The Thesis Committee for Rajath Shashidhara
Certifies that this is the approved version of the following Thesis:

TASNIC: A Flexible TCP offload with Programmable SmartNICs

APPROVED BY
SUPERVISING COMMITTEE:

Simon Peter, Supervisor

Aditya Akella

TASNIC: A Flexible TCP offload with Programmable SmartNICs

by

Rajath Shashidhara

Thesis

Presented to the Faculty of the Graduate School

of The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Computer Science

The University of Texas at Austin

May 2021

Abstract

TASNIC: A Flexible TCP offload with Programmable SmartNICs

Rajath Shashidhara, MSCompSci

The University of Texas at Austin, 2021

Supervisor: Simon Peter

The CPU overhead of TCP packet processing is increasingly prohibitive. Kernel-bypass stacks and existing hardware offloads are not enough, causing operational issues at scale and cannot keep up with network protocol evolution. We take advantage of programmable SmartNICs to build TASNIC, a flexible, yet high-performance TCP offload engine (TOE) in software. TASNIC eliminates almost all host CPU packet processing, but retains compatibility with POSIX sockets and places no demands on data center networks. TASNIC allows complete customization of transport logic, providing performance *and* flexibility.

TCP offload to SmartNICs is challenging. SmartNICs are geared towards massively parallel stateless offloads, while TCP is a complex stateful protocol, sensitive to packet reordering. TASNIC leverages fast-path processing of common TCP code paths, fine-grained parallelization of the TCP data-path, and near-memory computing for high performance, while remaining flexible via a modular design. We compare TASNIC to Linux, the TAS TCP accelerator, and the Chelsio Terminator TOE. We find that Memcached scales up to 38% better on TASNIC versus TAS, while saving up to 80% host CPU cycles versus Chelsio. For 64B RPCs, TASNIC cuts 99th-percentile latency to 42% and provides 70% higher throughput versus TAS, and an order of magnitude higher throughput under packet loss than Chelsio. TASNIC interoperates well with other TCP stacks and is easily extensible.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern datacