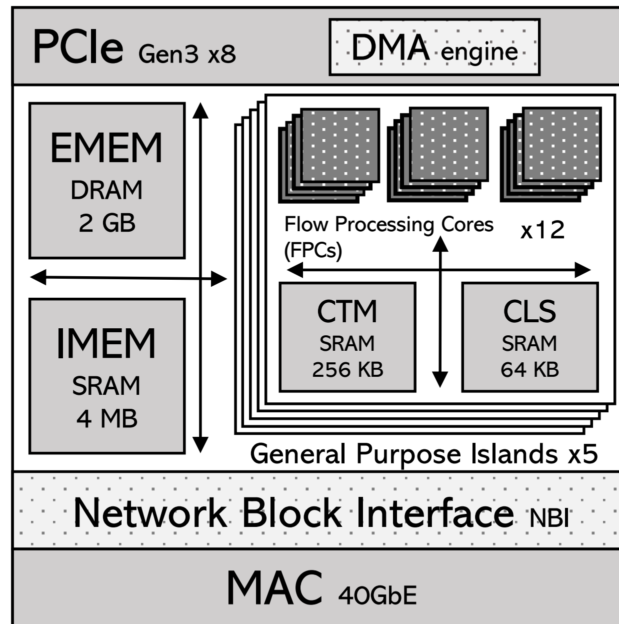

Figure 4.1: NFP-4000 overview.

show the relevant architecture in Figure 4.1. Like other NPU SmartNICs, FPCs are organized into islands with local memory and processing resources, akin to NUMA domains. Islands are connected in a mesh via a high-bandwidth interconnect (arrows in Figure 4.1). The *PCIe* island has up to two PCIe Gen3×8 interfaces and a DMA engine exposing DMA transaction queues [197]. FPCs can issue up to 256 asynchronous DMA transactions to perform IO between host and NIC memory. The *MAC* island supports up to two 40 Gbps Ethernet interfaces, accessed via a *network block interface* (NBI).

Flow Processing Cores (FPCs). 60 FPCs are grouped into five general-purpose islands (each containing 12 FPCs). Each FPC is an independent 32-bit core at 800 MHz with 8 hardware threads, 32 KB instruction memory, 4 KB data memory, and CRC acceleration.

While FPCs have strong data flow processing capabilities, they have small code stores, lack timers, as well as floating-point and other complex computational support, such as division.

This makes them unsuitable to execute computationally and control intensive TCP functionality, such as congestion, connection, and complex retransmission control. For example, congestion avoidance involves computing an ECN-ratio (gradient). We found that it takes 1,500 cycles (1.9 μs) per RTT to perform this computation on FPCs.

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

4.1.2 Implications for Flexible Offload

The NFP-4000 supports a broad range of protocols, but the computation and memory restrictions require careful offload design. More generally, as FPCs are wimpy and memory latencies high, sequential instruction execution is much slower than on host processors. Conventional run-to-completion processing that assigns entire connections to cores [138, 120, 32] results in poor per-connection throughput and latency. In some cases, it is beyond the feasible instruction and memory footprint. Instead, an efficient offload needs to leverage more fine-grained parallelism to limit the per-core compute and memory footprint.

4.2 Overview

In addition to flexibility, FlexTOE has the following goals:

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

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

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

FlexTOE offload architecture. Figure 4.2 shows the offload architecture of FlexTOE, with a host control-plane (each box is a protection domain). libTOE, data-path, and control-plane communicate via pairs of *context queues* (CTX-Qs), one for each communication direction, as explained in Section 3.1. In FlexTOE, we design and integrate a *data-path running efficiently* on the NPU SmartNIC (§4.3).

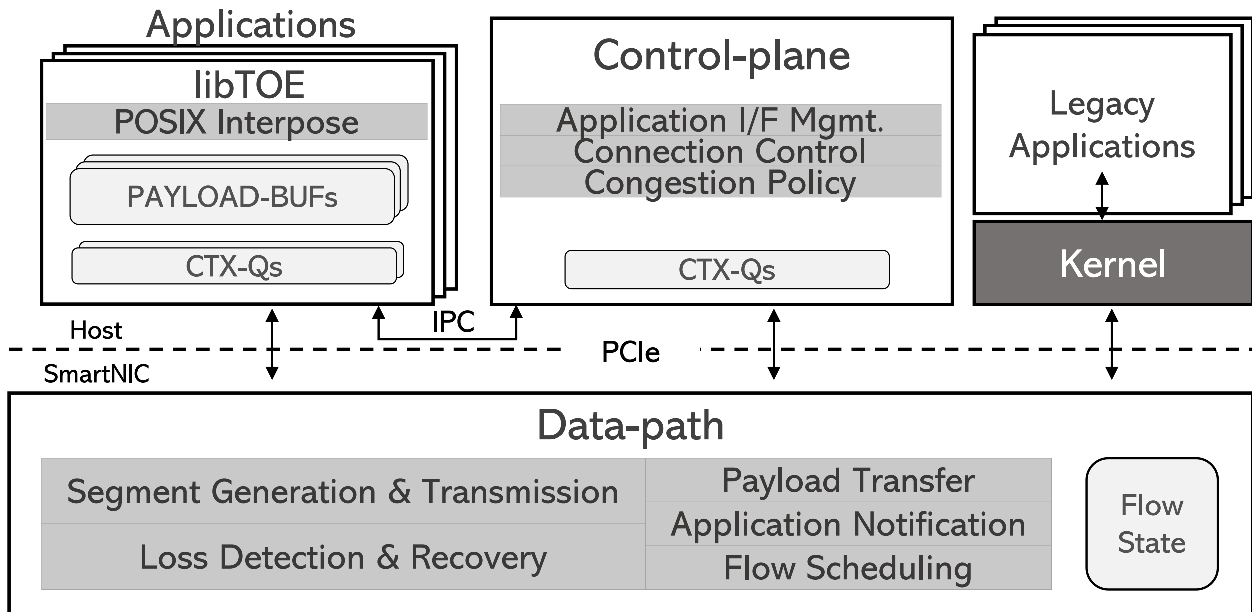

Figure 4.2: FlexTOE offload architecture (host control-plane).

4.3 TCP Data-path Parallelization

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

- **Host control (HC)**: When an application wants to transmit data, executes control operations on a socket, or when retransmission is necessary, the data-path must update the connection's transmit and receive windows accordingly.
- **Transmit (TX)**: When a TCP connection is ready to send—based on congestion and flow control—the data-path prepares a segment for transmission, fetching its payload from a socket transmit buffer and sending it out to the MAC.

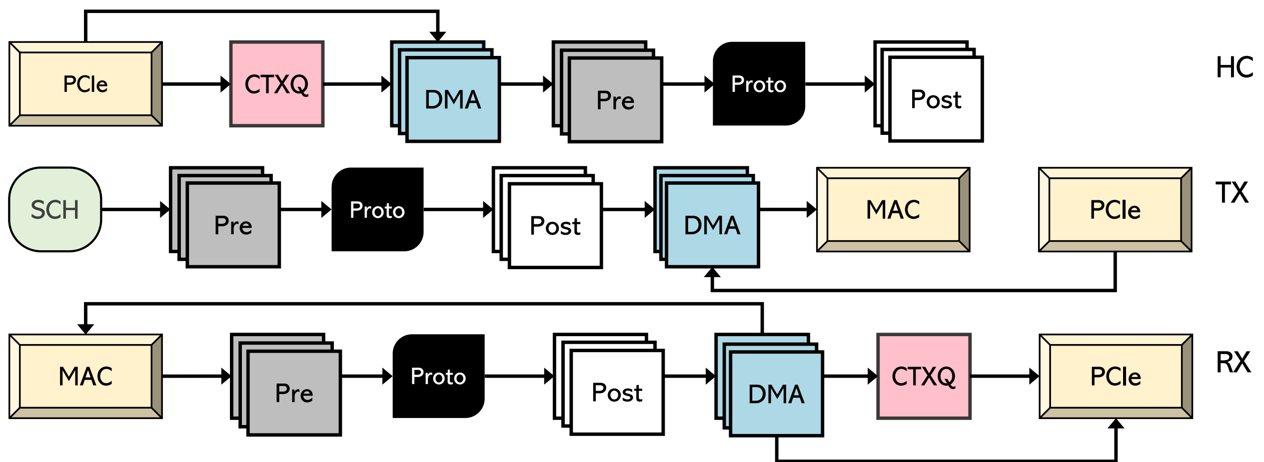

Figure 4.3: FlexTOE per-connection data-path workflows. *Protocol* is atomic. Other stages may be replicated for parallelism.

- **Receive (RX)**: For each received segment of an established connection, the data-path must perform byte-stream reassembly—advance the TCP window, determine the segment's position in the socket receive buffer, generate an acknowledgment to the sender, and, finally, notify the application. If the received segment acknowledges previously transmitted segments, the data-path must also free the relevant payload in the socket transmit buffer.

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

For efficient offload, we decompose this data-path into an up to five-stage parallel pipeline of processing modules: *pre-processing*, *protocol*, *post-processing*, *DMA*, and *context queue* (Figure 4.3). Accordingly, we partition connection state into module-local state (cf. Table 4.1). The pipeline stages are chosen to maximize data-path parallelism. Pre-processing accesses connection identifiers such as MAC and IP addresses for segment header preparation and filtering. The

post-processing block handles application interface parameters, such as socket buffer addresses and context queues. These parameters are read-only after connection establishment and enable coordination-free scaling. Congestion control statistics are collected by the post-processor, but are only read by forward stages and can be updated out-of-order (updates commute). The protocol stage executes data-path code that must atomically modify protocol state, such as sequence numbers and socket buffer positions. It is the only *pipeline hazard*—it cannot execute in parallel with other stages. The DMA stage is stateless, while context queue stages may be sharded. Both conduct high-latency PCIe transactions and are thus separate stages that execute in parallel and scale independently.

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

4.3.1 Data-path Workflows

We now describe how we parallelize each data-path workflow by decomposing it into these pipeline stages.

4.3.1.1 Host Control (HC)

HC processing is triggered by a PCIe doorbell (DB) sent via memory-mapped IO (MMIO) by the host to the context queue stage. Figure 4.4 shows the HC pipeline for two transmits (the

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

Table 4.1: FlexTOE connection state partitions (total: 108B).

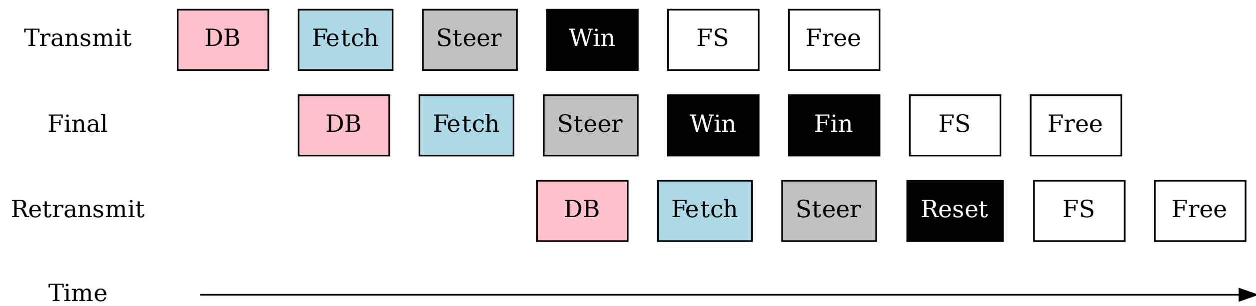

Figure 4.4: FlexTOE HC pipeline: Transmit, FIN, and retransmit.

second transmit closes the connection) triggered by libTOE, and a retransmit triggered by the control-plane. HC requests may be batched.

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

Retransmissions in response to timeouts are triggered by the control-plane and processed the same as other HC events (fast retransmits due to duplicate ACKs are described in §4.3.1.3). The protocol stage resets the transmission state (Reset) to the last ACKed sequence number (go-back-N retransmission).

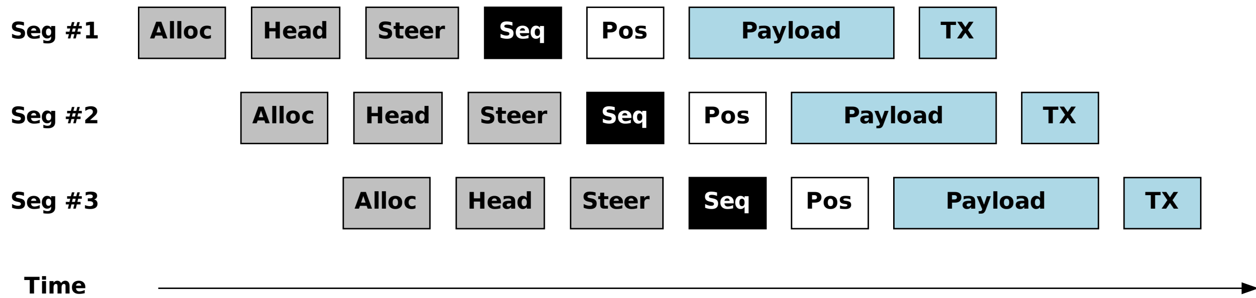

Figure 4.5: FlexTOE TX pipeline sending 3 segments.

4.3.1.2 Transmit (TX)

Transmission is triggered by the flow scheduler (SCH) when a connection can send segments. Figure 4.5 shows the TX pipeline for 3 example segments.

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

4.3.1.3 Receive (RX)

Figure 4.6 shows the RX pipeline for 3 example segments, where segment #3 arrives out of order.

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

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

Out-of-order arrivals (segment #3 in Figure 4.6) need special treatment. Like TAS [138], we track one out-of-order interval in the receive window, allowing the protocol stage to perform reassembly directly within the host socket receive buffer. We merge out-of-order segments within the interval in the host receive buffer. Segments outside of the interval are dropped and generate acknowledgments with the expected sequence number to trigger retransmissions at the sender. This design performs well under loss (cf.§4.5.3).

Post-processing. The post-processor prepares an acknowledgment segment (Ack). FlexTOE provides explicit congestion notification (ECN) feedback and accurate timestamps for RTT estimation (Stamp) in acknowledgments. It also collects congestion control and transmit

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

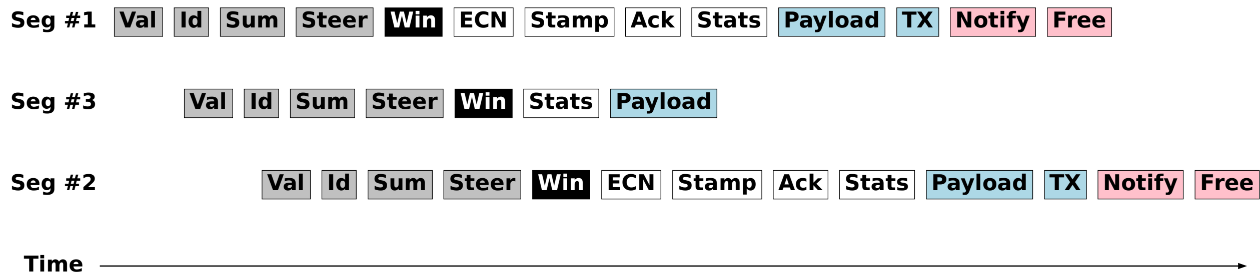

Figure 4.6: FlexTOE RX pipeline receiving 3 segments, 1 out of order.

window statistics, which it sends to the control-plane and flow scheduler (Stats). Finally, it determines the physical address of the host socket receive buffer, payload offset, and length for the DMA stage. If libTOE is to be notified, the post-processor allocates a context queue descriptor with the appropriate notification.

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

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

4.3.2 Sequencing and Reordering

TCP requires that segments of the same connection are processed in-order for receiver loss detection. However, stages in FlexTOE's data-parallel processing pipeline can have varying processing time and hence may reorder segments. Figure 4.7 shows three examples on a

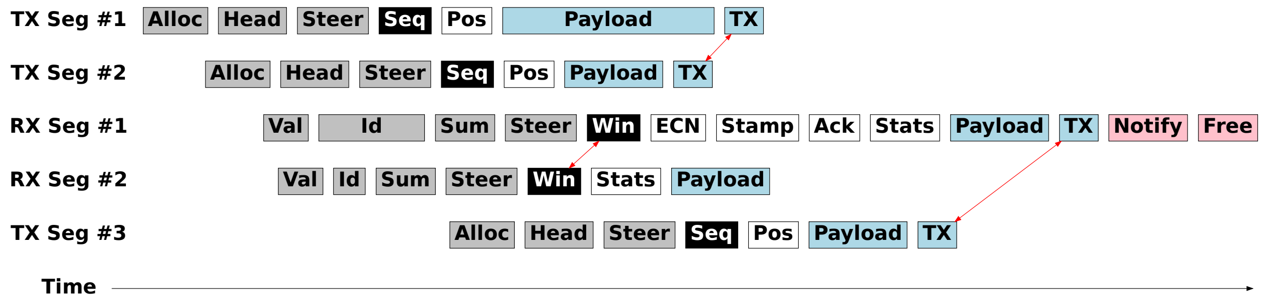

Figure 4.7: Undesirable pipeline reordering (red arrows) in FlexTOE.

bidirectional connection where undesirable segment reordering occurs.

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

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

4.3.3 Flexibility

Data center networks evolve quickly, requiring TCP stacks to be easily modifiable by operators, not just vendors [172, 201, 200]. Many desirable data center features require TOE modification and are adapted frequently by operators. FlexTOE provides flexibility necessary to implement and maintain these features even beyond host stacks such as TAS, by relying on a programmable SmartNIC. To simplify development and modification of the TCP data-path, FlexTOE provides an extensible, data-parallel pipeline of self-contained modules, similar to the Click [189] extensible router.

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

The replication factor of pipeline stages and assignment to FPCs is manual and static in FlexTOE. As long as enough FPCs are available, this approach is acceptable. Operators can determine an appropriate replication factor that yields acceptable TCP processing bandwidth for a pipeline stage via throughput micro-benchmarks at deployment. Stages that modify connection state atomically may be deployed by inserting an appropriate steering stage that steers segments of a connection to the module in the atomic stage, holding their state (cf. protocol processing stage in §4.3).

XDP modules. FlexTOE also supports eXpress Data Path (XDP) modules [116, 118, 119], implemented in eBPF. XDP modules operate on raw packets, modify them if necessary, and

output one of the following result codes: (i) `XDP_PASS`: Forward the packet to the next FlexTOE pipeline stage. (ii) `XDP_DROP`: Drop the packet. (iii) `XDP_TX`: Send the packet out the MAC. (iv) `XDP_REDIRECT`: Redirect the packet to the control-plane.

XDP modules may use BPF maps (arrays, hash tables) to store and modify state atomically [163], which may be modified by the control-plane. For example, a firewall module may store blacklisted IPs in a hash map and the control-plane may add or remove entries dynamically. The module can consult the hash map to determine if a packet is blacklisted and drop it. XDP stages scale like other pipeline stages, by replicating the module. FlexTOE automatically reorders processed segments after a parallel XDP stage (§4.3.2).

Using these APIs, we modified the FlexTOE data-path many times, implementing the features listed in §2.1 (evaluation in §4.5.1). Further, ECN feedback and segment timestamping (cf. §4.3.1.3) are optional TCP features that support our congestion control policies. Operators can remove the associated post-processing modules if they are not needed.

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

4.3.4 Flow Scheduling

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

We implement our flow scheduler based on Carousel [247]. Carousel schedules a large number of flows using a time wheel. Based on the next transmission time, as computed from rate limits and windows, we enqueue flows into corresponding slots in the time wheel. As the time slot deadline passes, the flow scheduler schedules each flow in the slot for transmission (§4.3.1.2). To conserve work, the flow scheduler only adds flows with a non-zero transmit window into the time wheel and bypasses the rate limiter for uncongested flows. These flows are scheduled round-robin.

4.4 Implementation

We first describe FlexTOE implementation on the Netronome Agilio-CX40 SmartNIC (§4.4.1). We also port FlexTOE on x86 and Mellanox Bluefield off-path SmartNIC to demonstrate that its' design principles generalize across platforms (§4.4.2). FlexTOE's design across the different ports is identical. We do not merge or split any of the fine-grained modules or reorganize the pipeline across ports.

4.4.1 Agilio-CX40 Implementation

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

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

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

Context queues. Context queues use shared memory on the host, but communication between SmartNIC and host requires PCIe. We use scalable and efficient PCIe communication techniques [77] that poll on host memory locations when executing in the host and on NIC-internal memory when executing on the NIC. The NIC is notified of new queue entries via MMIO to a NIC doorbell. The context queue manager notifies applications through MSI-X interrupts,

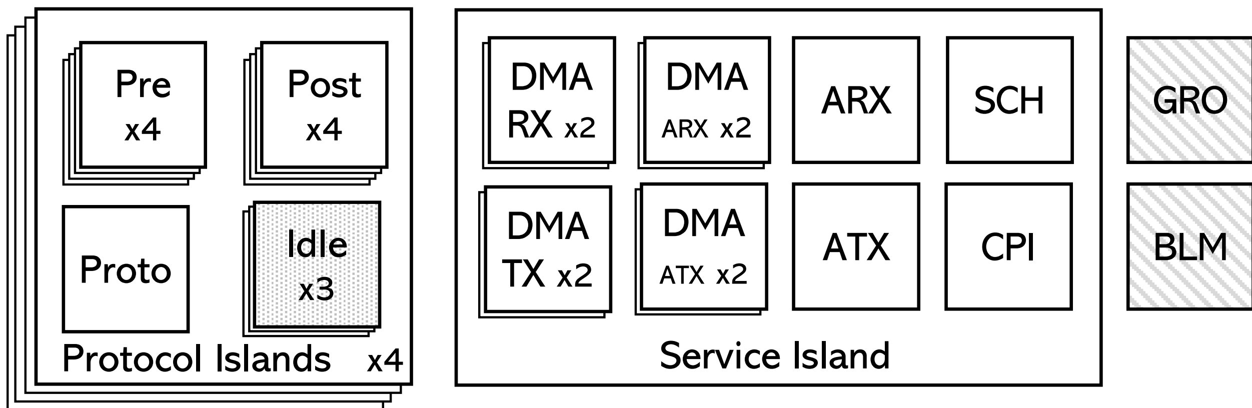

Figure 4.8: FlexTOE data-path to FPC and island assignment on Agilio CX40.

converted by the driver to an `eventfd`, after a queue has been inactive.

4.4.1.1 Near-memory Processing

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

Caching. We use each FPC's CAM to build 16-entry fully-associative local memory caches that evict entries based on LRU. The protocol stage adds a 512-entry direct-mapped second-level cache in CLS. Across four islands, we can accommodate up to 2K flows in this cache. The final level of memory is in EMEM. When an FPC processes a segment, it fetches the relevant

state into its local memory either from CLS or from EMEM, evicting other cache entries as necessary. We allocate connection identifiers in such a way that we minimize collisions on the direct-mapped CLS cache.

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

FPC mapping. FlexTOE's pipeline fully leverages the Agilio CX40 and is extensible to further FPCs, e.g., of the Agilio LX [194]. For island-local interactions among modules, we use CLS ring buffers. CLS supports the fastest intra-island producer-consumer mechanisms. Among islands, we rely on work-queues in IMEM and EMEM.

Figure 4.8 shows how pipeline stages and relevant modules map onto FPCs on the NFP-4000 islands. We use all but one general-purpose islands for the first three stages of the data-path pipeline (*protocol islands*). Each island manages a *flow-group*. While protocol and post-processing FPCs are local to a flow-group, pre-processors handle segments for any flow. We assign 4 FPCs to pre-/post-processing stages in each flow-group. Each island retains 3 unassigned FPCs that can run additional data-path modules (§4.5.1).

On the remaining general-purpose island (called *service island*), we host remaining pipeline stages and adjacent modules, such as context queue FPCs, the flow scheduler (SCH), control plane interface (CPI), and DMA managers. DMA managers are replicated to hide PCIe latencies.

To hide the latency of PCIe transactions when interacting with applications, we further separate the CTXQ module into application RX (ARX) and application TX (ATX) context manager FPCs, for processing the RX and HC workflows, respectively. The number of FPCs assigned to each functionality is determined such that no functionality may become a bottleneck. Sequencing and reordering FPCs (GRO & BLM) are located on a further island with miscellaneous functionality.

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

4.4.2 FlexTOE x86 and BlueField Ports

FlexTOE's decomposition, pipeline parallelism, and per-stage replication all generalize across platforms. Both ports are also almost identical to the Agilio-CX40 implementation and were completed within roughly 2 person-weeks, demonstrating the great development velocity of a software TCP offload engine. We describe the implementation differences of each port to the Agilio-CX40 version in this section.

Hardware cache management. The hardware-managed cache hierarchies of x86 and BlueField obviate the need for software-managed caching that was implemented on Agilio. Instead

of leveraging near-memory processing acceleration of the NFP-4000 (cf.§4.4.1.1), our ports implement multi-core ring buffers, flow lookup and packet sequencers in software. The more powerful x86 and BlueField cores make up for the difference in performance.

Symmetric core mapping. Unlike the NFP-4000, where FPCs are organized into islands, cores on x86 and BlueField have mostly symmetric communication properties, so the assignment of modules to cores is arbitrary and the manual FPC mapping step is omitted. However, we note that core mapping may still be beneficial, for example to leverage shared caches and node locality on multi-socket x86 systems. Each instance of a module runs on its own core. Apart from the six fine-grained pipeline modules: *pre-processing*, *protocol*, *post-processing*, *DMA*, *context queue*, and *SCH* shown in Figure 4.3, the ports utilize an additional *netif* module to interface with DPDK NIC queues to receive and transmit packets. Therefore, FlexTOE-scalar uses 7 cores and the FlexTOE-2× configuration uses 2 additional cores to replicate the pre and post-processing stages for a total of 9 cores.

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

Platform-specific parameters. The replication factor of each pipeline stage is platform dependent. Stage-specific micro-benchmarks on each platform can determine it. Our generalization experiments (§4.5.2) are designed to show that FlexTOE's data-parallelism can improve single connection throughput. Hence, we configure only one instance of the FlexTOE data-path pipeline in these versions (no flow-group islands—we do not process multiple connections in

these experiments). Each port's pipeline uses the same number of stages as the Agilio-CX40 version, but we set different replication factors for the pre and post processing stages on x86 and BlueField (no replication and 2× replication). We do not attempt to find the optimal replication factor for best performance nor compact stages to reduce wasted CPU cycles.

4.5 Evaluation

We answer the following evaluation questions:

- **Flexible offload.** How many CPU cycles does FlexTOE save? Can flexible offload improve throughput, latency, and scalability of data center applications? Can we implement common data center features? (§4.5.1)
- **RPCs.** How does FlexTOE's data-path parallelism enable TCP offload for demanding RPCs? Do these benefits generalize across hardware architectures? Does FlexTOE provide low latency for short RPCs? Does FlexTOE provide high throughput for long RPCs? To how many simultaneous connections can FlexTOE scale? (§4.5.2)
- **Robustness.** How does FlexTOE perform under loss and congestion? Does it provide connection-fairness? (§4.5.3)

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

100 Gbps NICs, whereas the Xeon E3 machines have 82599ES 10 Gbps NICs. The machines are connected to a 100 Gbps Ethernet switch.

Baseline. We compare FlexTOE performance against the Linux TCP stack, Chelsio's kernel-based TOE[3], and the TAS kernel-bypass stack[4]. TAS does not perform well with the Agilio CX40 due to a slow NIC DPDK driver. We run TAS on the Intel XL710 NIC, as in [138], unless mentioned otherwise. We use identical application binaries across all baselines. DCTCP is our default congestion control policy.

4.5.1 Benefit of Flexible Offload

CPU savings. Table 4.2 shows FlexTOE's CPU profile for TCP processing per Memcached request (cf. §2.1). FlexTOE eliminates all host TCP stack overheads. FlexTOE's instruction (and Icache) footprint is at least 2× lower than the other stacks, leading to an execution profile similar to TAS, where 46% of all cycles are spent retiring instructions. In addition, 53% of all cycles can be spent in Memcached—an improvement of 2× versus TAS, the next best solution. The remaining cycles are spent in the POSIX sockets API, which cannot be eliminated with TCP offload.

Application throughput scalability. Offloaded CPU cycles may be used for application work. We quantify these benefits by running a Memcached server, as in §2.1, varying the number of server cores. Figure 4.9 shows that, by saving host CPU cycles (cf. Table 2.1), FlexTOE achieves up to 1.6× TAS, 4.9× Chelsio, and 5.5× Linux throughput. FlexTOE eliminates all host TCP stack overheads. FlexTOE's instruction (and Icache) footprint is at least 2× lower than the other

[3] Chelsio does not support kernel-bypass.

[4] TAS [138] performs better than mTCP [120] on all of our benchmarks. Hence, we omit a comparison to mTCP and AccelTCP [188], which uses mTCP.

Module	**FlexTOE**	
	kCycles	%
NIC driver	0	0
TCP/IP stack	0	0
POSIX sockets	0.74	44
Application	0.89	53
Other	0.04	3
Total	1.67	100
Retiring	0.77	46
Frontend bound	0.34	21
Backend bound	0.46	27
Bad speculation	0.09	6
Instructions (k)	2.93	
IPC	1.75	
Icache (KB)	19.00	

Table 4.2: FlexTOE: CPU profile of TCP processing per Memcached request.

stacks, leading to an execution profile similar to TAS, where 46% of all cycles are spent retiring instructions. In addition, 53% of all cycles can be spent in Memcached—an improvement of 2× versus TAS, the next best solution. The remaining cycles are spent in the POSIX sockets API, which cannot be eliminated with TCP offload. FlexTOE and TAS scale similarly—both use per-core context queues. The Agilio CX becomes a compute-bottleneck at 12 host cores. Linux and Chelsio are slow for this workload, due to system call overheads, and do not scale well due to in-kernel locks.

Low (tail) latency. We repeat a single-threaded version of the same Memcached benchmark for all server-client network stack combinations. Latency distributions are shown in Figure 4.10. We can see that FlexTOE consistently provides the lowest median and tail Memcached operation latency across all stack combinations. Offload provides excellent performance isolation by

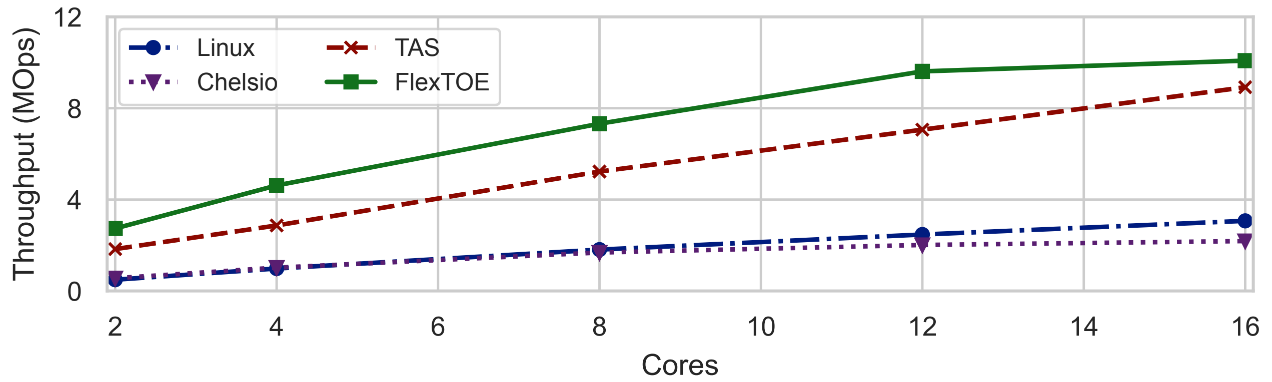

Figure 4.9: FlexTOE Memcached throughput scalability.

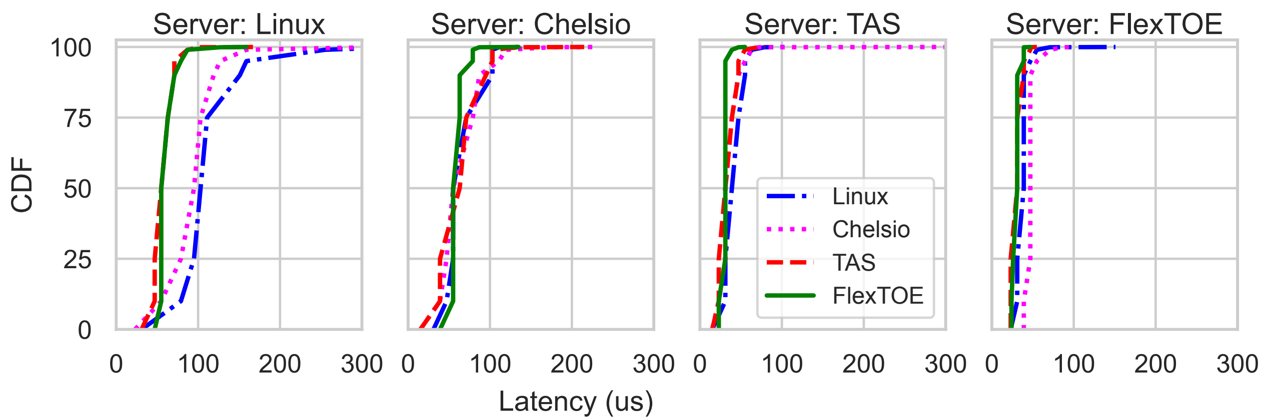

Figure 4.10: Latency of different server-client stack combinations incl. FlexTOE.

physically separating the TCP data-path, even though FlexTOE's pipelining increases minimum latency in some cases (cf. §4.5.2).

Flexibility. Unlike fixed offloads and in-kernel stacks, FlexTOE provides full user-space programmability via a module API, simplifying development. Customizing FlexTOE is simple and does not require a system reboot. For example, we have developed logging, statistics, and profiling capabilities that can be turned on only when necessary. We make use of these capabilities during development and optimization of FlexTOE. We implemented up to 48 different tracepoints (including examples from bpftrace [115]) in the data-path pipeline, tracking transport events such as per-connection drops, out-of-order packets and retransmissions, inter-module queue occupancies, and critical section lengths in the protocol module for various event types. Table 4.3 shows that profiling degrades data-path performance versus the baseline by up to 24% when all 48 trace points are enabled. We also implement `tcpdump`-style traffic logging, including packet filters based on header fields. Logging naturally has high overhead (up to 43% when logging all packets). FlexTOE provides the flexibility to implement these features and to turn them on only when necessary.

Furthermore, new data-plane functionality leveraging the XDP API may be dynamically loaded into FlexTOE as eBPF programs. eBPF programs can be compiled to NFP assembly. This level of dynamic flexibility is hard to achieve with an FPGA as it requires instruction set programmability (overlays [246]). We measure the overhead of FlexTOE XDP support by running a null program that simply passes on every packet without modification. We observe only 4% decline in throughput. Common XDP modules, such as stripping VLAN tags on ingress packets, also have negligible overhead. Finally, connection splicing (Listing 1) achieves a maximum splicing performance of 6.4 million packets per second, enough to saturate the NIC line rate with

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

Table 4.3: FlexTOE performance with flexible extensions.

MTU-sized packets, leveraging only idle FPCs[5].

4.5.2 Remote Procedure Calls (RPCs)

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

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

Figure 4.11 shows the results. For 250 cycles of processing overhead, FlexTOE provides up to 4× better throughput than Linux and 5.3× better throughput than Chelsio when receiving.

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

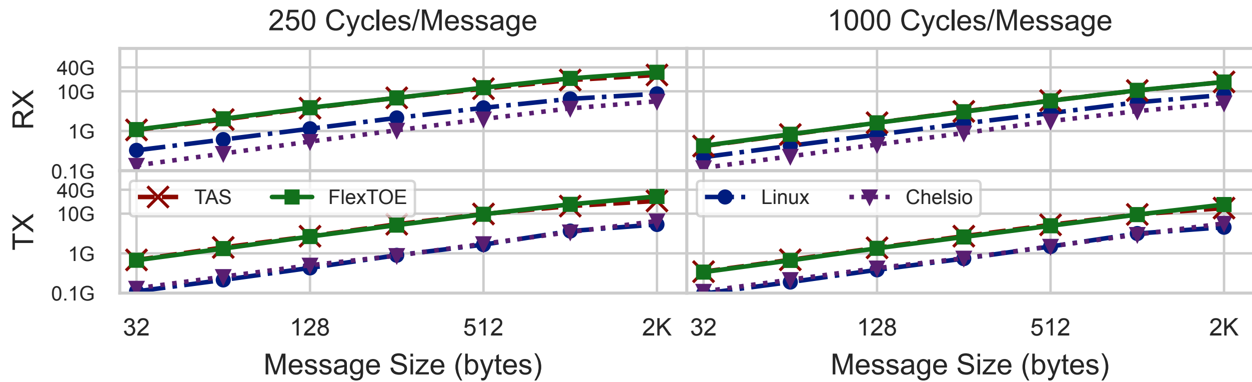

Figure 4.11: FlexTOE RPC throughput for saturated server.

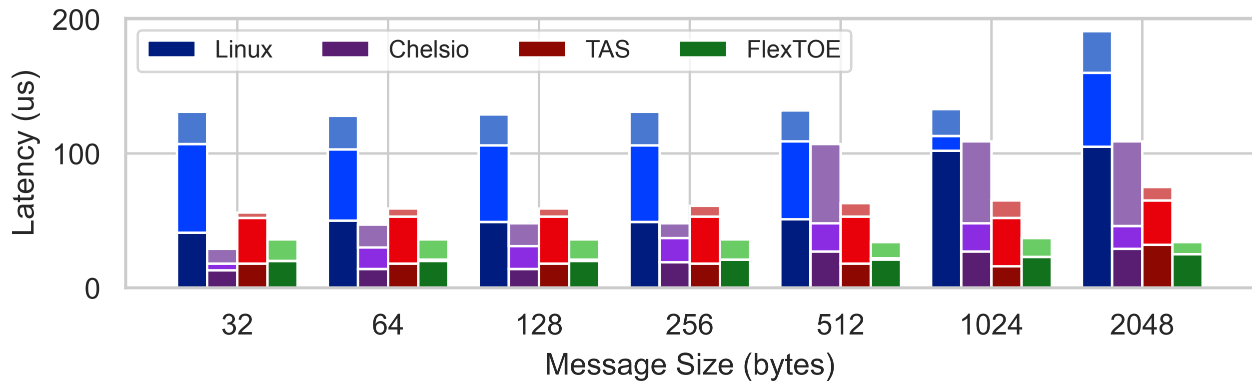

Figure 4.12: FlexTOE median, 99p and 99.99p RPC RTT.

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

We break down this result by studying the performance sensitivity of each TCP stack, varying each RPC parameter within its sensitive dynamic range. For these benchmarks, we evaluate the

raw performance of the stacks, without application processing delays.

RPC latency. A client establishes a single connection to the server and measures single RPC RTT. Figure 4.12 shows the median and tail RTT for various small message sizes (stacked bars). The inefficiency of in-kernel networking is reflected in the median latency of Linux, which is at least 5× worse compared to other stacks. For message sizes < 256 B, FlexTOE's median latency (20 us) is 1.4× Chelsio's median latency (14 us) and 1.25× TAS's median latency (16 us). FlexTOE's data-path pipeline across many wimpy FPCs increases median latency for single RPCs. However, FlexTOE has an up to 3.2× smaller tail compared to Chelsio and nearly constant per-segment overhead as the RPC size increases. In case of a 2 KB RPC (larger than the TCP maximum segment size), FlexTOE's latency distribution remains nearly unchanged. FlexTOE's fine-grain parallelism is able to hide the processing overhead of multiple segments, providing 22% lower median and 50% lower tail latency than TAS.

Per-connection throughput. In this setup, a client transfers a large RPC message to the server. In the first case (Figure 4.13a), the server responds with a 32 B response whereas in the second case (b), the server echoes the message back to the client (TAS performance is unstable with messages > 2 MB in this case—we omit these results). In the short-response case, Chelsio performs 20% better than the other stacks—Chelsio is a 100 Gbps NIC optimized for unidirectional streaming. However, it has 20% lower throughput as compared to FlexTOE in the echo case. Other stacks cannot parallelize per-connection processing, leading to limited throughput[6], while FlexTOE's throughput is limited by its protocol stage. FlexTOE currently acknowledges every incoming packet. For bidirectional flows, this quadruples the number of packets processed per second. Implementing delayed ACKs would improve FlexTOE's performance further for large

[6] With multiple unidirectional flows, all stacks achieve line rate (Figure 4.16b).

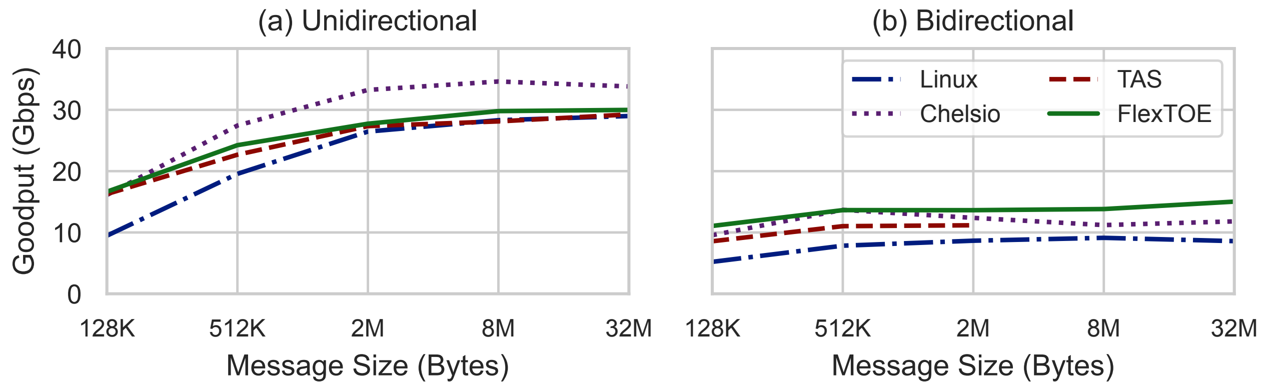

Figure 4.13: FlexTOE large RPC throughput with varying RPC size.

flows.

Connection scalability. We establish an increasing number of RPC client connections from all 5 client machines to a multi-threaded echo server. To stress TCP processing, each connection leaves a single 64 B RPC in-flight. Figure 4.14 shows the throughput as we vary the number of connections. This workload is very challenging for FlexTOE as it exhausts fast memory and prevents per-connection batching, causing a cache miss at every pipeline stage for every segment. Up to 2K connections, FlexTOE shows a throughput of 3.3× Linux. TAS performs 1.5× better than FlexTOE for this workload. FlexTOE is compute-bottlenecked[7] at the protocol stage, which uses 8 FPCs in this benchmark. Agilio CX caches 2K connections in CLS memory. Beyond this, the protocol stage must move state among local memory, CLS, and EMEM. EMEM's SRAM cache is increasingly strained as the number of connections increases. FlexTOE's throughput declines by 24% as we hit 8k connections and plateaus beyond that[8]. TAS's fast-path exhibits better connection scalability, as it has access to the larger host CPU cache, while Linux's throughput

[7] We expect that running FlexTOE on the Agilio LX with 1.2 GHz FPCs—1.5× faster than Agilio CX—would boost the peak throughput to match TAS performance. Agilio LX also doubles the number of FPCs and islands. It would allow us to exploit more parallelism and cache more connections.

[8] While we evaluate up to 16K connections, FlexTOE can leverage the 2 GB on-board DRAM to scale to 1M+ connections.

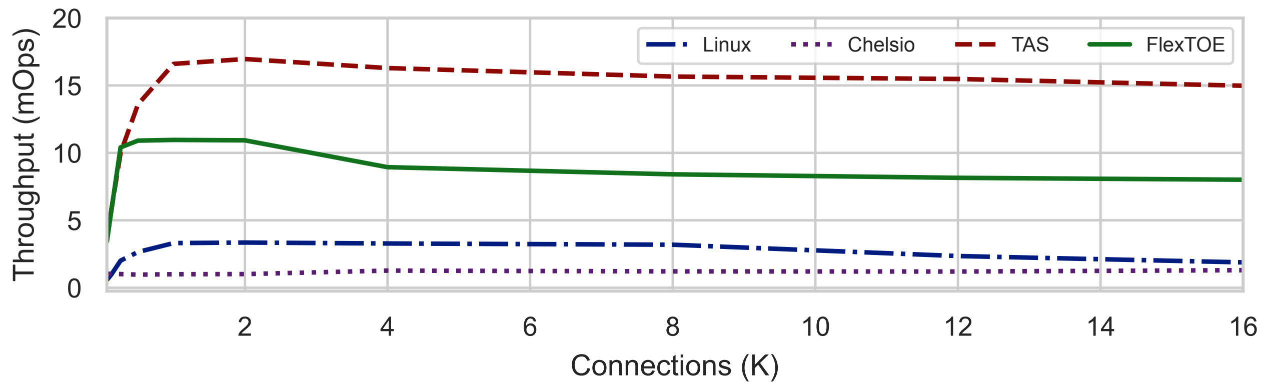

Figure 4.14: FlexTOE connection scalability benchmark.

declines significantly. Chelsio has poor performance for this workload, as `epoll()` overhead dominates.

Benefit of data-path parallelism. To break down the impact of FlexTOE's data-parallel design on RPC performance, we repeat the echo benchmark with 64 connections, with each connection leaving a single 2 KB RPC in-flight (to be able to evaluate both intra and inter connection parallelism). Table 4.4 shows the performance impact as we progressively add data-path parallelism. Our baseline runs the entire TCP processing to completion on the SmartNIC before processing the next segment. Pipelining improves performance by 46× over the baseline. As we enable 8 threads on the FPCs (2.25× gain), we hide the latency of memory operations and improve FPC utilization. Next, we replicate the pre-processing and post-processing stages, leveraging sequencing and reordering for correctness, to extract 1.35× improvement and finally, with four flow-group islands, we see a further 2× improvement. We can see that each level of data-path parallelism is necessary, improving RPC throughput and latency by up to 286×.

Do these benefits generalize?. We investigate whether data-path parallelism provides benefits across platforms. In particular, we investigate single connection throughput of pipelined RPCs

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

Table 4.4: FlexTOE data-path parallelism breakdown.

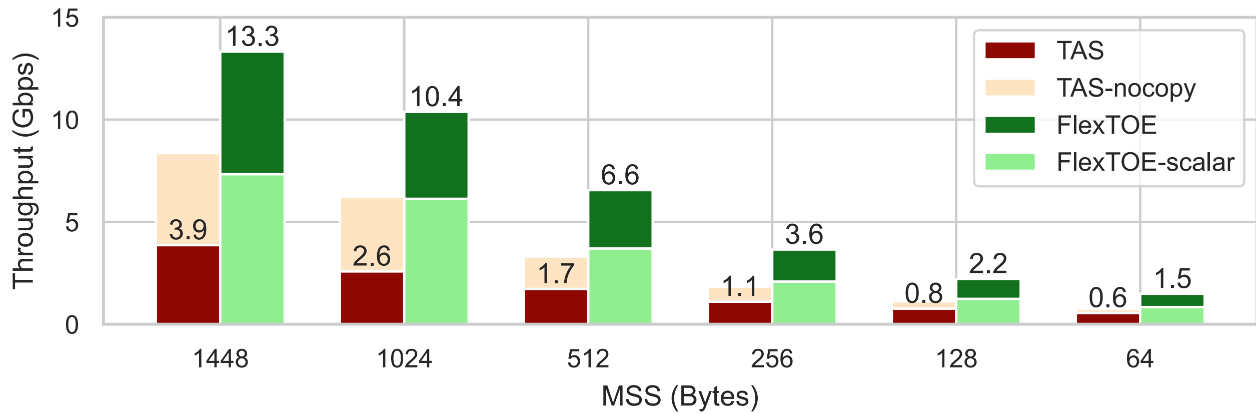

Figure 4.15: FlexTOE benefits on BlueField SmartNIC.

across a range of maximum segment sizes (MSS) on a Mellanox BlueField [175] MBF1M332A-ASCAT 25 Gbps SmartNIC and on a 32-core AMD 7452 @ 2.35 GHz host with 128 GB RAM, 148 MB aggregate cache, and a conventional 100 Gbps ConnectX-5 NIC. We use a single-threaded RPC sink application, running on the same platform[9]. We compare TAS's core-per-connection processing to FlexTOE's data-parallelism. We replicate each of FlexTOE's pre and post processing stages 2×, resulting in 9 FlexTOE cores. Further gains may be achievable by more replication. To break down FlexTOE's benefits, we also compare to a FlexTOE pipeline without replicated stages (FlexTOE-scalar), using 7 cores.

Figure 4.15 shows BlueField results. FlexTOE outperforms TAS by up to 4× on BlueField (and 2.4× on x86). Depending on RPC size, FlexTOE accelerates different stages of the TCP data-path. For large RPCs, FlexTOE accelerates data copy to socket payload buffers. To show this, we eliminate the step in TAS (TAS-nocopy), allowing TAS to perform at 0.5× FlexTOE on BlueField (and identical to FlexTOE on x86). For smaller RPCs, TAS-nocopy benefits diminish and FlexTOE supports processing higher packet rates. FlexTOE-scalar achieves only up to 2.3× speedup over TAS on BlueField (and 1.47× on x86), showing that only part of the benefit comes from pipelining. Finally, FlexTOE speedup is greater on the wimpier BlueField, resembling our target architecture (§4.1.1), than on x86. To save powerful x86 cores, some stages may be collapsed, even dynamically (cf. Snap [172]), at little performance cost.

4.5.3 Robustness

Packet loss. We artificially induce packet losses in the network by randomly dropping packets at the switch with a fixed probability. We measure the throughput between two machines for

[9] BlueField is an off-path SmartNIC that is not optimized for packet processing offload to host-side applications (§4.1.1).

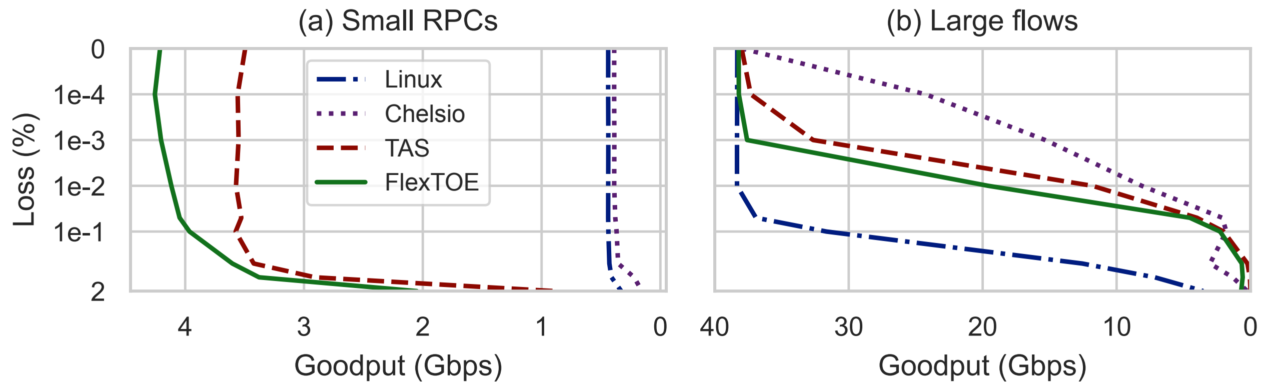

Figure 4.16: FlexTOE throughput, varying packet loss rate.

100 flows running 64 B echo-benchmark as we vary the loss probability, shown in Figure 4.16a. We configure the clients to pipeline up to 8 requests on each connection to trigger out-of-order processing when packets are lost. FlexTOE's throughput at 2% losses is at least twice as good as TAS and an order of magnitude better than the other stacks for this case. We repeat the unidirectional large RPC benchmark with 8 connections and measure the throughput as we increase the packet loss rate. For this case (b), Chelsio has a very steep decline in throughput even with 10^{-4}% loss probability. Linux is able to withstand higher loss rates as it implements more sophisticated reassembly and recovery algorithms, including selective acknowledgments—FlexTOE and TAS implement single out-of-order interval tracking on the receiver-side and go-back-n recovery on the sender. FlexTOE's behavior under loss is still better than TAS. FlexTOE processes acknowledgments on the NIC, triggering retransmissions sooner, and its predictable latency, even under load, helps FlexTOE recover faster from packet loss. We note that RDMA tolerates up to 0.1% losses [185], while eRPC falters at 0.01% loss rate [131]. Unlike FlexTOE, RDMA discards all out-of-order packets on the receiver side [185]. TAS [138] provides further evaluation of the benefits of receiver out-of-order interval tracking.

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

Table 4.5: FlexTOE congestion control under incast.

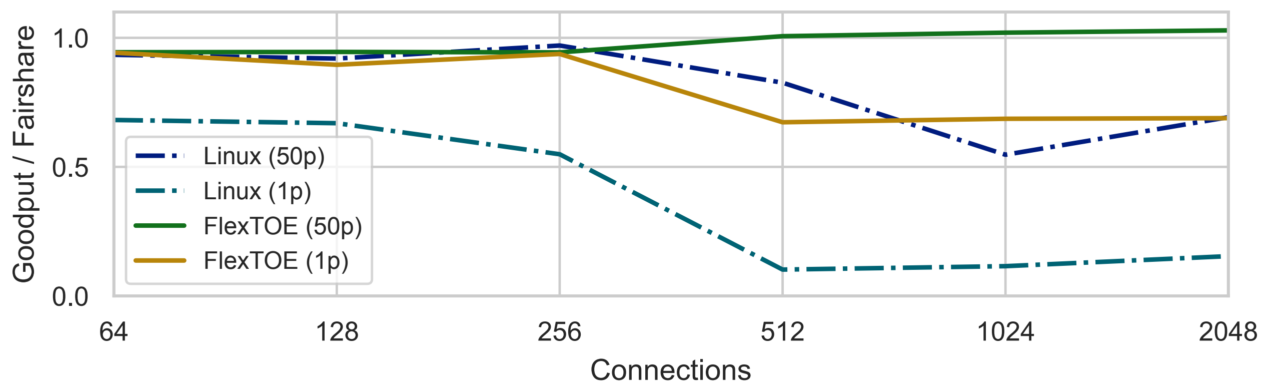

Figure 4.17: FlexTOE connection throughput distribution at line rate.

Fairness. To show scalability of FlexTOE's SCH (§4.3.4), we measure the distribution of connection throughputs of bulk flows between two nodes at line rate for 60 seconds. Figure 4.17 shows the median and 1st percentile throughput of FlexTOE and Linux as we vary the number of connections. For FlexTOE, the median closely tracks the fair share throughput and the tail is 0.67× of the median. Linux's fairness is significantly affected beyond 256 connections. Jain's fairness index (JFI) drops to 0.36 at 2K connections for Linux, while FlexTOE achieves 0.98. Above 1K connections, Linux' median throughput is worse than FlexTOE's 1st percentile.

Incast. We simulate incast by enabling traffic shaping on the switch to restrict port bandwidth to various incast degrees and we configure WRED to perform tail drops when the switch buffer is exhausted. In this experiment, the client transfers 64 KB RPCs and the server responds with

a 32 B response on each connection. As shown in Table 4.5, control-plane-driven congestion control in FlexTOE is able to achieve the shaped line rate, maintain low tail latency, and ensure fairness among flows under congestion. Disabling it causes excessive drops, inflating tail latency by 18.8× and skewing fairness by 2×.

4.6 Conclusion

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

Listing 1 Connection splicing with XDP in FlexTOE.

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

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

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

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

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

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

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

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

Chapter 5

Laminar: A Match-Action TCP Stack for the Terabit Era

In search of a TCP stack that rivals ASICs in efficiency, but also remains flexible via software programmability, we investigate the reconfigurable match-action table (RMT) architecture [38]. Deployed in high-speed programmable network switches [59] and SmartNICs [222], RMT has emerged as a promising architecture for flexible packet processing [272, 256, 257, 92, 154, 149, 281, 141, 140]. Its sequential match-action stages enable deterministic, energy-efficient line-rate packet processing, making it well-suited for many networking tasks. However, implementing TCP in RMT is challenging: TCP requires complex state updates with intricate dependencies, consistent state for bidirectional flow coordination, and buffering for segments awaiting reassembly and acknowledgement. All are ill-suited to RMT's unidirectional pipeline with limited local state and strict timing that complicates dependencies, consistency, and buffering.

We present Laminar, the first TCP stack tailored to the RMT architecture, enabling practical TCP processing at terabit speeds without compromising throughput, latency, energy efficiency, or flexibility. Laminar re-imagines TCP as a pipeline of lightweight match-action operations, enabling one-shot, line-rate processing without buffering. Achieving pipeline-parallel TCP state management requires resolving dependencies between state variables. To address RMT's key constraints, Laminar introduces: (1) optimistic concurrency with deferred validation, performing fast-path speculative updates validated downstream, to handle dependencies on state further down the line; (2) pseudo segment injection to effect updates to state variables in prior stages, resolving cyclic state dependencies without side-effects or stalling the pipeline; and (3) a bump-in-the-wire design that eliminates recirculation and performs stateful TCP segment processing in a single pipeline pass. Together, these principles enable terabit-scale performance, while maintaining TCP semantics and POSIX socket compatibility.

We make the following contributions:

- We present the design of Laminar, describing how to overcome the challenges of mapping stateful protocols to programmable match-action hardware with limited resources and timing. By addressing these challenges, Laminar informs efficient design of next-generation TCP stacks, including on SmartNICs, FPGAs, and hybrid systems that integrate programmable and fixed-function components.

- We prototype Laminar on an Intel Tofino2 switch [59, 191], demonstrating for the first time that RMT can support a TCP stack with ASIC-like performance and energy efficiency while retaining flexibility. To show that Laminar generalizes across platforms, we port it to an FPGA-based SmartNIC, where it supports 3× higher packet rates than ToNIC [23] and orders of magnitude higher than Beehive [156] under identical timing constraints,

while consuming significantly fewer hardware resources. We will open-source Laminar.

- We compare Laminar to Linux, the TAS [138] kernel-bypass TCP stack, and RDMA (§5.5). Laminar surpasses TAS's peak throughput using 16 fewer CPU cores, matching RDMA performance and efficiency. Laminar achieves 25Mpps per-core for streaming workloads, enough to saturate 1.6Tbps with 8K MTU. At scale, Laminar drives nearly 1Bpps while keeping RPC tail latency near 20μs. Laminar is POSIX sockets compatible, interoperates well with other network stacks, and benefits real-world applications: a key-value store on Laminar achieves 2× throughput-per-watt while maintaining a 99.99p tail latency lower than TAS's best case tail latency, while SPDK's NVMe-oTCP with Laminar delivers RDMA-level performance and CPU efficiency. Laminar is also resilient to packet loss and congestion, achieving up to 2× higher throughput than TAS under packet loss. We showcase Laminar's flexibility by accelerating a shared log application using an in-network sequencer similar to NoPaxos [153], and by adding transport extensions, such as Timely congestion control and delayed ACKs.

The rest of this chapter is organized as follows. We begin with background on the match-action architecture and challenges for TCP offload (§5.1). Next, we present the design (§5.2) and implementation (§5.4) of Laminar. Finally, we evaluate Laminar in depth (§5.5).

5.1 Background

We introduce the match-action architecture, the foundation of modern programmable switches and SmartNICs, and explain how its design principles enable high-speed packet processing (§5.1.1). Then, we discuss the unique challenges that arise when mapping TCP's stateful transport logic onto this architecture, ranging from pipeline constraints to consistency and

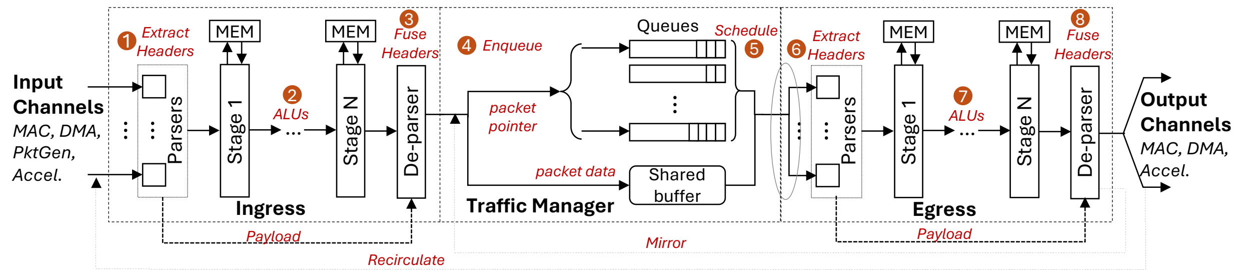

Figure 5.1: Typical RMT pipeline in switches and SmartNICs.

resource limitations (§5.1.2).

5.1.1 RMT Architecture

The reconfigurable match-action table (RMT) architecture, popular for high-performance programmable switches, is optimized for line-rate packet processing. It exploits the parallelism inherent in network packet processing tasks, enabling simultaneous packet header operations, and a pipeline to chain dependent operations via multiple stages. This design sustains over 6 Billion packets per second with $\sim$ 400 ns of latency at a power consumption comparable to ASICs [38, 13]. In this section, we describe a typical RMT architecture, found in switches and SmartNICs (Figure 5.1).

Overview. Packets enter the processing pipeline over one or more MAC or DMA input channels, originating from the network or the host, respectively. The ingress pipeline begins with a programmable parser extracting header fields into a Packet Header Vector (PHV) for use throughout the pipeline ①. Parsed packets are then multiplexed into the ingress pipeline, consisting of multiple sequential, reconfigurable match-action stages ②. Packets pass through the pipeline stages sequentially, executing the same processing steps in a synchronized, lockstep manner. Each stage consists of programmable elements, like ALUs and gateways, which enable

simultaneous updates to several PHV fields using primitive arithmetic operations, conditioned on matching specific PHV fields with match tables in stage-local memory. The updated PHV is passed along to the next stage to carry intermediate results. To maintain state across packets, each stage is also equipped with a Stateful ALU that executes read-modify-write (RMW) operations on stage-local memory. The pipeline determines the output channel, for example, by matching the destination MAC address with a forwarding table. Next, the programmable deparser recombines the modified PHV with the payload ③, and enqueues the packet in the Traffic Manager (TM) into a destination queue for the output channel based on the QoS priority assigned by ingress ④. To handle bursty loads and output channel congestion — such as during incast — the TM temporarily buffers packets. Subsequently, the traffic manager (TM) schedules the packet to the egress pipeline for execution, matching bandwidth constraints ⑤. The egress pipeline, independently programmed, mirrors ingress functionality ⑥-⑧, enables customization of packets prior to transmission, and to maintain state following congestion events. Finally, the recombined packet is passed on to the output channel for transmission on the physical interface.

Processing characteristics. Any valid program compiled for the RMT architecture operates at the maximum packet processing bandwidth of the pipeline, sufficient to saturate the combined line rate of all input channels even with small packets ($\sim$ 128B in typical implementations). The pipeline has a fixed aggregate processing latency (typically a few hundred nanoseconds) for each packet determined at compile-time, based on the number of stages utilized by the program.

Traffic Manager (TM). RMT implementations typically partition the pipeline into ingress and egress, with a traffic manager (TM) in-between (cf. Figure 5.3). The TM may temporarily buffer

packets, such as during traffic bursts, and it schedules packets to the egress pipeline, matching pipeline and channel bandwidth constraints. The TM enables lossless egress operation by responding to backpressure mechanisms, such as Ethernet pause frames and PCIe flow control, temporarily halting packet transmissions on the corresponding output channel. Importantly, the egress pipeline never drops packets, as the TM schedules packets to match the bandwidth of the pipeline and the output channel.

Stateful ALUs. Each stage consists of a programmable Stateful ALU to store and update application-specific state in memory. They are computationally constrained due to the need for each stage to operate within fixed timing constraints. Timing is critical, as the pipeline operates in a synchronized, lockstep manner, and packets requiring access to the same state can arrive back-to-back. Consequently, the Stateful ALUs are stage-local, meaning read and update operations must occur within the same pipeline stage. Memory access is limited to only a few bytes, with restricted comparison or arithmetic operations, with additional constraints on operand types. When multiple operations involve the same state, they must share the limited resources of the Stateful ALU. For example, the Intel Tofino2 architecture has only four 32-bit comparators and four 32-bit primitive arithmetic operators in each Stateful ALU, which may be shared among 4 independent operations [59].

Packet injection mechanisms. RMT supports several useful mechanisms to trigger further processing and to carry state back into the pipeline:

- **Recirculation:** RMT implementations typically incorporate a recirculation channel, enabling multiple passes through the pipeline. However, each pass reduces the effective bandwidth of the RMT pipeline and doubles its latency, making this mechanism infeasible for line-rate processing.

- **Mirroring:** The deparser may generate copies of a PHV and re-inject them at the TM. Each mirrored PHV is scheduled separately by the TM and may be further modified in egress. Mirroring also reduces effective RMT bandwidth, but much less than recirculation, as only the PHV is mirrored.
- **Packet Generator:** A packet generator can emit packets on triggers, commonly on periodic timers.

Control plane. RMT programs typically rely on a control plane for configuration, management, and to handle exceptions and other uncommon processing tasks. The control plane is implemented to execute on a CPU, either on the host or adjacent to the RMT pipeline itself.

5.1.2 Challenges for TCP on RMT

A TCP stack for RMT must achieve fine-grained pipeline parallelization of the transport logic to provide the segment processing rates necessary for line rate performance. In particular, doing so comprises the following challenges:

1. **Unidirectional lock-step pipeline:** The RMT architecture utilizes a lock-step pipeline, where each stage operates independently with limited local memory and timing. As a consequence, TCP state and logic must be partitioned across stages. This introduces *forward* read dependencies, where a current stage depends on state in future pipeline stages, and *circular* write dependencies that require state updates in prior stages.
2. **Limited mirroring:** Mirroring may be employed to loop state back into the pipeline, but must be used sparingly to preserve performance; it can take several hundred nanoseconds to execute and may experience congestion.
3. **Consistency:** Several segments can concurrently process in separate pipeline stages, but all

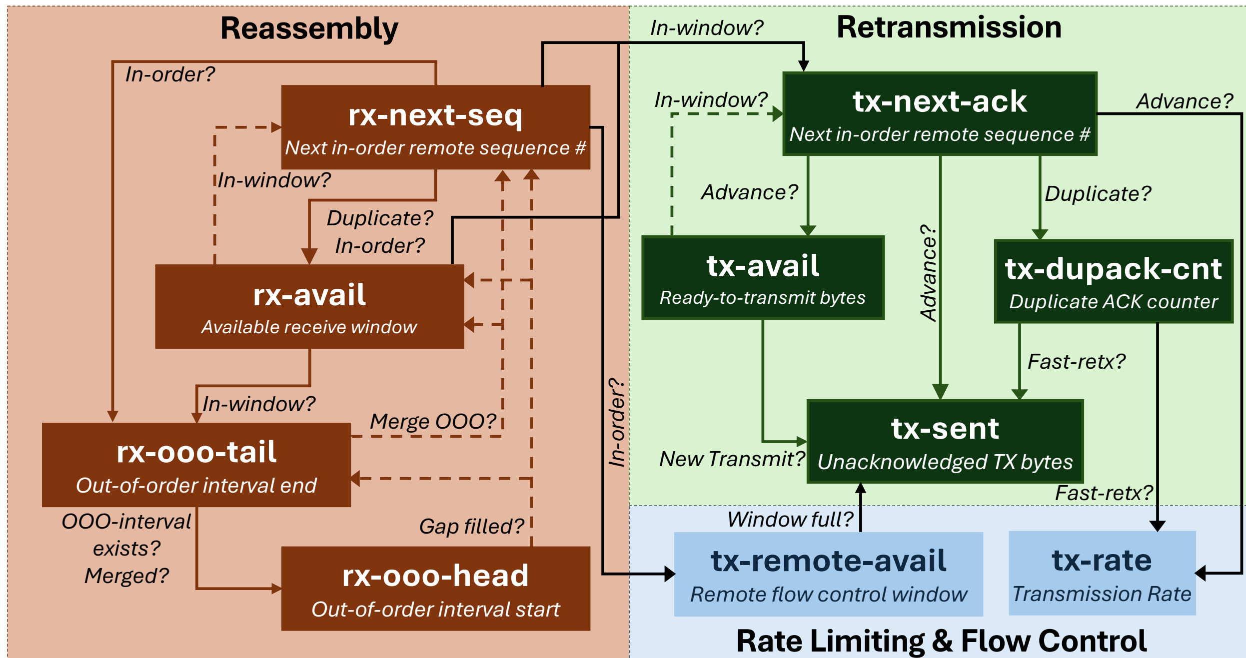

Figure 5.2: TAS TCP connection state and dependencies.

segments must observe a consistent and up-to-date state snapshot, even while segments are looped back to resolve circular dependencies and despite congestion loss.

4. **No buffering:** Limited capacity prevents prolonged segment buffering within the pipeline, particularly with increasing bandwidth-delay products of terabit networks.

To understand the challenges involved, we analyze TAS [138], a state-of-the-art data center kernel-bypass TCP stack. TAS already streamlines the TCP transport logic to minimize overheads, providing an advanced baseline. We particularly focus on TAS' per-connection data-path logic and its state dependencies—stateless and read-only logic is easily parallelized, while off-path logic does not benefit from acceleration.

Figure 5.2 categorizes TAS' per-connection state into three groups (cf. RFC 9293 [74] Section 3.3.1), each corresponding to essential TCP data-path functions: reassembly, retransmission,

and rate limiting and flow control, with arrows depicting dependencies among variables: solid arrows indicate forward dependencies, while dotted arrows indicate circular dependencies. We omit congestion control state, as it is implemented out-of-band in TAS.

TCP is bidirectional protocol, with each segment containing both receive and transmit updates, resulting in intricate dependencies between reassembly and retransmission state. We now describe the three groups in more detail.

Reassembly. TCP is a stream-oriented protocol, but data can arrive in fragments and out-of-order (OOO), requiring the receiver to buffer these segments to reassemble the data stream before passing it to the application. TAS defines the receive window, which represents the receiver's buffer capacity for incoming data, using two variables: `rx-next-seq`, the next expected in-order TCP sequence number, and `rx-avail`, the number of available bytes currently within the receive buffer. TAS tracks a single out-of-order (OOO) interval within the receive window, bounded by `rx-ooo-head` and `rx-ooo-tail` (cf. §5.5.4). Figure 5.2 shows the complex cyclic dependencies within the reassembly state (cf. RFC 9293 [74] Section 3.4). For instance, advancing `rx-next-seq` upon segment arrival involves determining whether the segment fits into the receive window (`rx-avail`), determining if it merges with an existing OOO interval (`rx-ooo-*`). Critically, the subsequent update to `rx-next-seq` itself determines the changes to the window and the OOO interval, thus tightly intertwining all reassembly state.

(Re-)transmission. TCP stacks detect lost segments by tracking acknowledgments and recover segments via retransmission, requiring segments to be buffered until they are acknowledged. The transmit window is defined by `tx-next-ack`, the first unacknowledged TCP sequence number, `tx-avail`, the bytes ready for transmission within the window, and `tx-sent`, transmitted but unacknowledged bytes. Additionally, `tx-dupack-cnt` counts duplicate acknowledgments for

fast retransmission. TAS implements go-back-N retransmission [138], which does not require additional state. Figure 5.2 shows the dependencies among (re-)transmission state. ACKs usually fall before `tx-next-ack` + `tx-sent`. However, retransmissions reset `tx-sent`, allowing valid ACKs to exceed it, but not `tx-avail`. This creates a forward dependency between `tx-next-ack`, `tx-avail`, and `tx-sent` for ACK validation and transmit window advancement (cf. RFC 9293 [74] Section 3.10.7.4).

Rate limiting and Flow control. Flow control prevents a fast sender from overwhelming a slow receiver's capacity to buffer received segments. Congestion control mitigates network congestion by rate limiting senders based on network capacity. Both mechanisms can require senders to buffer ready-to-transmit segments. `tx-remote-avail` tracks the remote receive window, advertised by the remote peer in the TCP segment header, guiding the stack to limit the in-flight bytes (`tx-sent`) accordingly. Updating `tx-remote-avail` solely for in-order segments introduces dependencies on the reassembly state. Likewise, halving `tx-rate` under fast retransmission introduces dependencies on the (re-)transmission state.

5.2 Overview

Laminar is, to our knowledge, the first TCP stack designed for line-rate TCP segment processing. Laminar is general-purpose—it scales to thousands of connections, is robust against packet loss and congestion, and maintains TCP endpoint compatibility. Like many other proposals [259, 138], Laminar accelerates TCP processing for established connections—the common case for high-performance applications.

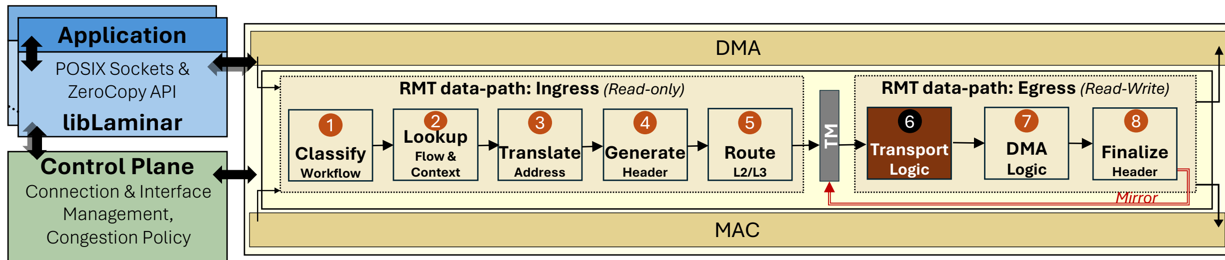

Figure 5.3: Laminar overview. The RMT TCP transport logic in ⑥ is our focus (§5.3).

Laminar offload architecture. To do so, it divides the TCP stack into three key components (Figure 5.3): control plane, application interface, and accelerated RMT data-path (§3.1). The application interface, libTOE, and the RMT data-path communicate using DMA. Each libTOE instance (*context*) operates an independent, flow-controlled DMA channel.

RMT data-path. The RMT data-path executes core TCP transport logic for established connections to ensure reliable and ordered delivery of byte streams directly to/from libTOE. To do so, ingress pipeline stage ① first classifies a segment into one of three workflows, based on its origin and header:

1. **RX** processes segments received from the MAC, performing reassembly and delivering the payload into libTOE's per-connection receive buffers via DMA.
2. **TX** handles DMA transmit notifications from libTOE, preparing segments and enforcing congestion and flow control.
3. **SYNC** periodically updates data-path RX and TX window availability, triggered from multiple sources.

In ②, Laminar determines the connection and context identifiers, used throughout the pipeline. The lookup is based on the segment's 5-tuple on RX, or DMA header fields upon TX and SYNC. In ③, the TCP sequence number is translated to a host virtual address for RX segments, and

vice versa for TX packets (§5.3.3). In ④, Laminar preemptively prepares the segment for its destination, by preparing DMA headers for RX & SYNC, and IP&TCP headers for TX. After this, the pipeline performs L2/L3 forwarding for TX and non-Laminar packets ⑤, and submits the segments to the traffic manager (TM) for egress processing.

The egress pipeline realizes the core TCP transport logic (§5.3), including reassembly (§5.3.1) and (re-)transmission (§5.3.2) ⑥. It updates the DMA state as needed for issuing DMA operations ⑦ (§5.4) and finalizes segment headers before transmission ⑧. For RX, a DMA engine transfers the payload to host memory and notifies the application. If an acknowledgment is needed, the pipeline mirrors the segment to the source channel for processing and transmission (§5.3.1.4).

Leveraging lossless transmission in the egress pipeline, we design the ingress pipeline for read-only operation and maintain mutable state in the egress pipeline. This ensures that Laminar is *drop resistant*. Any packets dropped in the TM are equivalent to network losses and handled TCP's retransmission logic.

5.3 RMT TCP Transport Logic Design

The heart of Laminar is its RMT TCP transport logic. Laminar re-architects TAS' per-connection logic (Figure 5.2) to maximize RMT pipeline parallelism in the common case. To overcome the challenges of §5.1.2, Laminar adopts the following design principles:

1. **Bump-in-the-wire processing:** To preserve RMT pipeline processing bandwidth and to achieve near-hardware latency and throughput, Laminar processes each TCP segment no more than once in the pipeline. Segments are never buffered in the pipeline, making judicious

use of the limited pipeline memory capacity.

2. **Optimistic segment processing:** Laminar optimizes handling forward read dependencies with *optimistic processing*, assuming the *common case*—in-order segment receipt and delivery—and verifying the assumption later, potentially restoring critical state via cyclic updates.
3. **Pseudo segment cyclic update:** To resolve cyclic write dependencies without disrupting the pipeline, Laminar injects *pseudo segments* into the RMT pipeline. These segments are processed like regular TCP segments, but are carefully crafted to cause the desired state updates in earlier stages without side effects. TCP state is always kept consistent, even if such updates may, in exceptional cases, briefly reduce pipeline performance.

We now detail how Laminar realizes TCP transport logic in RMT. First, we describe segment reassembly (§5.3.1), including how we handle in-order, out-of-window, and out-of-order segments efficiently. Next, we present the mechanisms for (re-)transmission (§5.3.2), emphasizing congestion control and acknowledgment processing. Finally, we illustrate how Laminar translates between TCP sequence numbers and host memory addresses (§5.3.3), efficiently supporting buffer management.

5.3.1 Segment Reassembly

On RX, segment reassembly involves incorporating any relevant part of a received segment within the receive window. It requires several computational steps, including identifying which bytes of the segment are within window, trimming out-of-window bytes, advancing the window, notifying libTOE, and acknowledging successfully received bytes to the sender. For robustness, Laminar tracks and buffers segments of up to one out-of-order (OOO) interval. This process introduces the complex state dependencies shown in Figure 5.2 that existing TCP stacks process atomically for each connection. We now describe how Laminar realizes reassembly in sequential

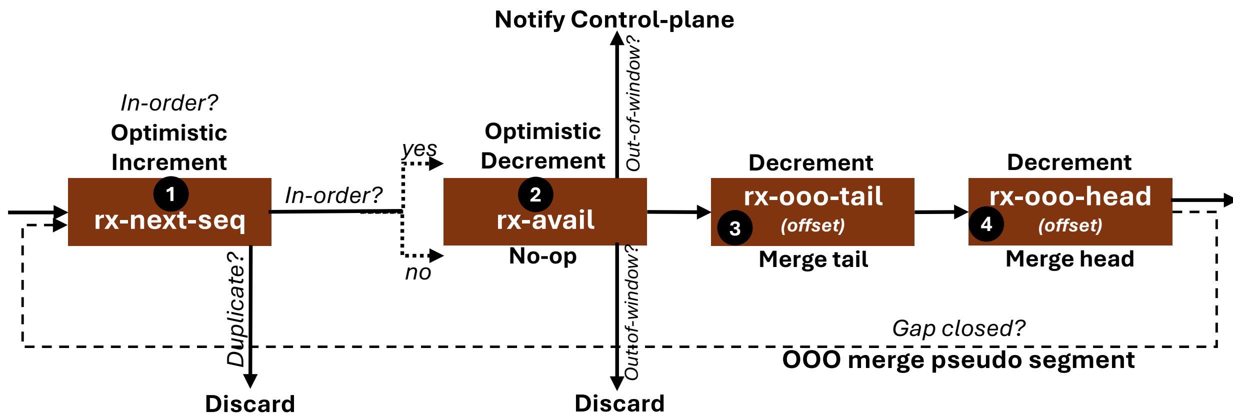

Figure 5.4: RMT TCP segment reassembly paths.

pipeline stages, as illustrated in Figure 5.4, starting with the common case: in-order segment reception.

5.3.1.1 In-order

Laminar initially assumes segments arrive in order. It consequently updates `rx-next-seq` optimistically for the *in-order* portion of a segment ①, deferring the *out-of-window* (OOW) check. Fully duplicate segments, where the entire segment's data sequences before `rx-next-seq`, are dropped, while partially duplicate segments are trimmed. A snapshot of `rx-next-seq` is forwarded to subsequent stages along with the (trimmed) segment. Subsequent stages handle in-order and out-of-order segments differently. We follow the in-order path (top in Figure 5.4). In ②, `rx-avail` is optimistically decremented based on the trimmed in-order segment. This update must happen, even for segments that are fully or partially out-of-window, to maintain consistency with the already updated `rx-next-seq`. ② now conducts the OOW check. If the segment is OOW, Laminar processes it in the control plane (§5.3.1.2). If an OOO interval is tracked, then ③ and ④ decrement `rx-ooo-head` and `rx-ooo-tail`, respectively, as they are represented as offsets from `rx-next-seq`. This design guarantees that all segments operate on

consistent state.

5.3.1.2 Out-of-window

A challenging scenario arises for segments that are *in-order* and (partially) *out-of-window*. For such segments, we have already advanced the receive window. In this situation, Laminar must (a) roll back the state, and (b) provide consistency for any intervening segments.

Well-behaved TCP stacks will not exceed the advertised flow control window, making this scenario rare. Laminar leverages the control plane to resolve it. When the exceptional *out-of-window* scenario is first detected, the segment is dropped and the control plane is notified to reset the TCP state to the value prior to the dropped segment. Meanwhile, any intervening segments will present as *out-of-window* via a zero `rx-avail` and are also dropped, generating acknowledgments advertising a window size of 0 to pause further transmissions.

5.3.1.3 Out-of-order (OOO)

The OOO path (bottom in Figure 5.4) concerns only stages ③ and ④—the other stages do nothing. Leveraging our *bump-in-the-wire* design principle, OOO segments are directly placed at their position in the per-connection receive payload buffer. If no OOO interval is currently being tracked, it is set to the segment's boundaries. Otherwise, the segment merges with the interval's boundaries, extending `rx-ooo-tail` ③ and `rx-ooo-head` ④, respectively, if they overlap with the segment's boundaries. Non-overlapping OOO segments are dropped.

5.3.1.4 Acknowledgment

For each received TCP segment containing payload, Laminar generates a pseudo segment carrying snapshots of the reassembly state (⑥ in Figure 5.3). This pseudo segment is mirrored

and then turned into a TCP acknowledgment (⑧ in Figure 5.3). If the pipeline drops the pseudo segment, the effect is identical to network packet loss of a TCP acknowledgment.

5.3.1.5 OOO interval merge

A cyclic dependency arises when a segment closes the receive window gap to a tracked OOO interval, as it necessitates advancing `rx-next-seq` to the end of the OOO interval and decrementing `rx-avail` by the same amount. To resolve the dependency, Laminar adopts a two-step merge process: after an in-order segment closes the gap (`rx-ooo-head` becomes 0) in ④, Laminar injects a *pseudo segment* into the pipeline (combined with any acknowledgment pseudo segment) that has no payload but covers the entire range between `rx-next-seq` and `rx-ooo-tail`. Pseudo segments allow in-order segment reassembly to proceed normally, even when the OOO interval is temporarily unmerged. If pseudo segments are dropped, the OOO interval will stay unmerged, causing further incoming segments to generate further pseudo segments. Until the merge completes, Laminar's ability to handle out-of-order packets is diminished due to the inability to establish a new OOO interval. However, this scenario is exceedingly rare due to the RMT pipeline's high segment processing bandwidth.

Example. Table 5.1 illustrates this process, detailing reassembly state updates for segments #1 to #7. Segment #1 starts an OOO-interval. Segment #2 closes the gap between the OOO-interval and `rx-next-seq`, triggering the packet Mirror #2, which carries state snapshots for the second merge step. Meanwhile, *in-order* segments #3 and #4 merge at `rx-ooo-tail` and `rx-next-seq`, respectively, each generating mirrored packets. Mirror #2 is partially *duplicate* and is trimmed before partially merging the OOO-interval with the stream. Mirror #3 is lost due to internal congestion, yet state consistency is maintained even for *in-order* segments #5 and #6 reassembly to be processed. Out-of-order segment #7 is dropped as Laminar cannot

	Segment seq#	plen	Stage #1 rx-next-seq	Stage #2 rx-avail	Stage #3 rx-ooo-tail	Stage #4 rx-ooo-head
Initial state			1000	10000	0	0
#1	1500	500	1000	10000	1000	500
#2	1000	500	1500	9500	500	0
#3	2000	500	1500	9500	1000	0
#4	1500	100	1600	9400	900	0
P#2	1500	500	2000	9000	500	0
P#3	1500	1000				
#5	2500	100	2000	9000	600	0
#6	2000	100	2100	8900	500	0
#7	3000	100	2100	8900	500	0
P#4	1600	900	2500	8500	100	0
P#5	2000	600	2600	8400	0	0
P#6	2100	500	2600	8400	0	0

Table 5.1: Example: Laminar TCP out-of-order (OOO) merge using pseudo segments (shown as P#x).

start a new OOO-interval. Mirrored packets #4 and #5 fully unify the receive window, resetting the OOO-interval. Mirror #6 is fully *duplicate* and is discarded.

5.3.1.6 Notification

Each *in-order* segment that advances the receive window triggers a notification to libTOE, indicating the payload buffer offset up to which data is ready for application consumption. When a segment closes the gap to an OOO interval, libTOE is notified ahead of the pseudo segment merge.

5.3.1.7 Receive window replenishment

As the reassembled payload fills the receive buffer, it must be replenished periodically. libTOE triggers the SYNC workflow when the application has consumed more than a predefined portion

of the buffer (default: $\frac{1}{4}$ of its size). SYNC increments *rx-avail* by the amount specified by libTOE.

5.3.2 (Re-)transmission

(Re-)transmission processing is considerably simpler and without cyclic dependencies due to go-back-N retransmission. After reassembly, SYNC (§5.3.2.1), TX (§5.3.2.2), and RX (§5.3.2.3) all traverse these paths as TCP is bidirectional due to acknowledgments being piggybacked on received segments.

5.3.2.1 Congestion Control (SYNC)

To remain robust to packet loss and congestion, Laminar supports DCTCP [16] as the default congestion control policy.

Like other TCP stacks [138, 259, 190, 268], Laminar locates the congestion control policy in the control plane, which simplifies implementation, making it easy to add new policies. The data-path gathers per-connection metrics like acknowledged bytes, ECN-marked bytes, duplicate ACKs, and round-trip times (RTTs). The control plane periodically processes these metrics to compute a new transmission rate and updates the data-path tables. The data-path grants transmission credits based on the rate to libTOE via periodic SYNCs, updating `tx-credits` and potentially `tx-next-ack`. The data-path leverages the packet generator to generate these SYNCs directly within the RMT pipeline, without further control-plane involvement. Upon SYNC, libTOE can reclaim buffer space for acknowledged data and initiate new transmissions. SYNC may also be initiated upon a retransmission timeout. In this case, SYNC signals libTOE to reset its transmission head and initiate retransmissions. To conserve pipeline bandwidth, Laminar pauses SYNC workflows for inactive connections (no data transmission for *N* RTTs).

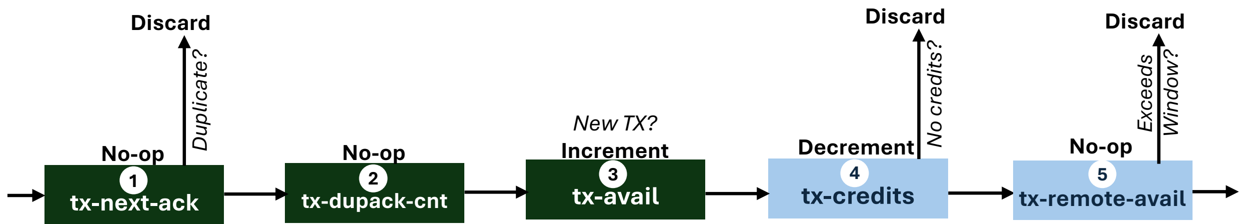

Figure 5.5: RMT TCP processing for TX workflow.

For uncongested flows with short messages that underutilize credits, the control plane reduces SYNC frequency. libTOE may also self-pace SYNCs for such flows, triggering them only as the transmit window nears exhaustion.

5.3.2.2 Transmission (TX)

When sufficient credits are available, libTOE *pushes* any new data into the data-path (Figure 5.5). This approach minimizes latency and data-path complexity, leveraging Laminar's *bump-in-the-wire* design principle. Laminar determines TCP sequence numbers based on the DMA offset within the host transmit buffer. Any segments with sequence numbers below `tx-next-ack` (e.g., due to stale retransmissions that have since been ACK'ed to the data-path) are dropped ①. If the segment extends beyond the current transmit window, `tx-avail` is updated to reflect it ③. Transmission is credit limited (cf. §5.3.2.1) and flow controlled in ④—the segment size is deducted from `tx-credits` and the segment is dropped when `tx-credits` is zero or when the number of unacknowledged bytes exceeds the flow control window `tx-remote-avail` ⑤.

5.3.2.3 Acknowledgment Processing (RX)

For any received ACK, Laminar optimistically updates `tx-next-ack` in ①, discarding any duplicates (Figure 5.6). Similar to reassembly, Laminar is unable to check whether an acknowledgment is *out-of-window* at this stage. In ②, `tx-dupack-cnt` increments for duplicate ACKs and

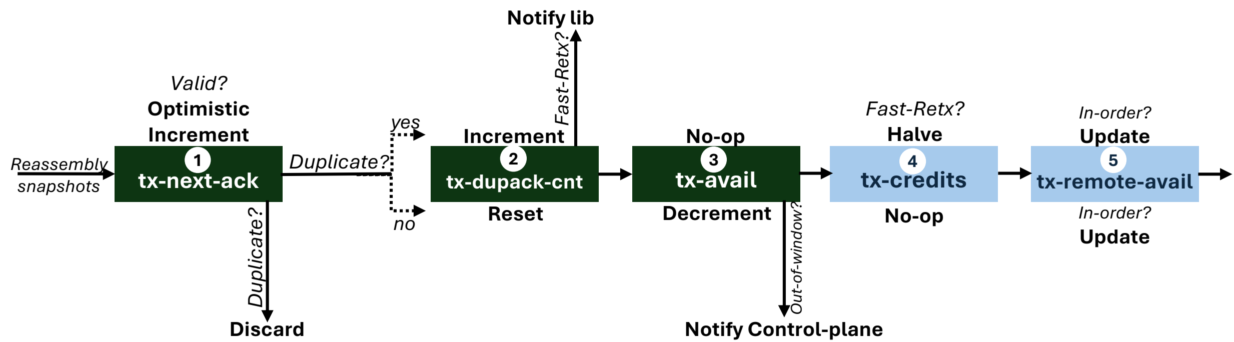

Figure 5.6: RMT TCP processing path for RX workflow.

resets otherwise. Upon receiving 3 duplicate ACKs, Laminar triggers fast retransmission and notifies libTOE of the latest `tx-next-ack` via DMA. The *out-of-window* check occurs in stage ③ when updating `tx-avail`. A well-behaved TCP stack should never send acknowledgments for untransmitted data. If it does, Laminar notifies the control plane to reset the connection. Upon fast retransmission, `tx-credits` is halved in ④, similar to TCP Reno. ACKs that are in-order as determined by the reassembly state snapshots (§5.3.1), update `tx-remote-avail` — the remote peer's receive window status in ⑤.

5.3.3 Sequence-Address Translation

Laminar's per-connection payload buffers adopt double buffering, where the entire buffer is mapped twice consecutively in libTOE virtual address space (Figure 5.7). This design efficiently handles wrap around at the buffer boundary, eliminating the need to split a DMA operation into two separate transfers. The pipeline translates addresses by adding the connection's payload buffer base address to an offset derived from the segment's TCP sequence number and vice versa.

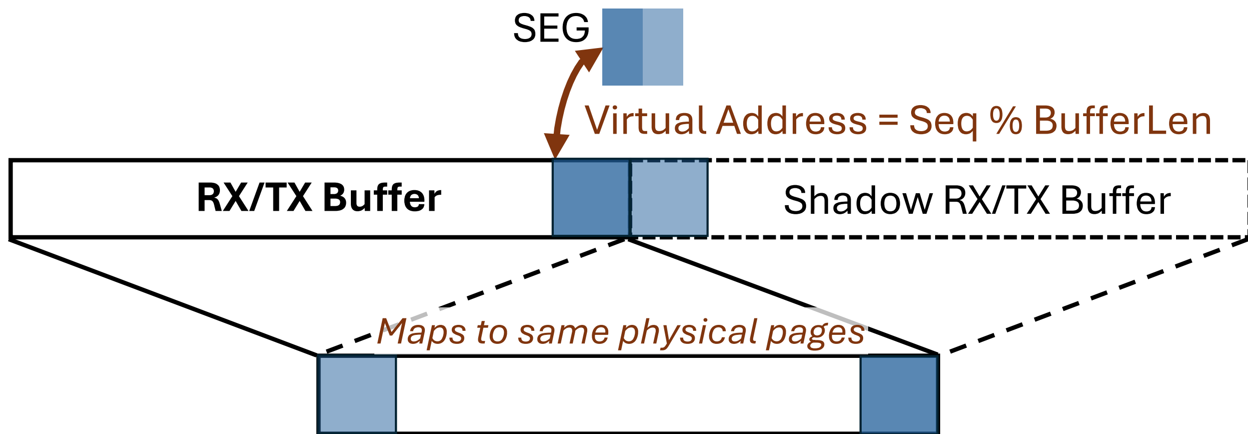

Figure 5.7: Sequence-address translation with double buffer in Laminar.

5.4 Implementation

Laminar's data-path is implemented on an Intel Tofino2-based switch, a commercially available RMT pipeline architecture. For DMA, we utilize RDMA. Laminar is implemented in 20,775 lines of code (LoC). The P4 code for the offloaded data-path has 2,597 LoC. The control-plane is split into two components: one on the host and another, `SwitchConf`, on the switch CPU. The host control-plane interacts with `SwitchConf` to configure switch data-path resources such as match-action tables. Moreover, `SwitchConf` runs the congestion control policy and processes all control plane notifications. `SwitchConf` interacts with the data-path via Intel's proprietary bfrt C++ API. They are implemented in 3,352 and 6,917 C++ LoC respectively. libTOE is implemented in 7,909 C++ LoC, utilizing `libibverbs` to interact with the RDMA NIC without kernel intervention. Our prototype leverages some RDMA and Tofino2 features for optimizations, described below.

DMA. libTOE interacts with the data-path via the RDMA verbs API. During initialization, it establishes an unreliable connection (UC) RDMA queue pair per context with the data-path.

Typically, each application thread uses a separate context, akin to per-core NIC queues in traditional network stacks. Connections owned by a context are multiplexed onto a single queue pair, avoiding RDMA NIC connection scalability bottlenecks. Both libTOE and the data-path use RDMA writes (with immediate for notifications) to transmit or receive data from host payload buffers. libTOE allocates per-connection payload buffers and registers them with the RDMA NIC, storing the buffer's virtual address and remote memory key in the data-path. Based on the available transmission credits, libTOE can issue RDMA writes larger than the MTU, relying on the NIC's segmentation offload to dynamically packetize the data. The egress pipeline assigns an RDMA sequence number to each packet. To ensure lossless RDMA operation, PFC is enabled between the host and the switch.

Interrupt-driven operation. By default, libTOE operates in polling mode, constantly querying the completion queue for events. However, after 10ms of inactivity, libTOE switches to interrupt-mode by leveraging RDMA completion event channels (`ibv_comp_channel`), enabling the application to sleep when waiting for IO.

Rate limiting. For rate limiting, Laminar utilizes Tofino2's metering units to efficiently compute per-flow transmission rates. The metering unit marks packets that exceed the configured congestion control rate, allowing the data-path to drop them. Tofino2 does not support modifying metering rates in the data-path, as is necessary to halve the congestion window during fast retransmission. We continue to rely on the `tx-credit` state variable in this case (§5.3.2.1).

Laminar-Laminar connections. Same-switch Laminar-Laminar connections require recirculation. For example, an RPC request sent from libTOE as an RDMA write is converted to a TCP packet by the switch data-path. To perform TCP reassembly for the peer, the data-path must

recirculate this packet, which is then converted into an RDMA write. The RPC response and ACKs also undergo recirculation. Recirculation adds latency ($\approx 750ns$) and is limited by the 3×100G recirculation port bandwidth. This limitation is specific to our testbed.

Limitations. Tofino2's checksum engine does not support computing or validating payload checksums, preventing our implementation from handling TCP and RDMA iCRC checksums. However, TCP and RDMA hardware checksum units are common and adding support would be straightforward.

5.5 Evaluation

We evaluate Laminar by addressing the following questions:

- **Performance:** Can Laminar deliver predictable μs-scale latency and high message rate for RPCs (§5.5.1), sustain terabit-scale throughput with near-zero CPU utilization (§5.5.2), scale efficiently to thousands of applications and connections (§5.5.1), and benefit real-world storage (§5.5.2.2) and key-value stores (§5.5.1.4)?
- **Efficiency:** Does Laminar improve performance-per-watt over host-based stacks (§5.5.1.4)?
- **Flexibility:** Can Laminar customize transport logic for critical data center applications (§5.5.3)?
- **Robustness:** Is Laminar resilient to loss and congestion, and does it ensure performance isolation between latency-sensitive and bandwidth-intensive workloads (§5.5.4)?
- **Generalizability:** Do Laminar's pipelined transport logic principles extend to FPGA-based SmartNICs (§5.5.5)?

Testbed. Our evaluation cluster comprises eight Intel Xeon Gold 6430 machines, each with 32-core (64 hyperthreads) running at 2GHz, 256GB RAM, and 126.5MB aggregate cache. Each machine is equipped with a RDMA-capable 200G Mellanox ConnectX-6 NIC, all connected to an 32×400G Intel Tofino-2 based Netberg Aurora 810 switch. We configure the NICs in 100G mode as our switch fails to negotiate our NICs into 200G speeds. Two machines are designated as servers, with Priority Flow Control (PFC) enabled for lossless RDMA, while the remaining machines function as clients. To ensure high performance and low latency, we disable turbo boost and set frequency scaling to performance mode. Power measurements are obtained via the IPMI interface of both the machines and the switch. `SwitchConf` runs on a 4-core (8 hyperthreads) onboard Intel Xeon D-2123IT CPU, and its power consumption is included in the switch power measurements.

Baselines. We compare Laminar performance against the Linux TCP stack, the TAS kernel-bypass TCP stack, and RDMA. Unless otherwise specified, clients use the TAS network stack to generate network traffic. TAS struggles with large flows due to high payload copy overheads, failing to saturate the 100G line rate with a single connection. To mitigate this bottleneck in such client workloads, we elide payload copying in TAS and transmit junk data instead (TAS-nocopy). We use identical application binaries that use sockets-based `epoll()` interface across all baselines, except for RDMA. For the RDMA baseline, we adapt our benchmarks to use the `libibverbs` interface. We use DCTCP as the default congestion control policy.

5.5.1 Remote Procedure Calls (RPCs)

RPC workloads demand low latency, high packet rates, and scalability across many applications and connections, requirements that heavily tax host TCP stacks. Laminar's bump-in-the-wire design efficiently delivers constant, low TCP processing latency even at multi-million segment

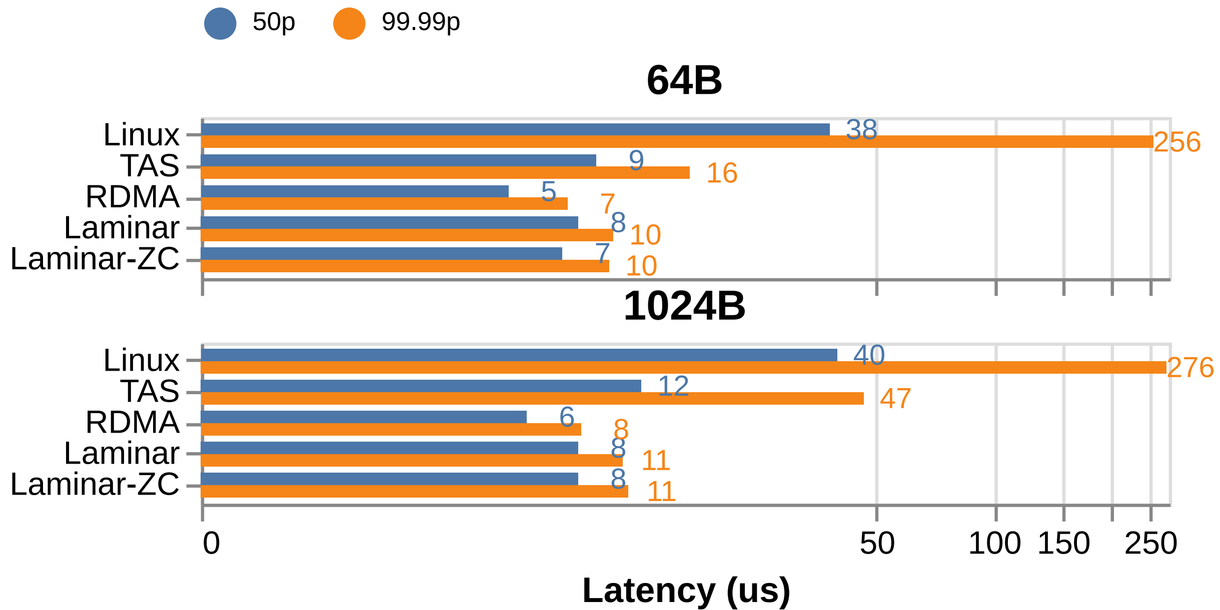

Figure 5.8: Laminar unloaded RPC round-trip time.

rates, fully eliminating all CPU overhead. We demonstrate the benefits with an echo server benchmark.

5.5.1.1 Unloaded latency

Figure 5.8 compares the round-trip time distribution of different network stacks for a single connection with one in-flight request, evaluated for two message sizes: 64B and 1024B.

Laminar's bump-in-the-wire design achieves low and predictable latencies, outperforming TAS by as much as 4.3× at the tail, and Linux by 5× at the median and 25.6× at the tail. Unlike the RMT-pipeline, CPUs fail to exploit the inherent parallelism in network processing tasks, and their sequential execution causes high latency. Additionally, CPU-based stacks such as TAS and Linux exhibit higher latencies for larger message sizes due to the overhead of copying the payload to/from network buffers. Supporting the sockets interface in Laminar requires an additional

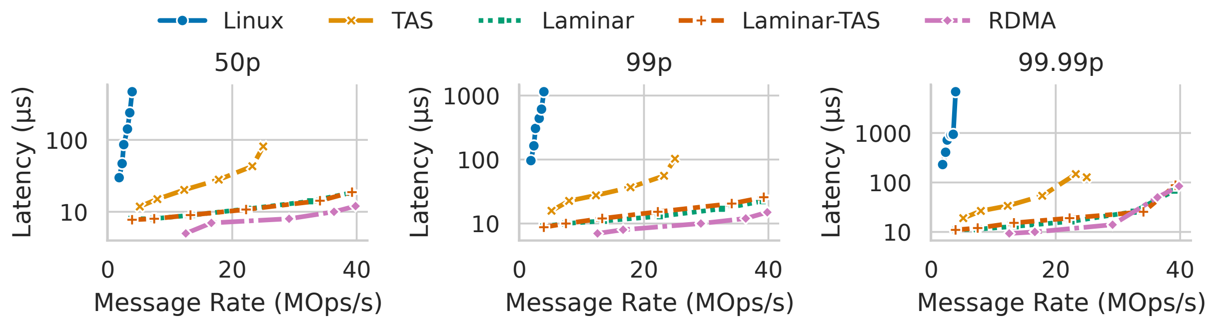

Figure 5.9: Load-latency across network stacks incl. Laminar (note different y-axis ranges).

copy from libTOE's per-connection payload buffers to user-supplied buffers, introducing a small latency overhead. For small RPCs, this overhead is negligible. The libTOE-ZC zero-copy interface eliminates payload copy overhead and incurs only a $2\mu s$ higher latency than RDMA.

We note that our measurements overestimate Laminar's true latency due to implementation-specific factors. Our setup bypasses egress processing for non-Laminar packets, reducing RTTs for baselines, whereas a conventional switch would require both ingress and egress processing. Additionally, recirculation in Laminar-Laminar connections introduces extra latency relative to RDMA (§5.4). An implementation leveraging a SmartNIC would reduce latency to $< 1\mu s$ of RDMA.

5.5.1.2 Load-latency

Figure 5.9 presents the round-trip latency profile of a 32-core echo server across stacks, as load increases. The benchmark scales connections across multiple clients, each leaving a single 64B RPC in-flight. Laminar performance closely matches RDMA, sustaining nearly 40 MOps/sec (1.67× TAS) with a $18\mu s/24\mu s$ median/99.99p tail latency. Recirculation in Laminar-Laminar connections adds $\sim 2\mu s$ relative to RDMA (§5.4). Both Laminar and RDMA experience 99.99p tail latency spikes as they approach the NIC's processing limits. A hybrid setup (Laminar server

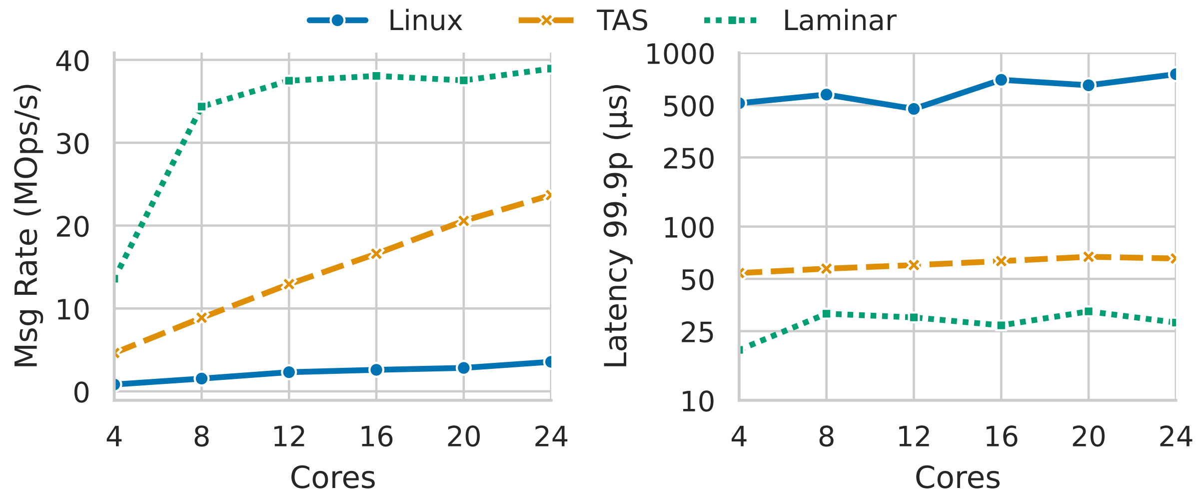

Figure 5.10: Laminar core scalability.

with TAS clients) achieves a similar peak throughput but with slightly higher latency due to client-side TAS overheads. Linux is especially inefficient, delivering 10× lower throughput and 78× higher latency than Laminar. Supporting the sockets interface in Laminar requires an additional copy among libTOE's payload buffers and user-supplied buffers, introducing overhead. The libTOE-ZC interface eliminates this overhead, bringing it on par with RDMA.

5.5.1.3 Scalability

We evaluate scalability along cores, connections, and pipeline capacity. We repeat the previous experiment sweeping across host, core, and connection counts, finding the throughput-latency knee for each configuration.

Cores. Figure 5.10 shows that Laminar scales linearly with server cores. Data-path performance is invariant to the number of cores, as Laminar multiplexes TCP connections onto per-core DMA channels (*contexts*) to steer payload directly to application cores (akin to aRFS [70]), avoiding RDMA's scalability limits. Performance peaks at 8 cores, where Laminar hits the NIC

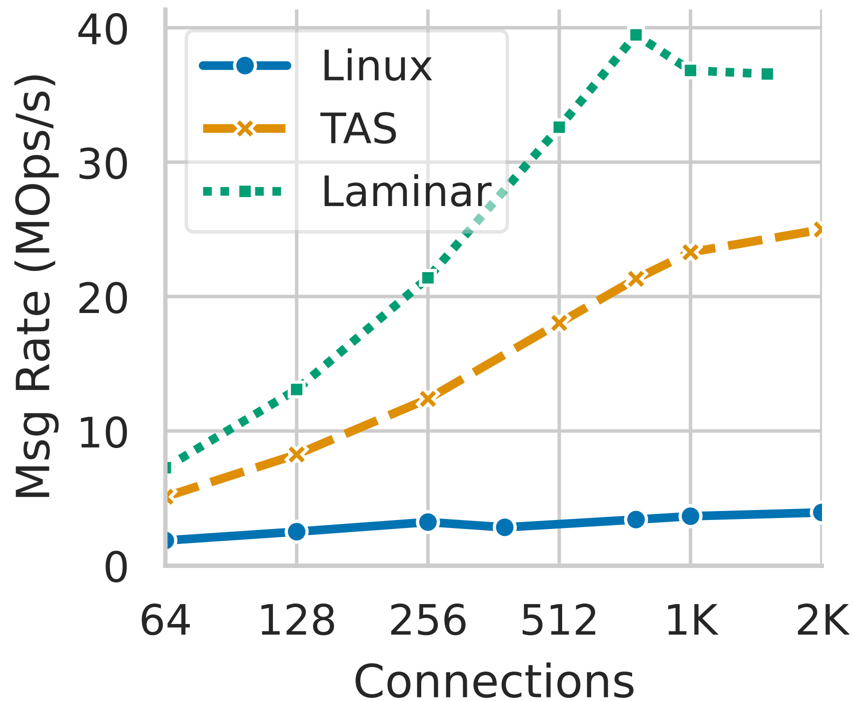

Figure 5.11: Laminar connection scalability.

line rate, surpassing TAS's peak throughput keeping tail latency 4.3× lower. Linux remains far behind.

Connections. Figure 5.11 showcases Laminar's superior connection scalability for a single host. Laminar saturates NIC bandwidth at 768 connections, achieving 1.8× TAS and 10× Linux throughput. Further connections lead to a slight drop in throughput, as we go beyond NIC line rate. Laminar accesses connection state via stage-local memory with constant access latency, sustaining up to 32K connections without performance loss. Higher connection counts can be supported by sharding TCP state tables across stages (§5.5.3). Other high-performance stacks degrade with many connections due to cache pressure [131] or limited on-NIC memory [259].

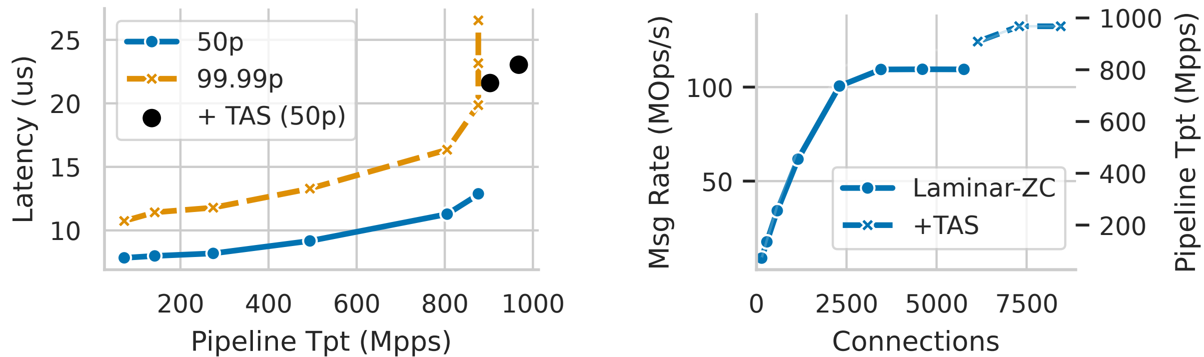

Figure 5.12: Laminar pipeline packet processing scalability.

Pipeline. Each RMT pipeline in our testbed is capable of processing 1.2 Bpps across 8 × 400G links. Using 8 hosts running Laminar, exchanging 8B echo RPCs, stresses the pipeline. Each RPC generates 8 packets through the egress pipeline—4 for the request (TX, RX, ACK generation, ACK processing) and 4 for the response. Figure 5.12 highlights pipeline scalability. Latency remains relatively flat and narrow up to 850 Mpps (71% utilization), after which recirculation bandwidth saturates and tail latency rises. Adding two more TAS clients (black dots) extends throughput close to 1 Bpps (80% utilization), or 130 MOps/sec in goodput, with aggregate pipeline throughput nearing 500 Gbps even with 8B payload packets. Latency is higher in this configuration due to client-side TAS overheads. We have tested scalability up to 8K concurrent connections and 256 contexts in this benchmark.

5.5.1.4 Energy Efficiency

We measure the power consumption of the TCP stacks using FlexKVS [136], a scalable, high-performance key-value store inspired by memcached. Short RPCs dominate these workloads, with host TCP stacks consuming up to 48% of per-request CPU cycles [259]. Our benchmark uses 1M uniformly distributed key-value pairs (8B keys, 64B values, 0.99 GET:SET ratio).

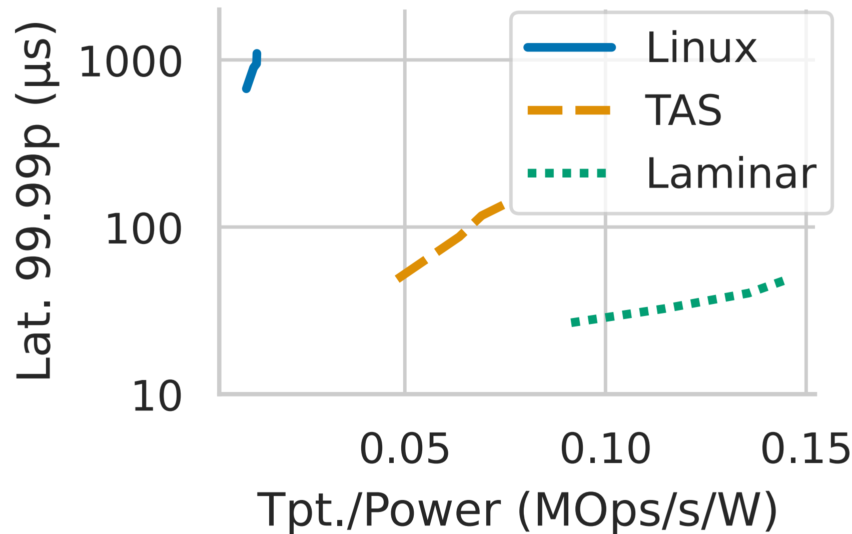

Figure 5.13: Throughput-per-watt and tail latency for FlexKVS key-value store workload.

Baseline power is first recorded for the server (including NIC) and switch. The workload then runs for 5 minutes while measuring total power and application performance. We vary server threads and connections, with each connection limited to a single in-flight request.

Figure 5.13 shows throughput-per-watt and tail latency across server configurations (# threads, # connections). Laminar consistently outperforms host stacks: even its worst case exceeds TAS's best by 1.2× in throughput-per-watt while delivering 5× lower 99.99p latency; its best configuration doubles TAS's throughput-per-watt and beats TAS's best tail latency. Linux fares far worse, with at least an order of magnitude lower efficiency and two orders higher latency.

Switch power varies by no more than 4W between baselines and Laminar, with efficiency driven primarily by reduced host CPU usage. RMT-based switches consume power comparable to non-programmable commodity switches of similar bandwidth [13, 38], in contrast to power-hungry FPGAs [20].

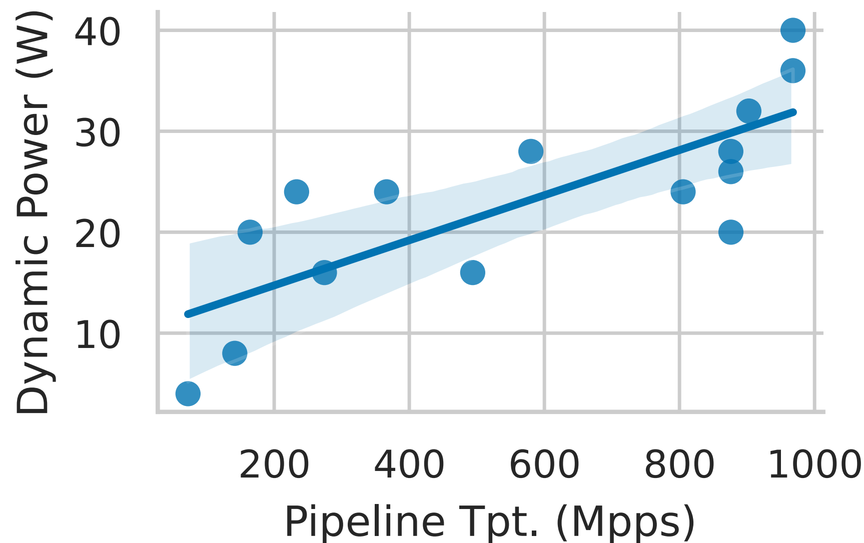

Figure 5.14: Pipeline dynamic power consumption with utilization.

Finally, Figure 5.14 breaks down dynamic power (increase from idle, measured over 10 minutes) as a function of pipeline throughput, showing Laminar processes TCP at ≈ 25 Mpps/W for 8B payload packets. These switch measurements include the on-board CPU, RMT pipeline, transceivers, and other fixed-function components (cf. Table 5.2).

5.5.2 Byte Streaming

Streaming bandwidth is critical for modern storage and AI workloads. CPU-based stacks like Linux and TAS fail to saturate line rate on a single core, even with hardware-assisted payload copy optimizations [44, 259]. Accelerating copies with specialized DMA engines [270] does not resolve this, as single-core protocol processing is the bottleneck (§2.3). Consequently, hardware transports like RDMA and Chelsio TOE, despite their inflexibility, are increasingly preferred for streaming [26, 88, 51, 234]. Can Laminar match their performance?

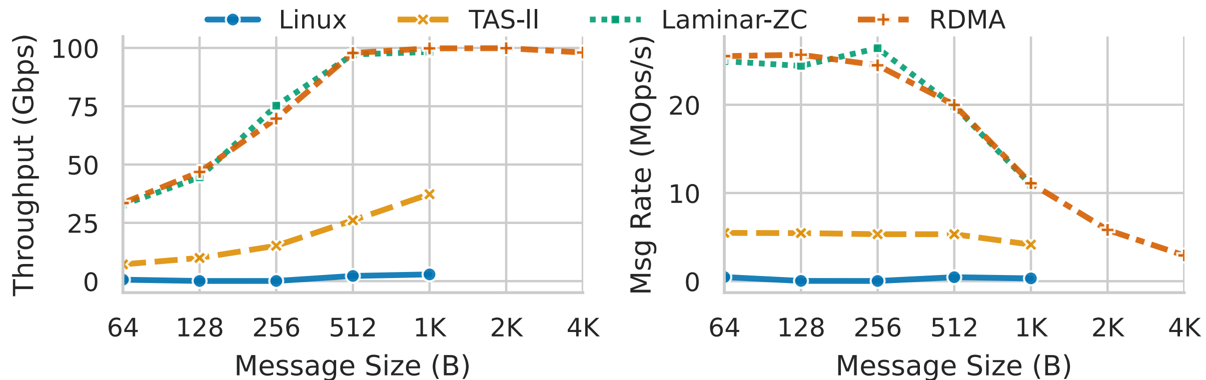

Figure 5.15: Laminar streaming performance with different segment sizes.

5.5.2.1 Single-core Throughput

In this benchmark, a TAS-nocopy client transmits packets in an open-loop (similar to iperf [117]) to a single server core that discards incoming payloads. To provide different intensities of protocol to payload processing, the benchmark varies the segment size by adjusting the maximum transmission unit (MTU). To avoid payload copying in the sockets layer, we use the libTOE-ZC interface, comparing against the equivalent low-level TAS interface (TAS-ll), though TAS still requires copying the payload between network buffers and application memory.

Figure 5.15 shows that Laminar matches RDMA's single-core packet rate of ~25M pkts/sec, independent of packet size, reaching line rate for higher MTUs, sufficient to exceed 1.6 Tbps ($= 25M * 8 * 8K = 1600G$) with 8K MTU. The bottleneck remains NIC and CPU processing [129], as the RMT pipeline's processing capacity is far greater (cf. Figure 5.12). TAS performance is limited to ~5M pkts/s per core. It reduces for packets >512B due to copy overhead. This bottleneck is also present in NPU-based stacks like FlexTOE [259]. Linux, again, performs poorly.

5.5.2.2 NVMe-over-TCP

Large-scale disaggregated storage is rapidly gaining adoption in modern data centers. We integrate SPDK [83], an open-source user-space storage stack, with Laminar to show that NVMe-oTCP can rival RDMA in both performance and efficiency. We extend SPDK with a new transport implementation that interfaces with Laminar through libTOE-ZC, delivering NVMe commands together with their payload into per-connection buffers that are directly accessible by SPDK, avoiding intermediate copies and enabling high throughput. A key challenge in supporting NVMe-oTCP is that NVMe devices exploit parallelism by allowing commands to complete out-of-order (OOO). To support this, we provide a modified version of libTOE-ZC, which provides explicit mechanisms to free connection buffer space out-of-order, while tracking the buffer head. NVMe-oRDMA achieves the same functionality by requiring SPDK to post multiple receive buffers to the RDMA work queue, and each buffer is independently freed upon corresponding command completion.

We evaluate our NVMe-oTCP storage target using 4KB random writes. In our setup with a single NVMe SSD Laminar can easily exceed the 1M IOPS disk bandwidth, even when only partially utilizing a server core. To shift the bottleneck from the device to the CPU, we configure a RAM block device [273], which emulates Non-Volatile Memory (NVM) or Compute eXpress Link (CXL)-attached memory.

Figure 5.16 compares the performance of a single-core SPDK NVMe-oF target using the Linux kernel stack, the Laminar transport, and RDMA, as IO-depth increases. We use the SPDK perf [274] tool on a remote client node[1] that drives the load through Laminar's POSIX sockets interposition layer. We use an MTU of 4,400 with Linux, 4,096 with RDMA and Laminar.

[1] SPDK perf can generate more load compared to fio [274].

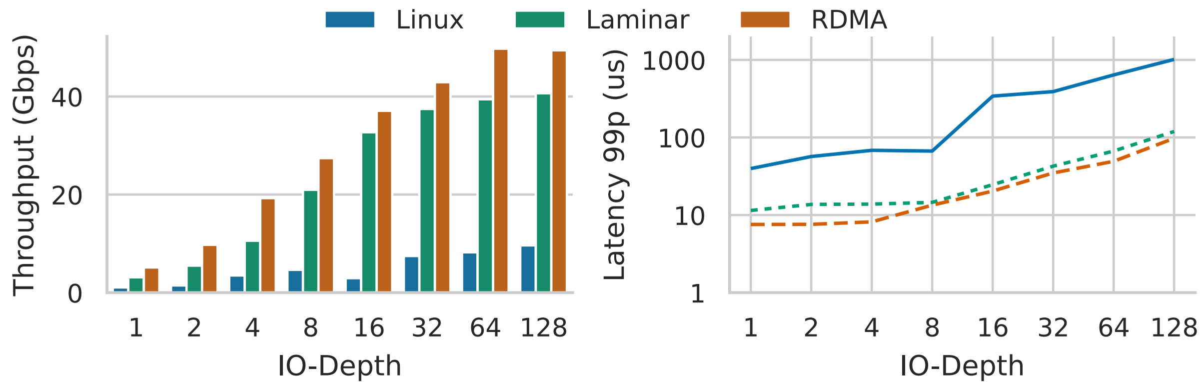

Figure 5.16: SPDK NVMe-oF performance: 4K Random Writes with different network stacks.

Laminar closely matches RDMA in both throughput and tail latency until the server core saturates at high IO depths due to storage stack processing and copies within the RAM block device. Laminar's peak throughput is slightly lower due to the overhead of processing periodic SYNC grants, and its latency slightly higher due to client-side copy overheads in the sockets layer. In contrast, Linux incurs extra copies and higher stack overheads, leaving fewer cycles for the application and resulting in $> 4\times$ lower throughput.

5.5.3 Flexibility

Unlike hardware transports and FPGAs, Laminar provides flexibility via high-level P4 programmability. As shown in Table 5.2, it consumes only $\sim$ 25% of the RMT-pipeline resources while supporting 32K connections and 1K contexts. Laminar's data-path sustains 3.2 Tbps (8×400G) line-rate or 1.2 Bpps at a total processing latency (excluding queuing delays) of 395ns and a power consumption of 2.81W for the RMT pipeline (excluding SerDes, MAC, TM). With basic L2/L3 forwarding already implemented, the low footprint leaves ample headroom for new functions. We demonstrate flexibility by accelerating a shared log [28] with transport customization, and adding transport extensions like Timely and delayed ACKs.

Resource	Usage
Stages	14/20
Processing Latency	395 ns
MAU Power (*worst case* per-pipe)	2.8 W
Memory	
SRAM	19.8 %
Map RAM	14.5 %
TCAM	0.4 %
Compute	
VLIW Instruction	9.5 %
Stateful ALU	26.3 %
Match Crossbar	10.8 %
Gateway	26.6 %

Table 5.2: Tofino2 RMT pipeline resource usage for Laminar.

Shared log. Databases [27], replicated state machines [68], and storage clusters [28] rely on shared logs to linearize state updates [170, 28]. Inspired by NoPaxos [153], we accelerate a shared log microbenchmark with a tailored Laminar API: the application registers its log buffer and client connections, and Laminar sequences incoming records across connections, appending them directly to the log via a buffer tail pointer within the data-path [2]. Our design guarantees that records from the same connection are appended to the shared log in order, resulting in a linearized shared log. If out-of-order segments arrive, Laminar reserves space up to the highest sequence number, creating gaps; the application later fills these gaps using the corresponding segments, which are redirected to it by Laminar. This extension required only 156 P4 LoC and 0.29W additional power.

With up to 32 clients (64–1024B records, 256 in-flight per client), Laminar saturates line-rate using one core, while TAS requires five (3 TAS + 2 app). The efficiency comes from

[2] Clients ensure log records are not fragmented across TCP segments.

removing sequencing, protocol handling, and payload copying from the CPU in the common case (no packet loss). Achieving similar acceleration with RDMA is challenging, especially for variable-sized records, due to requiring multiple round-trips for sequencing and writing each record [129].

Timely. Congestion control remains an evolving research area with numerous proposals addressing diverse network conditions and performance objectives [267]. Timely [184], which infers congestion from round-trip time (RTT) variations, is a notable example. We implement Timely by leveraging Tofino2's hardware timestamping to add TCP timestamps [36] and compute per-flow RTT exponentially weighted moving averages (EWMAs) in the data-path using its low-pass filter unit [96]. The control plane (290 C++ LoC) periodically adjusts transmission rates based on these RTTs, while the data-path implementation requires only 64 P4 LoC, ≤5% additional resources, and 0.37W extra MAU power.

Delayed ACKs. Delayed or cumulative ACKs [39] acknowledge every M consecutively received segments, reducing sender and network load [16]. For instance, for RPCs, piggybacking the ACK onto the response can halve sender overhead. In Laminar, this requires a counter to track unacknowledged segments, issuing an ACK pseudo-segment when the threshold M is reached, or immediately for out-of-order segments. Transmitted segments reset the counter. The control plane manages the delayed ACK timer. This addition required 42 P4 LoC, +3% SRAM, and +0.05W power. Benchmarking with a Laminar echo-server and TAS client (single fast-path core) shows delayed ACKs improve peak throughput by 24% by lowering client CPU load.

5.5.4 Robustness

Robustness to packet loss and congestion is vital for data center transports.

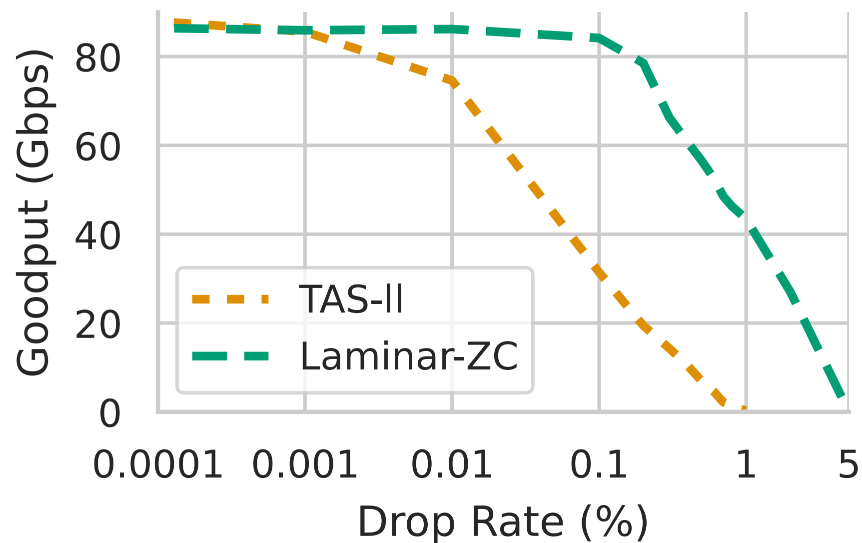

Figure 5.17: Laminar streaming performance under random packet drops.

5.5.4.1 Packet drops

We measure the streaming throughput while randomly discarding packets with varying probability. This workload maintains many in-flight packets, ensuring losses trigger OOO processing. Due to recirculation bottlenecks (§5.4), we always use a TAS sender. This isolates the evaluation to receiver-side reassembly.

Figure 5.17 shows that Laminar sustains full throughput up to a 0.1% drop rate, whereas TAS suffers an earlier decline, showing 2× lower throughput at 0.1% loss. Both Laminar and TAS provide 1-OOO reassembly on the receiver, but Laminar's faster acknowledgments accelerate recovery. Linux[3] withstands higher loss rates due to its more sophisticated reassembly and recovery mechanisms, including selective acknowledgments. In contrast, the ASIC-based Chelsio TOE suffers a steep performance decline even under moderate packet loss rates. Traditional

[3]Results for Linux and the Chelsio TOE can be found in [259].

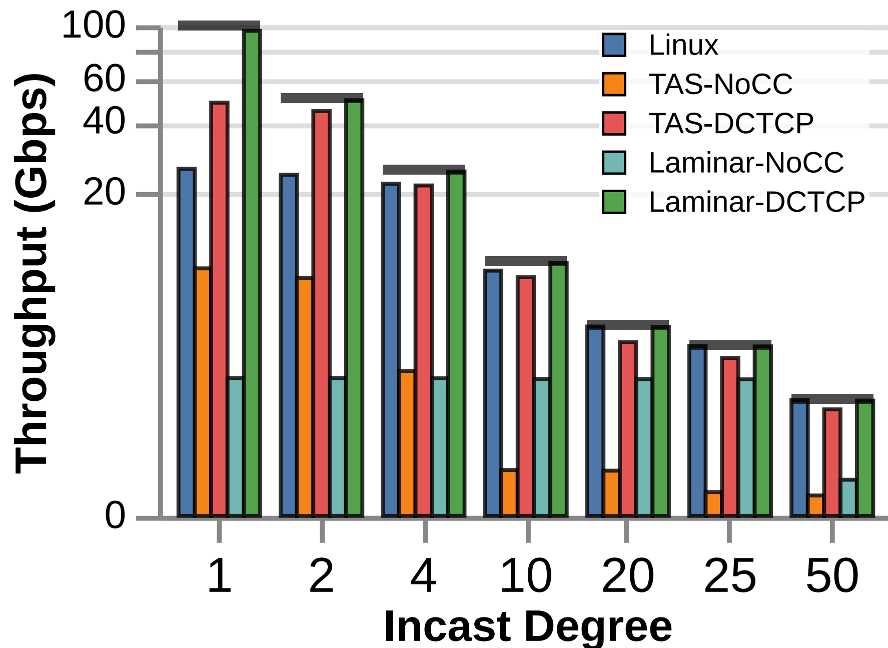

Figure 5.18: Laminar streaming performance under incast.

RDMA drops all out-of-order segments, leading to poor performance under packet loss [185, 268].

5.5.4.2 Congestion control

To evaluate congestion control, we simulate incast by adding a traffic shaper frontend to a TAS receiver. The shaper limits ingress bandwidth (corresponding to the incast degree), uses tail-drop, and marks ECN above a queue occupancy threshold (determined experimentally as 130). Figure 5.18 presents the results for a single-connection streaming workload under varying incast degrees. Black ticks mark the optimal bandwidth achievable at each degree.

Laminar with its control-plane-driven DCTCP and credit-based rate enforcement effectively regulates transmission, minimizing loss and matching the shaped line rate. In contrast, Laminar

without congestion control (Laminar-NoCC) transmits aggressively, causing excessive tail drops, even when traffic shaping is disabled (incast degree=1) since it significantly outpaces the TAS receiver, resulting in up to 50× lower throughput. TAS and Linux also regulate their rates effectively but fail to achieve maximum possible throughput at low incast degrees (≤ 2) due to transport stack inefficiencies.

5.5.4.3 Selective Acknowledgments (SACK)

To enhance our transport logic, we implement 1-block SACK in Laminar, encoding the OOO interval in ACKs to aid sender recovery. Due to Tofino2 compiler errors, we cannot experimentally evaluate it. Instead, we evaluate Laminar's 1-OOO SACK in simulation, using TCT's ns-3 [155] (TCT disabled). We model a 96-node leaf-spine data center topology with latency-sensitive partition-aggregate incasts of 8KB queries, competing with bandwidth-heavy background flows. We extend ns-3's TCP to support varying reassembly fidelity, tracking up to N OOO intervals. 0-OOO drops all OOO segments, while max-OOO tracks all segments with unlimited state.

Table 5.3 compares the impact of different OOO fidelities. As expected, the go-back-N baseline (0-OOO) performs poorly, greatly inflating tail FCTs. In contrast, Laminar's 1-OOO interval design substantially improves tail FCT, especially under DCTCP, which reacts proactively to congestion. Beyond 1–2 intervals, additional fidelity yields diminishing returns: congestion losses are typically bursty, and a few OOO intervals suffice for efficient SACK-based recovery. Overall, Laminar's 1-OOO approach strikes a balance between loss recovery efficiency and data-path state complexity, and is directly implementable on today's RMT hardware.

OOO	Flow Completion Time (FCT) [ms]							
	Foreground				Background			
	90p		99.9p		90p		99.9p	
NewReno								
max	4.97	—	17.05	—	68	—	283	—
0	5.80	×1.17	18.49	×1.08	91	×1.34	317	×1.12
1	5.03	×1.01	17.05	×1.00	84	×1.23	300	×1.06
2	4.97	×1.00	17.05	×1.00	78	×1.15	295	×1.04
DCTCP								
max	4.69	—	12.95	—	62	—	278	—
0	5.13	×1.09	28.52	×2.20	71	×1.14	283	×1.02
1	4.70	×1.00	12.91	×1.00	62	×1.00	279	×1.00
2	4.70	×1.00	12.98	×1.00	63	×1.01	275	×0.99

Table 5.3: Simulated Flow Completion Times (FCTs) across Out-of-Order (OOO) tracking fidelity levels.

5.5.4.4 Performance isolation

Run-to-completion based TCP stacks struggle to isolate the performance of latency-sensitive (LS) and bandwidth-intensive (BI) connections. Specifically, TAS maps connections to cores via hashing, mixing LS and BI connections, which causes interference, inflating LS tail latency and reducing BI bandwidth [44, 45].

To evaluate performance isolation, we co-locate an echo server RPC workload with 32 latency-sensitive (LS) connections (single in-flight request) and a streaming workload with 4 bandwidth-intensive (BI) connections (5 Gbps each). We first measure both in isolation, then compare with concurrent execution. We provision TAS with two fast-path cores to avoid bottlenecks.

As depicted in Figure 5.19, Laminar ensures optimal performance for both LS and BI flows, owing to its consistent per-packet processing overhead, steering payload directly to corresponding

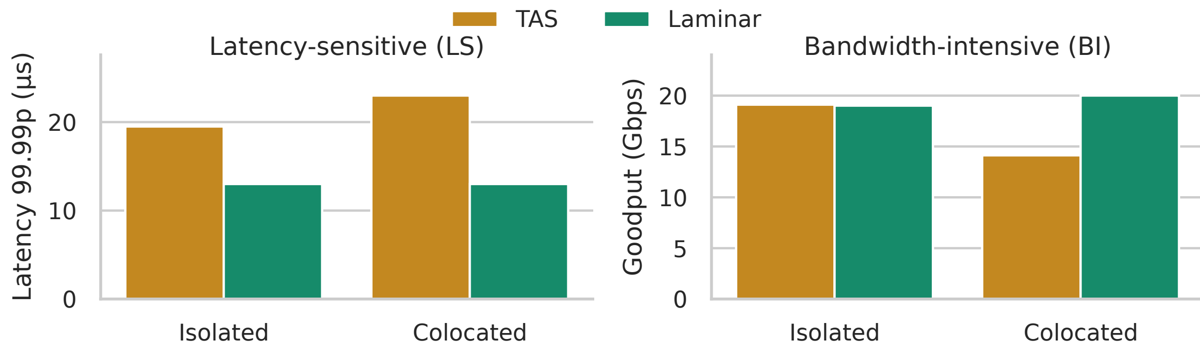

Figure 5.19: Laminar performance isolation for latency-sensitive and bandwidth-intensive connections.

application cores via independent DMA channels, preventing cache pollution. In contrast, TAS exhibits an 18% increase in tail latency for LS flows and a 30% decrease in bandwidth for BI flows under co-location.

5.5.5 Generalizability

Prior FPGA work implements TCP as a monolithic circuit, requiring ever shorter timing and more power to scale. Laminar eases these constraints by enabling fine-grained per-state pipelining of TCP, even within a single connection. To investigate whether Laminar generalizes to FPGA-based SmartNICs, we port Laminar's stateful egress pipeline to the AMD Alveo U250 [20] using the VitisNet P4 compiler [19]. Our design meets 10ns timing for 300 Mpps while using only 2.5% of LUTs and 0.78% of BRAM to support 32K connections, consuming 4.93W worst-case power. The low resource footprint leaves headroom for replicating the pipeline and sharding connections for higher rates and connection counts.

By comparison, on a similar FPGA and under the same timing constraint, ToNIC [23] peaks at 100 Mpps and consumes over 20% of LUTs and BRAM, while supporting only 1K connections. Beehive [156] peaks at 2.6 Mpps with comparable resource use. Direct comparison is limited

since each design implements a different subset of TCP features—ToNIC drops TCP's byte-stream abstraction in favor of a fixed segment size, while Beehive omits OOO handling and congestion control but implements connection handshake.

5.6 Conclusion

Laminar redefines the design space of high-performance TCP stacks by showing that full transport-layer functionality fits within the constraints of the RMT architecture. By embracing pipeline-parallelism, optimistic concurrency, and lightweight pseudo-segment updates, Laminar delivers terabit-scale throughput, low latency, and hardware-class efficiency while preserving the flexibility and compatibility of software-defined networking. Our Intel Tofino2 prototype demonstrates that RMT data planes can support stateful, loss-resilient, and scalable transport, setting a foundation for the next-generation SmartNICs and data center networks.

Chapter 6

Related Work

Prior efforts to accelerate network-intensive applications and optimize transport processing can broadly fall into three categories: (1) **Re-architecting** CPU transport stacks through optimized API design, efficient driver and NIC interfaces, and selective hardware assistance. (2) **Transport offload** to specialized hardware, including fixed-function TOEs and RDMA NICs, as well as programmable FPGAs and SmartNICs. (3) **Co-designing** applications, transport protocols, and hardware. We next survey these efforts to contextualize our contributions.

6.1 Software Stack Improvements

Significant efforts have been dedicated to rearchitecting software transport stacks to reduce CPU overheads and improve scalability.

6.1.1 Efficient Application Interfaces

The Linux kernel TCP stack has been enhanced with asynchronous, scalable event-driven I/O interfaces such as epoll, aio, and io_uring, which improve the overlap of computation and I/O, and amortize expensive system call overheads [3, 161, 162, 2]. Features like MSG_ZEROCOPY, sendfile(), and splice() eschew unnecessary data copies between application and kernel buffers [69, 171, 9]. These advancements build upon earlier academic proposals, including LAIO [75], FlexSC [271], and MegaPipe [98]. Systems like IX [32] and StackMap [290] introduced native, zero-copy, event-driven APIs for in-kernel TCP stacks. These ideas have influenced TAS [138], which we inherit. Both FlexTOE and Laminar adopt low-level, event-driven, asynchronous APIs with zero-copy support, but also retain compatibility with the POSIX sockets API.

6.1.2 Kernel-bypass

Kernel-bypass stacks, such as Arrakis [228], mTCP [120], Snap [172], and Demikernel [292], eliminate kernel overheads by minimizing or entirely removing kernel involvement in the data-path. These stacks enable applications to directly interact with the NIC by leveraging hardware virtualization or dedicating the NIC to a single application. However, systems like Arrakis and mTCP lack mechanisms to enforce system-wide security and resource management policies. TAS and Snap address this limitation by splitting the TCP stack into three components: a trusted control plane, a shared data-path that enforces policies, and an untrusted application interface, achieving both high performance and robust security [138, 172]. Our systems build upon this architecture, further enhancing isolation and security by offloading the shared data-path to specialized network hardware. This approach eliminates side-channel vulnerabilities arising from sharing CPU caches and pipelines with untrusted applications.

Hardware transports such as RoCE [56] also enable kernel-bypass by allowing applications to directly interact with the NIC.

6.1.3 Busy Polling

High-performance transport stacks use busy polling to reduce latency and improve throughput [82, 138, 172]. By actively polling the NIC for packets instead of relying on interrupts, these stacks achieve lower latency and higher performance [138, 120, 172]. However, busy polling increases CPU usage, requiring dedicated cores and higher power consumption, making it unsuitable for all workloads. Notably, polling is not fundamental to kernel-bypass stacks — systems like TAS and Snap can operate in interrupt mode and dynamically scale data-path CPU resources based on load [138, 172], while the Linux kernel stack also supports busy polling via the `SO_BUSY_POLL` socket option [10]. Laminar and FlexTOE eliminate CPU needs for data-path TCP processing. By default, the libTOE application interface busy polls to minimize latency for high-demand applications, but switches to interrupt mode when idle to save power.

6.1.4 Hardware Assistance

Modern NICs incorporate stateless offload features like segmentation [71, 114], large-receive [71], and checksum offload [173, 114] to reduce CPU overhead during large data transfers. Hardware-assisted flow steering [181, 70] improve scalability by directing packets and interrupts to application-shared cores [227, 160]. Direct Cache Access (DCA) (e.g., Intel DDIO [113]) allows NICs to copy packets directly into the L3 cache. Laminar and FlexTOE go beyond these capabilities by offloading the entire TCP data-path, steering payloads and interrupts directly to application cores while transparently benefiting from DCA. Additionally, some software stacks use CPU DMA engines (e.g., Intel I/OAT [112]) to offload memory copy operations [6].

These techniques provide little benefit to dominant short RPC workloads, where the protocol processing overheads dwarf payload copy costs. Our stacks eliminate all payload copy overheads through a native zero-copy design.

6.1.5 Driver Optimizations

Numerous optimizations have been proposed to improve the NIC-driver interface and driver design to reduce CPU overheads, reduce PCIe communication costs, and improve scalability. PacketMill [78] employs code-optimization techniques and efficient packet metadata management to significantly improve driver efficiency. NIQ [81] minimizes the number of PCIe transactions, inlines small payloads inside descriptors, and develops efficient polling mechanisms to reduce latencies. TinyNF [230] redesigns the NIC driver to simplify packet buffer reuse to develop a formally verifiable driver, that also improves throughput. Reframer [90] reorders network packets to increase spatial and temporal traffic locality to improve cache efficiency. Enso [245] designs a streaming interface for NIC-application communication instead of a packetized interface typically found in NIC drivers, to reduce software overheads, reduce PCIe bandwidth, and improve cache hit rates. Both FlexTOE and Laminar employ a streaming interface to efficiently transfer payload between the NIC and application cores, eschewing packetization and buffer reuse complexity, a design later echoed in Enso [245]. Further, they adopt the ideas described in NIQ, and the NVMe specification [77] for efficient PCIe communication. Finally, by leveraging the deterministic RMT architecture for TCP processing, Laminar eliminates caching altogether, making its performance independent of traffic locality. Looking ahead, both FlexTOE and Laminar stand to benefit from emerging cache-coherent host-NIC interconnects like CCNIC [250], which offers lower latency and reduced CPU overhead.

6.1.6 Scalable Architectures

As CPU speeds plateau, streamlining TCP processing and scaling it across multiple CPU cores has become essential for achieving high performance. TAS [138] separates the TCP data-path into a fast path optimized for common-case processing, and a slow path that handles uncommon scenarios — a high-level architecture that both FlexTOE and Laminar adopt. Snap [172] decomposes the host network stack into coarse-grained functional modules, such as packet switching, virtualization, and shaping, enabling independent scaling across multiple cores. Building on this idea, FlexTOE further decomposes the TCP data-path into a fine-grained, data-parallel pipeline, allowing packets — even from the same connection — to be processed in parallel across multiple NPU cores. FlexTOE was among the first systems to demonstrate that such fine-grained TCP parallelism yields significant scalability benefits even on host CPUs. Similarly, NetChannel [45] disaggregates Linux's static packet processing pipelines into loosely coupled layers that scale independently and integrates a dynamic scheduler to multiplex packets based on load. Finally, Virtuoso [275] re-organizes a kernel-bypass TCP stack to improve resource utilization and performance isolation in multi-tenant environments by minimizing layering overheads and enabling fine-grained resource accounting. Additional network virtualization layers can be integrated atop FlexTOE and Laminar, as prior work has shown feasibility of such offloads on in-network accelerators [192, 18]. Laminar's deterministic line-rate TCP processing further simplifies resource accounting and ensures isolation among tenants. Policy extensions for fair sharing can also be flexibly incorporated into both Laminar and FlexTOE in future work.

6.2 Transport Offload

Transport offload is not a new idea; Early prototypes and conceptual designs date back over two decades [186, 232]. Historically, the cost and operational complexity limited widespread adoption. However, with stagnating CPU performance and exploding workload demands, transport offload is now gaining traction in modern data centers [268, 26]. Commercially, two main forms exist today: TCP Offload Engines (TOEs) [51] and RDMA-over-Converged Ethernet (RoCE) [60]. Both are hardwired, offering little flexibility to modify or extend transport logic for new workloads or protocols with vendor involvement and expensive hardware redesigns. As a result, their deployment is still limited to niche storage and high-performance computing clusters, limited by their inflexibility. Large-scale operators such as Google, Microsoft, and Amazon have reported substantial challenges deploying these fixed-function offloads at scale [267, 50, 95], leading them to develop custom, workload-specific solutions instead. Recent research efforts have explored ways to infuse flexibility into transport offload. We survey these efforts below.

6.2.1 Hardware Transports

Amazon's Scalable Reliable Datagram (SRD) transport [50], implemented on Nitro NICs [18] for HPC clusters, uses packet spraying and a fixed BBR-like congestion control [47] to sustain high throughput but forgoes TCP and RDMA ordering guarantees, pushing complexity to applications. Google's 1RMA [267] builds on the reliable Aquila fabric [91] with software-based congestion and flow control for flexibility, but onloads ordering, recovery, and segmentation to the CPU. Falcon [268] improves on 1RMA with hardware support for ordering and loss recovery, leveraging advanced TCP features such as Recent acknowledgement (RACK) [121] for efficient loss recovery. However, it buffers out-of-order packets on-NIC and tracks only a limited

number of in-flight packets (128), constraining scalability at high bandwidth-delay products. Moreover, Falcon's RACK requires $O(\text{window})$ operations per packet arrival, increasing power and tightening timing constraints. In contrast, Laminar demonstrates that maintaining a single out-of-order interval with efficient data structures suffices for high-performance TCP. Similarly, Ultra Ethernet Transport (UET) [103] targets AI/ML workloads by adding multi-path spraying and programmable congestion control to RDMA. These designs remain limited in flexibility and generality, while Laminar and FlexTOE are fully software-programmable. Orthogonal features such as multi-path spraying and advanced loss recovery can be integrated into our systems in future work.

6.2.2 FPGA Offloads

FPGAs are a highly expressive platform for high-performance transport offload [233, 80]. Microsoft was an early adopter of FPGA-based NICs for offloading network virtualization in Azure public cloud [80]. ClickNP [152] provides a modular programming abstraction for offloading network functions to FPGAs. Additionally, it automatically exploits pipeline or data parallelism based on module characteristics to maximize throughput. However, ClickNP does not support complex, stateful protocols like TCP. FlexTOE take inspiration from ClickNP's modularity and parallelism techniques to offload the TCP data-path to NPU-SmartNICs (§4). Tonic [23] provides building blocks for flexible transport protocol offload to FPGA SmartNICs and demonstrates, in simulation, that high-performance, flexible TCP offload might be possible. However, it only implements the transmit-side data-path, omits TCP flow control, and abandons TCP's stream abstraction, requiring fixed segment sizes. While FPGAs enable flexible and high-performance offloads, their near-hardware programming model makes complex protocol offloads, like TCP, difficult and slow, relative to agile software development life cycles on

CPUs [233]. ZeroNIC [270] co-designs the in-kernel TCP stack with an FPGA copy engine on the NIC to eliminate payload copy overheads for bandwidth-intensive flows. It provides no benefit for latency-sensitive flows, where protocol processing overheads dominate and execute on the CPU. Beehive [156, 242] designs a modular, FPGA-based network stack for direct-attached accelerators, providing OS-like abstractions, Network-on-Chip (NoC)-based inter-module communication, and tooling for automated scaling, load balancing, and debugging to improve development velocity [157]. The prototype lacks integration with general-purpose applications running on the host CPU, and TCP features such as out-of-order processing and congestion control. Critically, both Tonic and Beehive implement TCP transport logic as a monolithic circuit, requiring ever shorter timing and power to scale to higher bandwidths. In contrast, Laminar generalizes to FPGAs, and achieves fine-grained, per-state pipelining of TCP, even for a single connection. Under comparable timing constraints, Laminar achieves 3× higher packet rates compared to Tonic, and two orders of magnitude higher than Beehive's TCP tile, while using comparable or lower FPGA resources and power consumption (§5.5). We believe Laminar's TCP transport logic can be incorporated into Beehive's modular framework to enhance performance and enable integration with direct-attached FPGA accelerators, but leave this exploration to future work. NanoPU [111, 24] and Dagger [147] implement FPGA-based RPC transport stacks, primarily advancing specialized CPU-NIC co-designs: NanoPU introduces a register-based interface to RISC-V cores, while Dagger redesigns the memory interconnect to reduce latency and improve efficiency. Both target specific RPC workloads. These efforts are orthogonal to FlexTOE and Laminar, and optimized CPU-NIC interconnects could further enhance our systems in future work.

6.2.3 CPU-based SmartNIC Offloads

Arsenic [232] is an early example of flexible packet multiplexing on a SmartNIC. Amazon Nitro [18] offloads network virtualization and security tasks to an SoC-based SmartNIC. Google's Falcon [268] pairs ASIC transport engine with on-NIC CPU-driven congestion control for flexibility. IO-TCP [144] offloads disk I/O and TCP payload transfer to off-path SmartNIC cores to reduce host CPU overhead for content delivery, but the TCP data-path remains on the host. In general, CPU-based SmartNICs are off-path devices, where the on-board processors reside outside the critical path of network packet flow, lacking the ability to directly manipulate packets as they traverse between the host and network. These cores are lower performance and have lower core counts than host CPUs [175, 168, 166, 259], though general-purpose enough to run Linux, offering limited scalability and efficiency for full transport offload.

6.2.4 NPU-based SmartNIC Offloads

Similar to Falcon, SCR [296] drives packet pacing and congestion control of fixed-function RDMA using NVIDIA Bluefield-III's on-NIC NPU cores. This provides some flexibility, but the transport logic remains hardwired. AccelTCP [188] offloads TCP connection management and splicing [171] to NPU-SmartNICs, but keeps the TCP data-path on the host using mTCP [120]. Typical data center workloads utilize established connections for data transfer, making connection management offload less beneficial [23]. FlexTOE enables full TCP offload on NPU-SmartNICs by decomposing the TCP data-path into a fine-grained data-parallel pipeline (§4). However, FlexTOE's transport logic scalability is constrained by single-core NPU performance for a single connection. Laminar overcomes this limitation by leveraging the RMT pipeline to achieve full pipeline parallelism for TCP state management (§5).

6.2.5 RMT-pipeline Offloads

GEM was the first to leverage the RMT-pipeline to implement lightweight RDMA primitives to access external memory from the switch data plane [142, 140, 248]. Laminar takes inspiration to offload the entire TCP data-path within RMT. ConWeave [272] implements a network load balancing framework for RDMA utilizing the switch RMT-pipeline to buffer out-of-order packet arrivals. As an alternative to Ethernet Forward Error Correction (FEC), LinkGuardian [127] utilizes the RMT-pipeline on an Intel Tofino2 switch to implement link-local retransmissions for corrupted packets due to faulty links, improving TCP and RDMA flow completion times (FCT). Several proposals have leveraged the RMT-pipeline to implement transport features such as congestion control, network load balancing, and packet scheduling [256, 258, 257, 92]. None of the above provide transport offload, nor do they obviate the need for an end-to-end reliable transport protocol like TCP implemented by Laminar and FlexTOE. R2P2 [145] co-designs a UDP-based RPC protocol with an RMT-based switch data plane to achieve load balancing. NanoTransport [24] optimizes RPC applications by leveraging P4 programmability to express transport logic, though it relies on fixed-function FPGA hardware for reassembly and packetization. Laminar demonstrates that these functions can be fully realized using RMT hardware.

6.2.6 GPUs as Network Accelerators

GPUs have long been explored as accelerators for network processing tasks , owing to their highly parallel architecture. Early work focused on easily parallelizable tasks like packet routing and encryption [97, 276], while GASPP [279] demonstrated stateful offloads like TCP reassembly. However, subsequent studies found that applying GPU-style optimizations — such as memory latency hiding and vectorization — directly on CPUs yields better resource efficiency [132].

Moreover, commodity NICs now support inline hardware acceleration for compute intensive tasks like encryption and compression, further reducing the need for GPU offloads. Given their high compute density, floating point throughput, and memory bandwidth, GPUs are better suited for applications like AI/ML than transport offload. Still, FlexTOE's fine-grained parallelism techniques could extend to GPUs, as NPUs share similar architectural traits — massive parallelism, fast context switching, and deep memory hierarchies — though optimized for efficient packet processing under tighter resource constraints. Additionally, a long line of work has developed abstractions for GPU networking [64, 266, 151], which is orthogonal to FlexTOE and Laminar, and but could be integrated in future work.

6.3 Specialized Transport Co-designs

An alternative approach to improving performance and efficiency is to co-design applications with specialized transport protocols and hardware. Such tight application-protocol couplings, require developers to take transport constraints into account, such as weakened delivery semantics, complex buffer management, and constrained execution models. Hence, they lack generality, causing software reuse, management, and development agility to become a significant challenge. These approaches improve processing efficiency, but at the cost of requiring application re-design, all-or-nothing deployments, and operational issues at scale [95], often due to inflexibility [186, 278]. FlexTOE and Laminar instead offload the TCP protocol in a flexible manner, and remain broadly applicable and compatible with existing application and deployments. We survey these efforts below.

6.3.1 Software Stacks

eRPC [131] co-designs an RPC protocol and API with a kernel-bypass network stack to minimize CPU overhead per RPC. However, it adopts a constrained execution model, bundling threading model, buffer management, and relaxed transport semantics, requiring application re-design and limiting generality. Similarly, i10 [109] co-designs the storage stack with the in-kernel TCP stack to achieve CPU efficiency for remote storage access, providing no benefit for general applications.

6.3.2 RDMA Co-design

RDMA [56] is a popular combination of a networking API, protocol, and a (typically hardware) network stack. iWARP [238] builds RDMA atop TCP, offloading both. HERD [128] and Pilaf [183] co-design key-value stores with RDMA, while FaRM [72], DrTM+R [53], and DrTM+H [285] co-design distributed transactional systems for high performance and CPU efficiency. FaSST [130] further generalizes this approach with a two-sided RPC framework over RDMA. These co-designs achieve efficiency at the cost of generality. To improve compatibility, `rsocket` [8] offers a socket-like API over RDMA transport, but lacks full socket semantics (e.g., no `epoll` support) and cannot interoperate with standard file descriptors, limiting use in complex I/O applications. RedN [239] demonstrates RDMA's Turing completeness, showing that techniques like self-modifying operation chains and atomic verbs can express complex offloads (e.g., key-value lookups), though scalability remains constrained by limited NIC resources and slow atomic operations. Upper-layer primitives, like RDMA (e.g., iWARP), can also be implemented atop FlexTOE and Laminar in future work.

6.3.3 Co-design with In-network Accelerators

Numerous applications like key-value stores, distributed transactions systems, and consensus protocols have been co-designed with in-network accelerators [167, 125, 124, 251, 154, 281]. FlexNIC [136] demonstrates that co-designing key-value stores, stream processing, and intrusion detection systems with a programmable NIC that supports installable packet-processing rules can substantially improve both latency and throughput. To provide flexibility and reduce PCIe round trips, systems like Xenic [251] employ function shipping, moving application logic from the host server to coordinator-side SmartNICs. Such offloads may be flexibly layered atop FlexTOE and Laminar in future work. E3 [168] and iPipe [166] propose generic frameworks to offload entire applications to SmartNICs. Both these systems adopt modular programming models to decompose applications into microservices or actors, that can be independently offloaded to SmartNICs. LineFS [143] offloads a distributed file system to an off-path SmartNIC, leveraging pipeline parallelization to hide execution latencies of wimpy SmartNIC CPUs and data access across PCIe. We take inspiration from these designs to decompose the TCP data-path into fine-grained modules as a data-parallel pipeline for offload to NPU-SmartNICs in FlexTOE (§4).

Chapter 7

Generalizing Within and Beyond TCP

Flexibility is a core design goal of both FlexTOE and Laminar. A key aspect of this flexibility is the ability to express new transport protocols and features as they emerge, while remaining adaptable to evolving application needs. Over the past decade, transport protocols have diversified rapidly [184, 16, 99, 187], and numerous systems have shown the value of tightly integrating transport mechanisms with application semantics to optimize performance for specific workloads [137, 131].

In this chapter, we begin by outlining the limitations of our designs (§7.1), examining their implications in the context of TCP and discuss potential approaches to mitigate them. We then analyze the broader landscape of transport protocols and discuss how the design principles underlying FlexTOE and Laminar generalize beyond TCP to support a wide range of emerging transports (§7.2).

7.1 Limitations

Below, we highlight some limitations of our designs in the context of TCP, discuss potential approaches to address them. Many of these limitations can be addressed with further engineering effort, and do not reflect fundamental design constraints. We defer the discussion on more forward-looking work in Chapter 8.1.

Limited TCP Loss Recovery Mechanisms. Both FlexTOE and Laminar currently provide limited TCP Selective Acknowledgment (SACK) support, relying on 1-OOO interval reassembly and go-back-N retransmissions for loss recovery. In Laminar, we extend this to 1-OOO SACK-based recovery, which suffices for maintaining high performance under typical data center loss scenarios (§5.5). Tracking a small number of additional out-of-order intervals, or using packet bitmaps (sufficient in practice, cf. §5.5) and maintaining bitmaps like IRN [185] is relatively straightforward on NPUs (FlexTOE), albeit at the cost of additional on-NIC memory for storing state, and is more challenging but feasible with careful design on RMT (Laminar) due to strict state processing limits.

Out-of-Band Congestion Policy. Neither FlexTOE nor Laminar currently implement inline congestion policy, as such mechanisms are computationally intensive. Instead, both systems employ control-plane driven, out-of-band congestion policy, a practical approach widely adopted in large-scale data center deployments [268]. However, when the control plane is slow to react, this approach can lead to sluggish congestion response. Our prototypes exhibit this limitation due to the relatively high latency between the data-path and control plane; in contrast, several modern NPU and RMT-based NICs now provide low-latency custom interconnects between control CPUs and the data path [222, 85]. Encouragingly, emerging RMT hardware

now supports hardware-assisted mathematical operations — such as division and exponential moving averages [96] — that make inline implementations increasingly practical. In addition, approximate computation, table-based lookup of precomputed values, and fixed-point arithmetic offer viable techniques to further reduce complexity and enable efficient on-path congestion control. Traditional congestion control algorithms are themselves constrained by the feedback delay between sender and receiver, typically on the order of tens of microseconds even in low-latency data center networks [92]. Thus, while inline congestion control is not a panacea, it remains a promising direction for future work to reduce the CPU overhead of congestion control policies and improve responsiveness (cf. Receiver-driven Congestion Control §7.2).

Inability to Parse Message Boundaries. Flexible offloads for higher-level protocols, such as RPCs, often require parsing message boundaries, for example, to perform access-control checks and load balancing based on RPC methods. The stream orientation of TCP and lack of on-NIC buffering in our designs make this challenging, as message delimiters may be split across multiple TCP segments. Our current designs do not address this challenge directly, instead relying on the applications to respect message boundaries during segmentation. Message-orientation can be pushed down to the transport layer to address this limitation (cf. §7.2).

Head-of-Line (HoL) Blocking with libTOE-ZC. In libTOE-ZC, we currently lack mechanisms to mitigate HoL blocking when using zero-copy (ZC) APIs. Although applications can allocate multiple transmit buffers and issue out-of-order sends, segments are still transmitted in the order of allocation, which may occasionally delay later messages. In practice, however, this is rarely a major concern. Small RPCs — which gain little from Zero-Copy — can simply use copy-based APIs to avoid HoL blocking, while large throughput-oriented flows can mitigate it by maintaining multiple connections. More advanced queue-based interfaces, similar to

ibverbs, could further improve flexibility if needed, and can be integrated into both FlexTOE and Laminar designs.

Specialized Transmission Pacing Model. In FlexTOE, scheduling is *pull*-based: the data-path centrally schedules packets for all flows while enforcing congestion window and pacing constraints. This design increases latency for small flows, as libTOE must first notify the data-path of available data before transmission can occur. In contrast, Laminar adopts a *push*-based pacing model, where the data-path periodically grants transmission credits to libTOE, allowing immediate packet transmission when data becomes available. This reduces latency for small flows but adds processing overhead for large flows. The overhead can be mitigated by employing NIC hardware rate limiting or adopting a hybrid push-pull pacing scheme — both feasible extensions rather than fundamental design limitations.

7.2 Generalizability Beyond TCP

Over the past decade, numerous transport protocols [87, 17, 99, 187, 131] and congestion control mechanisms [92, 184, 16] have emerged to balance the diverse needs of modern data center applications, including low latency, high throughput, fairness, CPU overhead, reliability, and compatibility. This landscape continues to evolve rapidly, with no clear consensus on a single "best" transport protocol, if any. Hence, maintaining the flexibility to adapt transport stacks to new protocols and features as they arise is crucial. Both FlexTOE and Laminar are designed with this adaptability in mind. While both systems currently implement TCP, their underlying design principles extend to a wide range of transport protocols.

Our discussion aligns with the observation made by the authors of Tonic [23] — that many protocols share common structural patterns in transport logic. All reliable transports must

Protocol	Msg-based	Weak Rel. & Ord.	Multi-path	Conn-less	Recv-driven CC	Pkt. Priority
TCP [74]	✗	✗	✗	✗	✗	✗
UDP [7]	✓	✓	✓	✓	✗	✗
RoCE UC [5]	✓	✓	✗	✗	✗	✗
RoCE RC[1] [5]	✓	✗	✗	✗	✗	✓
RoCE UD [5]	✓	✓	✗	✓	✗	✗
SCTP [199]	✓	✓	✓	✗	✗	✗
pFabric [17]	✗	✗	✗	✗	✗	✓
pHost [87]	✗	✗	✗	✗	✓	✓
NDP [99]	✓	✗	✓	✓	✓	✓
Homa [187]	✓	✓	✓	✗	✓	✓
MPTCP [225]	✗	✗	✓	✗	✗	✗
Falcon [268]	✓	✗	✓	✗	✗	✗

Table 7.1: Comparison of modern data center transport protocols. Refer to §7.2 for definition of columns.

track in-flight segments for reassembly, detect and recover losses through retransmissions, and regulate in-flight segments to avoid congestion, though the mechanisms differ. Our designs illustrate how these shared patterns can be efficiently realized across diverse programmable network hardware platforms. Minor variations, such as packet header formats or feedback signaling, are easily accommodated on both architectures. Similarly, additional congestion signals can be collected in the data-path or by the control plane, facilitating implementation of most congestion control algorithms [296], while connection management operations (e.g., TLS Handshake) are purely done in control plane software. Offloads such as encryption, compression, and inline data transformations are orthogonal to the transport design and can be integrated into both systems using fixed-accelerators as needed.

Below, we focus on more substantive differences to the transport semantics from TCP across five key dimensions (Table 7.1), discussing the challenges they present in implementing them on FlexTOE and Laminar.

[1]+PFC,DCQCN [1, 76]

7.2.1 Message Orientation

The message abstraction aligns naturally with many RPC-oriented data center applications. Several modern protocols replace TCP's stream orientation with explicit message boundaries, simplifying application framing, enabling out-of-order delivery, supporting prioritization and congestion control, and facilitating in-network computation [123]. Our arguments below focus on the receiver side; similar reasoning applies to the transmit side.

Tracking In-Flight Messages. This shift does not fundamentally change reassembly or retransmission — messages are simply tracked using packet sequence numbers (PSN) rather than byte offsets. Accordingly, the OOO intervals in FlexTOE and Laminar map directly to ranges in PSN space.

Direct Data Placement and On-NIC Buffering. Message-oriented transports can also avoid on-NIC buffering by decoupling data placement from application notification. Typically, each message either carries a host buffer address for direct placement or the host provides a queue of pre-allocated buffers for incoming messages to be satisfied in order (cf. RDMA WRITE vs. SEND). For a pre-determined in-flight message window, both FlexTOE and Laminar can efficiently index these queues for address translation during reassembly and perform direct data placement into host memory. When the receive window advances, the base index into these queues can be updated or a bitmap can be bit shifted. If the transport requires in-order delivery semantics, the data-path can notify libTOE to deliver messages to the application only when all prior messages have been received. Address translation does not introduce state dependencies and can be efficiently parallelized across RMT pipeline stages or NPU modules.

State Requirements. Finally, the state requirements for message-oriented transports are comparable to TCP — under typical RPC workloads, the number of in-flight messages roughly matches the number of active TCP connections [187, 224]. Our designs eliminate per-segment state and on-NIC segment buffering. Hence, both message-oriented and stream-oriented transports scale similarly with the number of active flows, though the proportionality constants may differ.

7.2.2 Weaker Reliability and Ordering Semantics

Some protocols relax strict reliability and ordering guarantees of TCP to improve latency and flexibility, instead providing *weaker* semantics such as “best effort”, “at-least-once” or “out-of-order (OOO)” delivery.

Weak Reliability. Protocols with weak reliability semantics may not require retransmissions for lost packets, instead relying on higher-layer mechanisms for recovery. Both FlexTOE and Laminar can accommodate this by simply omitting retransmission logic from the data-path. The data-path can still track received packets for reassembly and notify libTOE of missing packets for higher-layer recovery. Protocols that allow “at-least-once” delivery can forgo duplicate detection logic. This potentially reduces on-NIC state and processing overhead, improving performance.

Out-of-Order Delivery. Message reassembly, data placement, and application notification can all be decoupled, and remain independently programmable in both FlexTOE and Laminar. On receiving a complete message or segment (in-order or OOO), detected programmatically, the data-path can notify libTOE to deliver it to the application immediately. The missing segments can continue to be tracked for reassembly and loss recovery, depending on the protocol’s reliability semantics.

7.2.3 Multi-pathing

Multi-pathing or packet spraying is a congestion spreading technique by leveraging multiple network paths between endpoints. Both FlexTOE and Laminar could implement multi-pathing by independently routing packets through the network fabric — either randomly or using flowlet-based approaches [46, 268]. This, however, may introduce packet reordering, requiring robust out-of-order (OOO) handling.

Robust OOO Handling. Our systems already include basic OOO handling mechanisms, primarily designed to mitigate occasional packet loss and reordering in data center networks. To support more aggressive multi-pathing, these mechanisms can be enhanced to better handle frequent reordering. Tracking a small number of additional out-of-order intervals and maintaining bitmaps like IRN [185] is feasible with careful design. However, more sophisticated mechanisms — like TCP Recent Acknowledgment (RACK) [121] — that require tracking per-packet transmission times and performing $O(\text{window})$ scans over unacknowledged packets are infeasible on RMT (Laminar) due to strict state processing limits, but are relatively straightforward on NPUs (FlexTOE). Short RPCs rarely require robust OOO handling since they typically fit within a single packet. In data center settings, only a small number of large, throughput-oriented flows are active per host. Consequently, robust OOO handling can be selectively enabled for these flows to conserve on-NIC resources. Such large flows also exhibit low packet rates, making it feasible to use slower RMT pipeline implementations that provide more powerful per-stage processing.

7.2.4 Connection-less Operation

Connection-less transport protocols eliminate high-latency handshakes by embedding the necessary state within the first packet of a message, or completely do away with connection state by including all necessary information in each packet.

On-the-Fly State Initialization. RMT-pipelines (Laminar) can extract and set up necessary connection state directly from packet headers using register arrays. Their inherently sequential processing model provides a natural synchronization point for initializing state from the first packet of a message. Curiously, NPUs (FlexTOE) face greater difficulty due to synchronization challenges among parallel processing units, where subsequent packets may depend on state set up by prior packets. Steering packets to specific NPU cores via hash-based flow affinity mitigates this issue but restricts each flow's scalability to a single core.

7.2.5 Receiver-driven Congestion Control

Several modern transport protocols employ receiver-driven congestion control, where the receiver regulates the sender's transmission rate by explicitly granting credits, often to optimize metrics such as RPC completion time. For example, Homa [187] uses receiver-driven scheduling to prioritize small messages, scheduling messages in shortest remaining processing time (SRPT) order.

Flexible Scheduling Policies. Supporting such mechanisms requires implementing flexible scheduling policies on the NIC. FlexTOE already includes a software flow scheduler module that maintains a priority queue for transmit scheduling, which can be extended to support receiver-driven schemes. On Laminar, the packet generator and register arrays can be composed to

implement FIFO or calendar queues. Implementing priority queues on RMT (Laminar) is more challenging due to limited processing capabilities; However, recent work demonstrates effective approximations of priority queue scheduling algorithms in RMT pipelines [256, 258, 257, 269] that can be adapted for this purpose. Alternatively, FPGA-based designs can synthesize these more sophisticated, flexible scheduling primitives directly in hardware, integrated with the RMT data-path [24].

7.2.6 Packet Prioritization and Trimming

Some transport protocols explicitly prioritize certain packets (e.g., control packets) over data packets to improve responsiveness, or trim packets under congestion to signal senders to reduce load [99]. These features are already compatible with fixed-function priority scheduler queues, packet truncation, and marking mechanisms available on modern network accelerators.

7.2.7 Summary

Table 7.1 summarizes how various transport protocols differ across key dimensions. While these extensions have not yet been implemented, we believe they can be realized with our existing designs based on our analysis above. Note that the underlying hardware platforms offer differing programming capabilities; some extensions may be more straightforward on one platform than the other.

Chapter 8

Conclusion

Data center end-host network stacks are undergoing a profound transformation driven by immense pressure from emerging workloads. Traditional CPU-driven software stacks suffer from poor performance and unsustainable overheads, driving operators towards alternatives. Consequently, highly efficient ASIC-based transports like RDMA are seeing increased adoption, despite challenges stemming from their inflexibility. This dissertation addresses the central question: *Can we build data center transport stacks that retain the flexibility of software while delivering the performance and energy-efficiency of hardware offload?*?

Leveraging hardware architectures optimized for flexible packet processing, we design two TCP stacks — FlexTOE and Laminar, which explore distinct points in this design space. Our main design insight is that scaling data center TCP and adapting it to network accelerators requires fine-grained parallelization of the transport logic to overcome the constraints of highly parallel, resource-limited hardware. Both systems dramatically improve performance and efficiency over CPU-based software stacks approaching ASIC-levels, retain application compatibility,

interoperate with existing TCP stacks and network deployments, and remain robust to packet loss and congestion. Crucially, they also enable agile modifications via high-level software programmability.

These stack designs generalize to a broad class of hardware architectures and transport protocols. We provide full-system implementations on multiple hardware targets, which we open-source to facilitate future research and development, and we extensively evaluate them across diverse, realistic workloads. Collectively, these contributions rigorously validate the thesis that high-performance, energy efficiency, and flexibility can be meaningfully balanced in practical data center transport stacks.

8.1 Future Directions

In this section, we outline several promising avenues for future research building on the ideas presented in this dissertation, and speculate on how they may evolve over time.

8.1.1 Programmable Architecture Search

Beyond Networking. The broader "programmability vs. efficiency" trade-off is universal; it extends beyond networking to domains such as storage, accelerators, and memory systems. Designs for Computational SSDs [29], processing-in-memory architectures [265, 198], and interconnects like NVLink (cf. SHARP [207, 93]) and CXL-based memory controllers [35] face similar challenges. The techniques developed in this dissertation for performing stateful computation and managing placement under tight resource constraints provide valuable case-studies for both hardware and software design in these domains. Looking ahead, devices tightly coupled through coherent interfaces such as CXL and UCIe must process enormous volumes of

cache-coherency messages at extremely low latency to avoid stalling host processors. Effective offloads on these devices will inevitably require maintaining state, making our design principles particularly relevant.

Beyond RMT: Relaxing Constraints for Greater Flexibility. A closely related research direction is relaxing the rigid constraints of existing RMT architectures to enhance programmability [180, 55, 79, 12, 231, 159, 287, 291, 282, 263, 289, 89, 54, 110, 150]. A notable example in this space is dRMT [55], which moves away from stage-local state towards more flexible memories. The RMT programming model simplifies consistency — but at the cost of expressivity. Allowing multiple stages to read and write shared state could unlock richer classes of applications, though it introduces synchronization challenges and non-deterministic performance due to pipeline stalls.

8.1.2 Towards Disaggregated Deployments

Scaling Disaggregation Fabrics. Emerging disaggregation technologies such as CXL are poised to reshape data center architectures by enabling flexible pooling of memory and I/O device resources across servers [165]. However, as CXL fabrics scale in range and complexity, they will inevitably encounter classical networking challenges of topology, routing, reliability, and congestion — all of which may benefit from broader networking literature and the techniques developed in this dissertation. Moreover, new interactions and congestion dynamics may arise at the boundary between the CXL traffic and traditional Ethernet networks, necessitating novel transport adaptations that span both domains.

Network Stack Disaggregation. While not a primary focus of this dissertation, Laminar's switch-based implementation naturally enables network stack disaggregation, addressing re-

source underutilization in modern cloud environments where each host provisions its own network stack instance. Prior work has shown the benefits of disaggregation within a single host across VMs [275]. A disaggregated stack like Laminar can extend these advantages across multiple hosts at the rack level or higher in the network topology, improving resource efficiency while preserving isolation. Moreover, the deterministic performance of programmable switches ensures predictable behavior even in multi-tenant settings. In the near term, CXL-pooled or multi-headed programmable NICs could further enable stack disaggregation within a rack.

Programmable Switches as Transport Gateways. A related challenge arises with TCP proxies in data centers, commonly deployed to interface internal networks with the public Internet [100]. A similar scenario exists in large-scale ML training clusters, where GPUs connect via a custom fabric or a siloed RDMA deployment (the *back-end network*), yet must exchange data with the broader data center over commodity transports like TCP (the *front-end network*) [86, 268, 264, 234]. High throughput in the front-end network is critical to efficiently pull training data from storage systems and keep GPUs well utilized. In both cases, Laminar running on a programmable switch could serve as an efficient gateway, translating between TCP and the internal transport protocol while offloading TCP processing from end hosts. However, Internet-facing TCP traffic differs markedly from data center workloads — featuring higher connection counts, lower per-connection throughput, variable RTTs, and greater packet loss. Adapting offloaded TCP stacks to efficiently support such traffic presents an interesting direction for future work.

SmartNIC-hosted Disaggregation Systems. Our stacks also enable the broader vision of SmartNIC-hosted disaggregation — where the server CPU is replaced by a headless SmartNIC driving I/O devices and accelerators [226, 261], obtaining cost and energy savings. These clusters inevitably require efficient transport protocols on the SmartNIC itself, driven by wimpy

on-NIC processors, making our contributions particularly relevant. Our designs may be integrated directly on these increasingly heterogeneous SmartNICs.

8.1.3 Towards Higher-level Application Interfaces

Specialized Interfaces. Flexible transport offload creates new opportunities to rethink the interface between applications and the transport layer. Traditional socket APIs, centered around byte streams, sometimes mismatch the semantics and performance needs of modern applications. To begin with, higher-level one-sided RDMA primitives, such as RDMA reads and writes, may be layered atop FlexTOE and Laminar (cf. iWARP [238]). Going further, flexible offload can extend one-sided communication primitives into higher-level operations like pointer chasing, and data structure traversal [122]. The shared log acceleration demonstrated in Laminar (§5.5) exemplifies how specialized interfaces and transport integration enable richer, more application-aware operations and efficiency.

Opportunities with Cache-Coherent Interconnects. Emerging cache-coherent interconnects such as CXL further strengthen this vision, enabling fine-grained memory operations that tightly couple transport mechanisms with application data structures and message serialization. Flexibly processing both cache-coherency protocol messages and network packets directly on NICs could unlock new levels of performance and efficiency for distributed applications [122, 288, 244], taking us one step closer to the ideal of distributed shared memory, as envisioned in early systems like Scale-out NUMA [202].

8.1.4 Beyond Transport: Offloading Broader RPC Operations

The "data center tax" imposed by networking extends well beyond transport protocols, encompassing overheads from RPC serialization and deserialization, load balancing, virtualization, access control, routing, proxying, rate limiting, and monitoring [301]. In many cases, the overheads from these auxiliary services rival or even exceed those of the transport layer itself [301, 252].

Flexible Offload for RPC Extensions. These extensions also demand the greatest flexibility and customization, as they evolve rapidly alongside the applications they serve — making them prime candidates for offload onto programmable hardware. Some of these features have already been partially offloaded onto programmable NICs and switches [18, 233, 62, 152, 293, 229], and could be integrated with our transport designs, but much remains to be explored. For instance, prior work on offloading RPC serialization and deserialization, often relies on custom hardware [135] or introduces specialized serialization formats and APIs that sacrifice compatibility with existing applications [286, 235]. Moreover, (de-)serialization remains serialized with respect to transport and application processing, and often incurs copies, adding latency. Enabling fine-grained overlap between (de-)serialization, transport, and application processing to minimize end-to-end latency remains an open and promising research challenge.

8.1.5 Networking for ML Workloads

Data center networking is undergoing a massive transformation driven by the rise of large-scale machine learning (ML) workloads. Interestingly, the tremendous growth in GPU and accelerator performance, has once again shifted the bottleneck back to networks [11].

New Traffic Patterns. Machine learning workloads are distinct from traditional data center traffic patterns in several ways — they are highly bandwidth-intensive, involve few connections with low entropy, often involve predictable communication patterns (e.g., all-reduce in distributed training), require tight integration with GPU execution models, and can tolerate lost or partial data [284]. These characteristics invalidate many long-standing assumptions underlying today's Clos network topologies optimized for random any-to-any traffic, conventional congestion control strategies, and reliable, in-order transport protocols [86, 264]. Consequently, large-scale ML training clusters increasingly rely on custom network fabrics or siloed deployments with specialized topologies and transport designs [234, 86, 302]. Naturally, such workloads compel a fundamental rethinking of network stack design.

Multi-pathing. Recent trends in ML networking emphasize aggressive use of multi-pathing, or packet spraying, embracing dynamic, fine-grained path diversity to maximize throughput and link utilization, mitigate congestion, and improve load balance under low connection entropy [268, 86, 264, 255]. Supporting this efficiently in our stacks, requires robust out-of-order handling. But, since the connection counts are low, and the transfer sizes are large, the packet rates and resource demands remain manageable to implement this effectively in programmable hardware, though design changes to FlexTOE and Laminar would be necessary (§7.2.3).

Reconfigurable Optical Networks. Interestingly, a seemingly opposite trend is also emerging — the move toward circuit-switched optical networks [11] built with reconfigurable optical switches [302], creating high-bandwidth straight-line paths between communicating endpoints. Predictable, bandwidth-intensive communication patterns in ML workloads make them ideal candidates for such networks. Hence, the design focus shifts towards optimal scheduling

strategies and re-configuration mechanisms for resource allocation [25, 139, 283]. While these networks promise to obviate the need for traditional transport protocols by provisioning direct paths between endpoints, this promise rarely holds in practice. Reliability remains a fundamental concern — packet loss and congestion can occur during reconfiguration, where transient data may need buffering [176, 177], requiring transport layer support [65].

GPU Networking. Another promising direction is to adapt the network stack itself better towards GPU execution models. Today's GPU clusters still rely on CPUs to drive (offloaded) network stacks, introducing significant overheads and latency [61]. Limited integration — where NICs can directly access GPU memory via mechanisms like GPUDirect RDMA [151] — into Laminar and FlexTOE is relatively straightforward. However, GPU-native networking APIs represent an emerging research frontier [64, 61], aiming to align collective communication primitives with the GPU's massively parallel execution model and to schedule them effectively across heterogeneous interconnects such as NVLink, PCIe, and RDMA [253, 298, 297, 105, 264].

8.2 Concluding Remarks

Transport offload is an inevitable reality — the question is not *if* and *when*, but *how*. This dissertation takes major steps towards that future, showing that flexible, programmable hardware can deliver high performance and efficiency for data center transport stacks. This work tackles the inherent difficulties of adapting stateful protocols to restrictive programmable hardware. Our design principles generalize to accelerating a broad range of networked systems and protocols, influence how next-generation hardware is built, and also guide the evolution of efficient transport protocols tailored for data center hardware.

This dissertation was written amidst what appears to be a paradigm shift in data center in-

frastructure, precipitated by emerging workloads, heterogeneous hardware, disaggregated architectures, and rising power constraints — forces that have prompted a record-breaking investment surge by hyperscale operators [106]. Taken together, these forces are reshaping priorities and design trade-offs, and exposing the limits of long-standing networking assumptions. They call for a fundamental rethinking of transport protocol design in modern data centers, challenging traditional notions of reliability, ordering, congestion control, and even the role of the transport layer itself. While some of these changes can be absorbed within the flexible frameworks of FlexTOE and Laminar, others may ultimately require clean-slate re-designs. The ideas and techniques presented in this dissertation provide a strong foundation for navigating this "uncomfortably exciting" [1] future.

[1] A term borrowed from Parthasarthy Ranganathan, who in turn borrowed it from Google's founders [236].

Bibliography

[1] 802.1Qbb: Priority-based Flow Control. https://1.ieee802.org/dcb/802-1qbb/. [Accessed 02-11-2025].

[2] aio(7) - Linux manual page. https://man7.org/linux/man-pages/man7/aio.7.html. [Accessed 03-11-2025].

[3] epoll(7) - Linux manual page. https://man7.org/linux/man-pages/man7/epoll.7.html. [Accessed 03-11-2025].

[4] How we built it: block storage for AI/ML workloads, powered by Lightbits — crusoe.ai. https://www.crusoe.ai/resources/blog/how-we-built-it-block-storage-for-ai-ml-workloads-powered-by-lightbits. [Accessed 18-10-2025].

[5] InfiniBand Specification. https://www.infinibandta.org/ibta-specification/. [Accessed 02-11-2025].

[6] networking:i:oat [Wiki] — wiki.linuxfoundation.org. https://wiki.linuxfoundation.org/networking/i/oat. [Accessed 21-10-2025].

[7] RFC 768: User Datagram Protocol. https://datatracker.ietf.org/doc/html/rfc768. [Accessed 02-11-2025].

[8] rsocket(7) - Linux manual page. https://linux.die.net/man/7/rsocket. [Accessed 03-11-2025].

[9] sendfile(2) - Linux manual page. https://man7.org/linux/man-pages/man2/sendfile.2.html. [Accessed 03-11-2025].

[10] socket(7) - Linux manual page — man7.org. https://man7.org/linux/man-pages/man7/socket.7.html. [Accessed 21-10-2025].

[11] a16z. Building the Real-World infrastructure for AI, with Google, Cisco, and a16z. https://www.youtube.com/watch?v=OsLRf6r5U9E, 2025. [Accessed 02-11-2025].

[12] Artem Ageev, M Foroushani, and Antoine Kaufmann. Exploring domain-specific architectures for network protocol processing. In *Proc. Cloud@ MICRO Virtual Workshop*, 2021.

[13] Anurag Agrawal and Changhoon Kim. Intel Tofino2–a 12.9 tbps P4-programmable ethernet switch. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–32. IEEE Computer Society, 2020.

[14] Marcos K Aguilera, Emmanuel Amaro, Nadav Amit, Erika Hunhoff, Anil Yelam, and Gerd Zellweger. Memory disaggregation: Why now and what are the challenges. *ACM SIGOPS Operating Systems Review*, 57(1):38–46, 2023.

[15] Hasan Al Maruf and Mosharaf Chowdhury. Memory disaggregation: advances and open challenges. *ACM SIGOPS Operating Systems Review*, 57(1):29–37, 2023.

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

[17] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review*, 43(4):435–446, 2013.

[18] Amazon. Lightweight Hypervisor - AWS Nitro System. https://aws.amazon.com/ec2/nitro/. [Accessed 03-11-2025].

[19] AMD. Vitis Networking P4. https://www.amd.com/en/products/adaptive-socs-and-fpgas/intellectual-property/ef-di-vitisnetp4.html.

[20] AMD. AMD Alveo U200 and U250 Data Center Accelerator Cards Data Sheet. https://docs.amd.com/r/en-US/ds962-u200-u250/Alveo-Product-Details, 2023.

[21] AMD. AMD Pensando 2nd generation ("Elba") Data Processing Unit. https://www.amd.com/content/dam/amd/en/documents/pensando-technical-docs/product-briefs/pensando-elba-product-brief.pdf, 2024.

[22] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 1–14, 2009.

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

[24] Serhat Arslan, Stephen Ibanez, Alex Mallery, Changhoon Kim, and Nick McKeown. NanoTransport: A low-latency, programmable transport layer for NICs. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 13–26, 2021.

[25] Rukshani Athapathu and George Porter. Reconfigurability within collective communication algorithms. In *Proceedings of the 2nd Workshop on Networks for AI Computing*, pages 43–49, 2025.

[26] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, et al. Empowering Azure storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, 2023.

[27] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Virtual consensus in Delos. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 617–632. USENIX Association, November 2020.

[28] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. CORFU: A distributed shared log. *ACM Trans. Comput. Syst.*, 31(4), December 2013.

[29] Antonio Barbalace and Jaeyoung Do. Computational storage: Where are we today? In *Conference on Innovative Data Systems Research 2020*, 2021.

[30] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. Pathways: Asynchronous distributed dataflow for ML. *Proceedings of Machine Learning and Systems*, 4:430–449, 2022.

[31] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. *The Datacenter as a Computer: Designing Warehouse-scale Machines*. Springer Nature, 2019.

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

[33] 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*, pages 267–280, 2010.

[34] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site reliability engineering: how Google runs production systems*. " O'Reilly Media, Inc.", 2016.

[35] David Boles, Daniel Waddington, and David A. Roberts. CXL-Enabled enhanced memory functions. *IEEE Micro*, 43(2):58–65, 2023.

[36] David Borman, Robert T. Braden, Van Jacobson, and Richard Scheffenegger. TCP Extensions for High Performance. RFC 7323, September 2014.

[37] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[38] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 99–110, New York, NY, USA, 2013. Association for Computing Machinery.

[39] Robert T. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122, October 1989.

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

[41] Broadcom. Trident4 / BCM56880 series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series, 2023.

[42] Broadcom. Trident5 / BCM78800 series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm78800, 2025.

[43] Mihai Budiu and Chris Dodd. The P416 programming language. *ACM SIGOPS Operating Systems Review*, 51(1):5–14, 2017.

[44] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 65–77, 2021.

[45] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards μs tail latency and terabit ethernet: disaggregating the host network stack. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 767–779, 2022.

[46] Peirui Cao, Wenxue Cheng, Shizhen Zhao, and Yongqiang Xiong. Network load balancing with parallel flowlets for AI training clusters. In *Proceedings of the 2024 SIGCOMM Workshop on Networks for AI Computing*, NAIC '24, page 18–25, New York, NY, USA, 2024. Association for Computing Machinery.

[47] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based congestion control. *Communications of the ACM*, 60(2):58–66, 2017.

[48] Cavium. Cavium LiquidIO SmartNICs. https://cavium.com/pdfFiles/LiquidIO_II_CN78XX_Product_Brief-Rev1.pdf, 2018.

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

[50] Sourav Chakraborty, Shulei Xu, Hari Subramoni, and Dhabaleswar Panda. Designing scalable and high-performance MPI libraries on Amazon Elastic Fabric Adapter. In *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 40–44, 2019.

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

[52] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, and Zeke Wang. Demystifying datapath accelerator enhanced off-path SmartNIC, 2024.

[53] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–17, 2016.

[54] Zhikang Chen, Yong Feng, Shuxin Liu, Haoyu Song, Hanyi Zhou, Tong Yun, Wenquan Xu, Tian Pan, and Bin Liu. OptimusPrime: Unleash dataplane programmability through a transformable architecture. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 904–920, New York, NY, USA, 2024. Association for Computing Machinery.

[55] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. dRMT: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 1–14, New York, NY, USA, 2017. Association for Computing Machinery.

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

[57] The P4 Language Consortium. P4-16 langauge specification. https://p4.org/p4-spec/docs/P4-16-v1.2.2.html, 2021.

[58] Intel Corporation. Intel Infrastructure Processing Unit (Intel IPU) ASIC E2000. https://www.intel.com/content/www/us/en/products/details/network-io/ipu/e2000-asic.html, 2021.

[59] Intel Corporation. Intel tofino 2. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino-2.html, 2023.

[60] NVIDIA Corporation. NVIDIA MLNX_OFED Documentation Rev 5.3-1.0.0.1. https://docs.nvidia.com/networking/display/mlnxofedv531001. [Accessed 02-11-2025].

[61] NVIDIA Corporation. NVSHMEM. https://developer.nvidia.com/nvshmem. [Accessed 02-11-2025].

[62] Tianyi Cui, Wei Zhang, Kaiyuan Zhang, and Arvind Krishnamurthy. Offloading load balancers onto SmartNICs. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '21, page 56–62, New York, NY, USA, 2021. Association for Computing Machinery.

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

[64] Feras Daoud, Amir Watad, and Mark Silberstein. GPUrdma: GPU-side library for high performance networking from GPU kernels. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '16, New York, NY, USA, 2016. Association for Computing Machinery.

[65] Federico De Marchi, Wei Bai, Jialong Li, and Yiting Xia. Rethinking transport protocols for reconfigurable data centers: An empirical study. In *Proceedings of the 1st SIGCOMM Workshop on Hot Topics in Optical Technologies and Applications in Networking*, HotOptics '24, page 7–13, New York, NY, USA, 2024. Association for Computing Machinery.

[66] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[67] Abhishek Dhamija, Balasubramanian Madhavan, Hechao Li, Jie Meng, Shrikrishna Khare, Madhavi Rao, Lawrence Brakmo, Neil Spring, Prashanth Kannan, Srikanth Sundaresan, and Soudeh Ghorbani. A large-scale deployment of DCTCP. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 239–252, Santa Clara, CA, April 2024. USENIX Association.

[68] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 325–338, Santa Clara, CA, February 2020. USENIX Association.

[69] Linux Networking Documentation. MSG_ZEROCOPY. https://docs.kernel.org/networking/msg_zerocopy.html, 2025.

[70] Linux Networking Documentation. Scaling in the Linux networking stack. https://www.kernel.org/doc/html/v6.14/networking/scaling.html, 2025.

[71] Linux Networking Documentation. Segmentation offloads in the Linux networking stack. https://www.kernel.org/doc/Documentation/networking/segmentation-offloads.txt, 2025.

[72] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[73] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, pages 195–216, 2017.

[74] Wesley Eddy. Transmission Control Protocol (TCP). RFC 9293, August 2022.

[75] Khaled Elmeleegy, Anupam Chanda, Alan L Cox, and Willy Zwaenepoel. Lazy asynchronous I/O for Event-Driven servers. In *USENIX Annual Technical Conference, General Track*, pages 241–254, 2004.

[76] Roni Even. Data Center Congestion Management requirements. https://datatracker.ietf.org/doc/html/draft-yueven-tsvwg-dccm-requirements-00. [Accessed 02-11-2025].

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

[78] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. PacketMill: toward per-core 100-gbps networking. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 1–17, New York, NY, USA, 2021. Association for Computing Machinery.

[79] Yong Feng, Zhikang Chen, Haoyu Song, Yinchao Zhang, Hanyi Zhou, Ruoyu Sun, Wenkuo Dong, Peng Lu, Shuxin Liu, Chuwen Zhang, Yang Xu, and Bin Liu. Empower programmable pipeline for advanced stateful packet processing. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 491–508, Santa Clara, CA, April 2024. USENIX Association.

[80] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public Cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI '18, pages 51–64, USA, 2018. USENIX Association.

[81] Mario Flajslik and Mendel Rosenblum. Network interface design for low latency request-response protocols. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, page 333–346, USA, 2013. USENIX Association.

[82] Linux Foundation. Data plane development kit (DPDK). https://www.dpdk.org/, 2015.

[83] Linux Foundation. Storage performance development kit (SPDK). http://www.spdk.io, 2024.

[84] Open Networking Foundation. ONF connect 18 keynote: Amin vahdat, engineering fellow and VP, Google. https://www.youtube.com/watch?v=KCTH8vG_GNo, 2019. [Accessed 18-10-2025].

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

[86] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. RDMA over Ethernet for distributed training at Meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 57–70, New York, NY, USA, 2024. Association for Computing Machinery.

[87] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. pHost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1–12, 2015.

[88] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533, 2021.

[89] Nadeen Gebara, Alberto Lerner, Mingran Yang, Minlan Yu, Paolo Costa, and Manya Ghobadi. Challenging the stateless quo of programmable switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 153–159, New York, NY, USA, 2020. Association for Computing Machinery.

[90] Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Q. Maguire Jr., and Dejan Kostić. Packet Order Matters! improving application performance by deliberately delaying packets. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 807–827, Renton, WA, April 2022. USENIX Association.

[91] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan MG Wassel, Zhehua Wu, Sunghwan Yoo, et al. Aquila: A unified, low-latency fabric for datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1249–1266, 2022.

[92] Prateesh Goyal, Preey Shah, Naveen Kr. Sharma, Mohammad Alizadeh, and Thomas E. Anderson. Backpressure flow control. In *Proceedings of the 2019 Workshop on Buffer Sizing*, BS '19, New York, NY, USA, 2020. Association for Computing Machinery.

[93] Richard L. Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, Lion Levi, Alex Margolin, Tamir Ronen, Alexander Shpiner, Oded Wertheim, and Eitan Zahavi. Scalable hierarchical aggregation protocol (SHArP): A hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 1–10, 2016.

[94] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.

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

[96] Vladimir Gurevich and Andy Fingerhut. P416 programming for Intel Tofino using Intel P4 studio. https://opennetworking.org/wp-content/uploads/2021/05/2021-P4-WS-Vladimir-Gurevich-Slides.pdf, 2021.

[97] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 40(4):195–206, 2010.

[98] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A new programming interface for scalable network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, pages 135–148, USA, 2012. USENIX Association.

[99] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 29–42, 2017.

[100] Yutaro Hayakawa, Lars Eggert, Michio Honda, and Douglas Santry. Prism: a proxy architecture for datacenter networks. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 181–188, New York, NY, USA, 2017. Association for Computing Machinery.

[101] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[102] Torsten Hoefler, Duncan Roweth, Keith Underwood, Bob Alverson, Mark Griswold, Vahid Tabatabaee, Mohan Kalkunte, Surendra Anubolu, Siyuan Shen, Abdul Kabbani, et al. Datacenter Ethernet and RDMA: Issues at hyperscale. *arXiv preprint arXiv:2302.03337*, 2023.

[103] Torsten Hoefler, Karen Schramm, Eric Spada, Keith Underwood, Cedell Alexander, Bob Alverson, Paul Bottorff, Adrian Caulfield, Mark Handley, Cathy Huang, et al. Ultra Ethernet's design principles and architectural innovations. *arXiv preprint arXiv:2508.08906*, 2025.

[104] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, et al. Characterization of large language model development in the datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 709–729, 2024.

[105] Zhiyi Hu, Siyuan Shen, Tommaso Bonato, Sylvain Jeaugey, Cedell Alexander, Eric Spada, James Dinan, Jeff Hammond, and Torsten Hoefler. Demystifying NCCL: An in-depth analysis of GPU communication protocols and algorithms. *arXiv preprint arXiv:2507.04786*, 2025.

[106] Stanford University Human-Centered Artificial Intelligence. Artificial Intelligence Index Report 2025. https://hai.stanford.edu/assets/files/hai_ai_index_report_2025.pdf, 2025. [Accessed 02-11-2025].

[107] Jack Tigar Humphries, Neel Natu, Kostis Kaffes, Stanko Novaković, Paul Turner, Henry M. Levy, David Culler, and Christos Kozyrakis. Wave: Offloading resource management to SmartNIC cores. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '25, page 264–281, New York, NY, USA, 2025. Association for Computing Machinery.

[108] Darby Huye, Yuri Shkuro, and Raja R Sambasivan. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 419–432, 2023.

[109] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP ≈ RDMA: CPU-efficient remote storage access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 127–140, Santa Clara, CA, February 2020. USENIX Association.

[110] Stephen Ibanez, Gianni Antichi, Gordon Brebner, and Nick McKeown. Event-Driven packet processing. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 133–140, New York, NY, USA, 2019. Association for Computing Machinery.

[111] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The NanoPU: A nanosecond network stack for datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 239–256, 2021.

[112] Intel. Accelerating High-Speed networking with Intel i/o acceleration technology. https://www.intel.com/content/dam/doc/white-paper/i-o-acceleration-technology-paper.pdf, 2006.

[113] Intel. Intel data direct i/o technology (Intel DDIO): A primer. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf, 2012.

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

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

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

[117] iperf. iPerf3, 2025. https://iperf.fr/iperf-download.php.

[118] Jakub Kicinski and Nicolaas Viljoen, Netronome Systems. eBPF hardware offload to SmartNICs: cls bpf and XDP. https://www.netronome.com/media/documents/eBPF_HW_OFFLOAD_HNiMne8_2_.pdf, 2021.

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

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

[121] Priyaranjan Jha. RFC 8985: The RACK-TLP loss detection algorithm for TCP. https://datatracker.ietf.org/doc/rfc8985/. [Accessed 02-11-2025].

[122] Houxiang Ji, Yifan Yuan, Yang Zhou, Ipoom Jeong, Ren Wang, Saksham Agarwal, and Nam Sung Kim. Re-architecting end-host networking with CXL: Coherence, memory, and offloading. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture®*, pages 1809–1823, 2025.

[123] Tao Ji, Rohan Vardekar, Balajee Vamanan, Brent E Stephens, and Aditya Akella. MTP: Transport for In-Network computing. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 959–977, 2025.

[124] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, 2018.

[125] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th symposium on operating systems principles*, pages 121–136, 2017.

[126] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A Berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[127] Raj Joshi, Cha Hwan Song, Xin Zhe Khooi, Nishant Budhdev, Ayush Mishra, Mun Choon Chan, and Ben Leong. Masking corruption packet losses in Datacenter networks with link-local retransmission. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 288–304, New York, NY, USA, 2023. Association for Computing Machinery.

[128] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. *SIGCOMM Comput. Commun. Rev.*, 44(4):295–306, August 2014.

[129] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX annual technical conference (USENIX ATC 16)*, pages 437–450, 2016.

[130] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, scalable and simple distributed transactions with Two-Sided(RDMA) datagram RPCs. In *12th USENIX*

Symposium on Operating Systems Design and Implementation (OSDI 16), pages 185–201, 2016.

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

[132] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G Andersen. Raising the bar for using GPUs in software packet processing. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 409–423, 2015.

[133] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture,* ISCA '15, page 158–169, New York, NY, USA, 2015. Association for Computing Machinery.

[134] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *CoRR*, abs/2001.08361, 2020.

[135] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. A hardware accelerator for protocol buffers. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 462–478, 2021.

[136] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas 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, pages 67–81, New York, NY, USA, 2016. Association for Computing Machinery.

[137] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with FlexNIC. *SIGARCH Comput. Archit. News*, 44(2):67–81, March 2016.

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

[139] Mehrdad Khani, Manya Ghobadi, Mohammad Alizadeh, Ziyi Zhu, Madeleine Glick, Keren Bergman, Amin Vahdat, Benjamin Klenk, and Eiman Ebrahimi. SiP-ML: high-bandwidth

optical network interconnects for machine learning training. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 657–675, 2021.

[140] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. TEA: Enabling State-Intensive network functions on programmable switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 90–106, New York, NY, USA, 2020. Association for Computing Machinery.

[141] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. RedPlane: enabling fault-tolerant stateful in-switch applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 223–244, New York, NY, USA, 2021. Association for Computing Machinery.

[142] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan. Generic External Memory for switch data planes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, page 1–7, New York, NY, USA, 2018. Association for Computing Machinery.

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

[144] Taehyun Kim, Deondre Martin Ng, Junzhi Gong, Youngjin Kwon, Minlan Yu, and KyoungSoo Park. Rearchitecting the TCP stack for I/O-Offloaded content delivery. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 275–292, Boston, MA, April 2023. USENIX Association.

[145] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs First-Class Datacenter citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference*, USENIX ATC '19, pages 863–879, USA, 2019. USENIX Association.

[146] Sakari Lahti and Timo D. Hämäläinen. High-Level synthesis for FPGAs—a hardware engineer's perspective. *IEEE Access*, 13:28574–28593, 2025.

[147] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: Efficient and fast RPCs in Cloud microservices with near-memory reconfigurable NICs. In *Proceedings of the 26th ACM International Conference on Architectural Support for*

Programming Languages and Operating Systems, ASPLOS '21, pages 36–51, New York, NY, USA, 2021. Association for Computing Machinery.

[148] Collin Lee and John Ousterhout. Granular computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 149–154, 2019.

[149] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. MIND: In-Network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 488–504, New York, NY, USA, 2021. Association for Computing Machinery.

[150] Alberto Lerner, Davide Zoni, Paolo Costa, and Gianni Antichi. Rethinking the switch architecture for stateful in-network computing. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*, HotNets '24, page 273–281, New York, NY, USA, 2024. Association for Computing Machinery.

[151] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2020.

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

[153] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 467–483, 2016.

[154] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: Tolerating skewed workloads in distributed storage with In-Network coherence directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 387–406. USENIX Association, November 2020.

[155] Hwijoon Lim, Wei Bai, Yibo Zhu, Youngmok Jung, and Dongsu Han. Towards timeout-less transport in commodity datacenter networks. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 33–48, New York, NY, USA, 2021. Association for Computing Machinery.

[156] Katie Lim, Matthew Giordano, Theano Stavrinos, Irene Zhang, Jacob Nelson, Baris Kasikci, and Thomas Anderson. Beehive: A flexible network stack for Direct-Attached accelerators. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 393–408, 2024.

[157] Katie Lim, Matthew Giordano, Irene Zhang, Baris Kasikci, and Thomas Anderson. Apiary: An OS for the modern FPGA. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, pages 103–110, 2025.

[158] Jiaxin Lin, Zhiyuan Guo, Mihir Shah, Tao Ji, Yiying Zhang, Daehyeok Kim, and Aditya Akella. Enabling portable and High-Performance SmartNIC programs with Alkali. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 107–126, 2025.

[159] Jiaxin Lin, Kiran Patel, Brent E Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 243–259, 2020.

[160] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. Scalable kernel TCP design and implementation for Short-Lived connections. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 339–352, New York, NY, USA, 2016. Association for Computing Machinery.

[161] Linux. Efficient I/O with *io_uring*. https://kernel.dk/io_uring.pdf, 2019.

[162] Linux. *io_uring* — Linux manual page. https://man7.org/linux/man-pages/man7/io_uring.7.html, 2020.

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

[164] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[165] Ming Liu. Fabric-centric computing. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 118–126, 2023.

[166] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto SmartNICs using IPipe. In *Proceedings of the*

2019 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '19, pages 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

[167] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 795–809, 2017.

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

[169] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM symposium on cloud computing*, pages 412–426, 2021.

[170] Xuhao Luo, Shreesha G. Bhat, Jiyu Hu, Ramnatthan Alagappan, and Aishwarya Ganesan. LazyLog: A new shared log abstraction for Low-Latency applications. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 296–312, New York, NY, USA, 2024. Association for Computing Machinery.

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

[172] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[173] Mellanox. ConnectX-3 Ethernet single and dual SFP+ port adapter card user manual. https://network.nvidia.com/pdf/user_manuals/ConnectX-3_Ethernet_Single_and_Dual_SFP+_Port_Adapter_Card_User_Manual.pdf, 2014. [Accessed 01-11-2025].

[174] Mellanox. ConnectX-3 pro Ethernet single and dual SFP+ port adapter card user manual. https://network.nvidia.com/files/doc-2020/

connectx-3-pro-ethernet-single-and-dual-qsfp+-port-adapter-card-user-manual.pdf, 2017. [Accessed 01-11-2025].

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

[176] William M Mellette, Alex Forencich, Rukshani Athapathu, Alex C Snoeren, George Papen, and George Porter. Realizing RotorNet: Toward practical microsecond scale optical networking. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 392–414, 2024.

[177] William M Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C Snoeren, and George Porter. RotorNet: A scalable, low-complexity, optical datacenter network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 267–280, 2017.

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

[179] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From Luna to Solar: the evolutions of the compute-to-storage networks in Alibaba cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 753–766, New York, NY, USA, 2022. Association for Computing Machinery.

[180] Oliver Michel, Roberto Bifulco, Gábor Rétvári, and Stefan Schmid. The programmable data plane: Abstractions, architectures, algorithms, and applications. *ACM Computing Surveys (CSUR)*, 54(4):1–36, 2021.

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

[182] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 106–122, New York, NY, USA, 2021. Association for Computing Machinery.

[183] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA reads to build a fast, CPU-Efficient Key-Value store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, San Jose, CA, June 2013. USENIX Association.

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

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

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

[187] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-Driven Low-Latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 221–235, New York, NY, USA, 2018. Association for Computing Machinery.

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

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

[190] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring endpoint congestion control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 30–43, New York, NY, USA, 2018. Association for Computing Machinery.

[191] Netberg. Netberg Aurora 810. https://netbergtw.com/products/aurora-810/, 2024.

[192] Netronome. Agilio SmartNIC Open-vSwitch user guide. https://help.netronome.com/support/solutions/articles/36000081172-agilio-open-vswitch-tc-user-guide. [Accessed 03-11-2025].

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

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

[195] Netronome. Agilio©CX 2x40GbE SmartNIC. https://www.netronome.com/media/documents/PB_Agilio_CX_2x40GbE-7-20.pdf, 2020.

[196] Netronome. NFP-4000 theory of operation. https://www.netronome.com/media/documents/WP_NFP4000_TOO.pdf, 2020.

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

[198] Joel Nider, Craig Mustard, Andrada Zoltan, John Ramsden, Larry Liu, Jacob Grossbard, Mohammad Dashti, Romaric Jodin, Alexandre Ghiti, Jordi Chauzi, et al. A case study of Processing-in-Memory in off-the-Shelf systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 117–130, 2021.

[199] Karen Nielsen. RFC 9260: Stream control transmission protocol. https://datatracker.ietf.org/doc/html/rfc9260. [Accessed 02-11-2025].

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

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

[202] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out NUMA. *ACM SIGPLAN Notices*, 49(4):3–18, 2014.

[203] NVIDIA. NVIDIA ConnectX-4 InfiniBand/Ethernet adapter cards user manual. https://docs.nvidia.com/networking/display/connectx4ib/specifications#

src-15050793_Specifications-MCX455A-ECATSpecifications. [Accessed 02-11-2025].

[204] NVIDIA. NVIDIA ConnectX-5 InfiniBand/Ethernet adapter cards user manual. https://docs.nvidia.com/networking/display/connectx5en/specifications. [Accessed 02-11-2025].

[205] NVIDIA. NVIDIA ConnectX-6 dx InfiniBand/Ethernet adapter cards user manual. https://docs.nvidia.com/networking/display/connectx6dxen/specifications. [Accessed 02-11-2025].

[206] NVIDIA. NVIDIA ConnectX-7 VPI adapter cards user manual. https://docs.nvidia.com/networking/display/connectx7vpi/specifications. [Accessed 02-11-2025].

[207] NVIDIA. NVIDIA Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) v2.6.1. https://docs.nvidia.com/networking/display/sharpv261. [Accessed 02-11-2025].

[208] NVIDIA. NVIDIA DGX A100: The universal system for AI infrastructure. https://images.nvidia.com/aem-dam/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf, 2022.

[209] NVIDIA. NVIDIA ConnectX-8 SuperNICs advance AI platform architecture with PCIe gen6 connectivity. https://developer.nvidia.com/blog/nvidia-connectx-8-supernics-advance-ai-platform-architecture-with-pcie-gen6-connectivity/, 2025. [Accessed 02-11-2025].

[210] NVIDIA. NVIDIA Spectrum-4 ASIC. https://nvdam.widen.net/s/pjlcwnrdbn/ethernet-switches-spectrum-4-asic-datasheet-us, 2025.

[211] NVIDIA Corporation. NVIDIA NICs performance report with DPDK 17.11. https://fast.dpdk.org/doc/perf/DPDK_17_11_Mellanox_NIC_performance_report.pdf, 2018.

[212] NVIDIA Corporation. NVIDIA NICs performance report with DPDK 18.11. https://fast.dpdk.org/doc/perf/DPDK_18_11_Mellanox_NIC_performance_report.pdf, 2019.

[213] NVIDIA Corporation. NVIDIA NICs performance report with DPDK 19.11. https://fast.dpdk.org/doc/perf/DPDK_19_11_Mellanox_NIC_performance_report.pdf, 2020.

[214] NVIDIA Corporation. NVIDIA NICs performance report with DPDK 20.11. https://fast.dpdk.org/doc/perf/DPDK_20_11_Mellanox_NIC_performance_report.pdf, 2021.

[215] NVIDIA Corporation. NVIDIA NICs performance report with DPDK 21.08. https://fast.dpdk.org/doc/perf/DPDK_21_08_Mellanox_NIC_performance_report.pdf, 2021.

[216] NVIDIA Corporation. NVIDIA NICs performance report with DPDK 21.11. https://fast.dpdk.org/doc/perf/DPDK_21_11_Mellanox_NIC_performance_report.pdf, 2022.

[217] NVIDIA Corporation. NVIDIA NICs performance report with DPDK 22.11. https://fast.dpdk.org/doc/perf/DPDK_22_11_NVIDIA_Mellanox_NIC_performance_report.pdf, 2022.

[218] NVIDIA Corporation. NVIDIA bluefield-2 Ethernet DPU user guide: Specifications. https://docs.nvidia.com/networking/display/bluefield2dpuenug/specifications, 2024.

[219] NVIDIA Corporation. NVIDIA ConnectX-7 400G adapter card. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/infiniband-adapters/infiniband-connectx7-data-sheet.pdf, 2024.

[220] NVIDIA Corporation. NVIDIA NICs performance report with DPDK 23.11. https://fast.dpdk.org/doc/perf/DPDK_23_11_NVIDIA_NIC_performance_report.pdf, 2024.

[221] NVIDIA Corporation. NVIDIA NICs performance report with DPDK 24.07. https://fast.dpdk.org/doc/perf/DPDK_24_07_NVIDIA_NIC_performance_report.pdf, 2024.

[222] NVIDIA Corporation. NVIDIA bluefield-3 DPU controller user manual: Specifications. https://docs.nvidia.com/networking/display/bf3dpucontroller/specifications, 2025.

[223] John Ousterhout. A Linux kernel implementation of the Homa transport protocol. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 99–115, 2021.

[224] John Ousterhout. It's time to replace TCP in the Datacenter. *arXiv preprint arXiv:2210.00714*, 2022.

[225] Christoph Paasch. RFC 8684: TCP extensions for multipath operation with multiple addresses. https://datatracker.ietf.org/doc/html/rfc8684. [Accessed 02-11-2025].

[226] Seo Jin Park, Ramesh Govindan, Kai Shen, David Culler, Fatma Özcan, Geon-Woo Kim, and Hank Levy. Lovelock: Towards SmartNIC-Hosted Clusters. *SIGENERGY Energy Inform. Rev.*, 4(5):172–179, April 2025.

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

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

[229] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A programming system for NIC-Accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, Carlsbad, CA, October 2018. USENIX Association.

[230] Solal Pirelli and George Candea. A simpler and faster NIC driver model for network functions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 225–241. USENIX Association, November 2020.

[231] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. FlowBlaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, Boston, MA, February 2019. USENIX Association.

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

[233] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating Large-Scale Datacenter services.

In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 13–24. IEEE Press, 2014.

[234] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, et al. Alibaba HPN: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 691–706, 2024.

[235] Deepti Raghavan, Shreya Ravi, Gina Yuan, Pratiksha Thaker, Sanjari Srivastava, Micah Murray, Pedro Henrique Penna, Amy Ousterhout, Philip Levis, Matei Zaharia, et al. Cornflakes: Zero-copy serialization for microsecond-scale networking. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 200–215, 2023.

[236] Parthasarathy Ranganathan. A Six-Word story on the future of VLSI: AI-driven, Software-defined, and Uncomfortably Exciting. In *2023 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*, pages 1–4, 2023.

[237] Parthasarathy Ranganathan and Urs Hölzle. Twenty five years of Warehouse-Scale computing. *IEEE Micro*, 44(5):11–22, 2024.

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

[239] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. RDMA is turing complete, we just did not know it yet! In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 71–85, 2022.

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

[241] 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 Conference on Special Interest Group on Data Communication*, pages 123–137, 2015.

[242] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio López-Buedo. Limago: An FPGA-based Open-Source 100 GbE TCP/IP stack. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 286–292, 2019.

[243] Karl Rupp. 50 years of microprocessor trend data. https://github.com/karlrupp/microprocessor-trend-data/blob/master/50yrs/50-years-processor-trend.pdf, Feb 2022.

[244] Anastasiia Ruzhanskaia, Pengcheng Xu, David Cock, and Timothy Roscoe. Rethinking programmed I/O for fast devices, cheap cores, and coherent interconnects. *arXiv preprint arXiv:2409.08141*, 2024.

[245] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S. Berger, James C. Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. Enso: A streaming interface for NIC-Application communication. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 1005–1025, Boston, MA, July 2023. USENIX Association.

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

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

[248] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostić, and Marco Chiesa. A High-Speed stateful packet processing approach for Tbps programmable switches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1237–1255, Boston, MA, April 2023. USENIX Association.

[249] Henrik Schuh. *Accelerating Networked Systems With Programmable and Tightly-Coupled NICs*. PhD thesis, 2023. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2024-01-30.

[250] Henry N. Schuh, Arvind Krishnamurthy, David Culler, Henry M. Levy, Luigi Rizzo, Samira Khan, and Brent E. Stephens. CC-NIC: a Cache-Coherent interface to the NIC. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, page 52–68, New York, NY, USA, 2024. Association for Computing Machinery.

[251] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-accelerated distributed transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 740–755, 2021.

[252] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M

Levy. A cloud-scale characterization of remote procedure calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 498–514, 2023.

[253] Aashaka Shah, Abhinav Jangda, Binyang Li, Caio Rocha, Changho Hwang, Jithin Jose, Madan Musuvathi, Olli Saarikivi, Peng Cheng, Qinghua Zhou, et al. MSCCL++: Rethinking GPU communication abstractions for cutting-edge AI applications. *arXiv preprint arXiv:2504.09014*, 2025.

[254] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*, pages 205–218, 2020.

[255] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. A Cloud-Optimized transport protocol for elastic and scalable HPC. *IEEE Micro*, 40(6):67–73, 2020.

[256] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 67–82, Boston, MA, March 2017. USENIX Association.

[257] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 1–16, Renton, WA, April 2018. USENIX Association.

[258] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable calendar queues for high-speed packet scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 685–699, Santa Clara, CA, February 2020. USENIX Association.

[259] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 87–102, Renton, WA, April 2022. USENIX Association.

[260] Arman Shehabi, Alex Hubbard, Alex Newkirk, Nuoa Lei, Md Abu Bakkar Siddik, Billie Holecek, Jonathan Koomey, Eric Masanet, Dale Sartor, et al. 2024 united states data center energy usage report. 2024.

[261] Justine Sherry. The I/O driven server: From SmartNICs to data movement controllers. *SIGCOMM Comput. Commun. Rev.*, 53(3):9–17, February 2024.

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

[263] Vishal Shrivastav. Stateful multi-pipelined programmable switches. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 663–676, New York, NY, USA, 2022. Association for Computing Machinery.

[264] Min Si, Pavan Balaji, Yongzhou Chen, Ching-Hsiang Chu, Adi Gangidi, Saif Hasan, Subodh Iyengar, Dan Johnson, Bingzhe Liu, Jingliang Ren, et al. Collective communication for 100k+ GPUs. *arXiv preprint arXiv:2510.20171*, 2025.

[265] Patrick Siegl, Rainer Buchty, and Mladen Berekovic. Data-centric computing frontiers: A survey on processing-in-memory. In *Proceedings of the Second International Symposium on Memory Systems*, pages 295–308, 2016.

[266] Mark Silberstein, Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, and Emmett Witchel. GPUnet: Networking abstractions for GPU programs. *ACM Transactions on Computer Systems (TOCS)*, 34(3):1–31, 2016.

[267] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 1RMA: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 708–721, New York, NY, USA, 2020. Association for Computing Machinery.

[268] Arjun Singhvi, Nandita Dukkipati, Prashant Chandra, Hassan MG Wassel, Naveen Kr Sharma, Anthony Rebello, Henry Schuh, Praveen Kumar, Behnam Montazeri, Neelesh Bansod, et al. Falcon: A reliable, low latency hardware transport. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 248–263, 2025.

[269] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 44–57, 2016.

[270] Athinagoras Skiadopoulos, Zhiqiang Xie, Mark Zhao, Qizhe Cai, Saksham Agarwal, Jacob Adelmann, David Ahern, Carlo Contavalli, Michael Goldflam, Vitaly Mayatskikh, et al. High-throughput and flexible host networking for accelerated computing. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 405–423, 2024.

[271] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with Exception-Less system calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[272] Cha Hwan Song, Xin Zhe Khooi, Raj Joshi, Inho Choi, Jialin Li, and Mun Choon Chan. Network load balancing with in-network reordering support for RDMA. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 816–831, New York, NY, USA, 2023. Association for Computing Machinery.

[273] SPDK. Block device user guide. https://spdk.io/doc/bdev.html.

[274] SPDK. Running benchmarks with perf tool. https://spdk.io/doc/nvme.html.

[275] Matheus Stolet, Liam Arzola, Simon Peter, and Antoine Kaufmann. Virtuoso: High resource utilization and μs-scale performance isolation in a shared virtual machine TCP network stack. *arXiv preprint arXiv:2309.14016*, 2023.

[276] Weibin Sun and Robert Ricci. Fast and flexible: Parallel packet processing with GPUs and Click. In *Architectures for Networking and Communications Systems*, pages 25–35. IEEE, 2013.

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

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

[279] Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. GASPP: A GPU-Accelerated stateful packet processing framework. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 321–332, 2014.

[280] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX annual technical conference (USENIX ATC 18)*, pages 133–146, 2018.

[281] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with In-Network cache coherence. In *19th USENIX Conference*

on File and Storage Technologies (FAST 21), pages 277–292. USENIX Association, February 2021.

[282] Tao Wang, Xiangrui Yang, Gianni Antichi, Anirudh Sivaraman, and Aurojit Panda. Isolation mechanisms for High-Speed Packet-Processing pipelines. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1289–1305, Renton, WA, April 2022. USENIX Association.

[283] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 739–767, 2023.

[284] Ertza Warraich, Ali Imran, Annus Zulfiqar, Shay Vargaftik, Sonia Fahmy, and Muhammad Shahbaz. Reimagining RDMA through the lens of ML. *IEEE Computer Architecture Letters*, pages 1–4, 2025.

[285] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 233–251, 2018.

[286] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. Zerializer: Towards zero-copy serialization. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 206–212, 2021.

[287] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. Runtime programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 651–665, Renton, WA, April 2022. USENIX Association.

[288] Pengcheng Xu and Timothy Roscoe. The NIC should be part of the OS. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, pages 151–157, 2025.

[289] Yifan Yang, Lin He, Jiasheng Zhou, Xiaoyi Shi, Jiamin Cao, and Ying Liu. P4runpro: Enabling runtime programmability for RMT programmable switches. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM ’24, page 921–937, New York, NY, USA, 2024. Association for Computing Machinery.

[290] 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 Conference on Usenix Annual Technical Conference*, USENIX ATC ’16, page 43–56, USA, 2016. USENIX Association.

[291] Yifan Yuan, Omar Alama, Jiawei Fei, Jacob Nelson, Dan R. K. Ports, Amedeo Sapio, Marco Canini, and Nam Sung Kim. Unlocking the power of inline Floating-Point operations on programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 683–700, Renton, WA, April 2022. USENIX Association.

[292] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel datapath OS architecture for microsecond-scale Datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 195–211, New York, NY, USA, 2021. Association for Computing Machinery.

[293] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. Gallium: Automated software middlebox offloading to programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 283–295, 2020.

[294] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 724–739, 2021.

[295] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. CRISP: Critical path analysis of Large-Scale microservice architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 655–672, 2022.

[296] Chenxingyu Zhao, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. White-Boxing RDMA with Packet-Granular software control. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 427–449, 2025.

[297] Liangyu Zhao, Saeed Maleki, Ziyue Yang, Hossein Pourreza, and Arvind Krishnamurthy. ForestColl: Throughput-Optimal collective communications on heterogeneous network fabrics. *arXiv preprint arXiv:2402.06787*, 2024.

[298] Liangyu Zhao, Siddharth Pal, Tapan Chugh, Weiyang Wang, Prithwish Basu, Joud Khoury, and Arvind Krishnamurthy. Optimal direct-connect topologies for collective communications. *arXiv preprint arXiv:2202.03356*, 2022.

[299] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik

Pol. Understanding data storage and ingestion for large-scale deep recommendation model training: industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 1042–1057, New York, NY, USA, 2022. Association for Computing Machinery.

[300] Lingjun Zhu, Yifan Shen, Erci Xu, Bo Shi, Ting Fu, Shu Ma, Shuguang Chen, Zhongyu Wang, Haonan Wu, Xingyu Liao, Zhendan Yang, Zhongqing Chen, Wei Lin, Yijun Hou, Rong Liu, Chao Shi, Jiaji Zhu, and Jiesheng Wu. Deploying user-space TCP at Cloud scale with LUNA. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 673–687, Boston, MA, July 2023. USENIX Association.

[301] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, XiongChun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. Dissecting overheads of service mesh sidecars. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 142–157, New York, NY, USA, 2023. Association for Computing Machinery.

[302] Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda, Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, et al. Resiliency at scale: Managing Google's TPUv4 machine learning supercomputer. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 761–774, 2024.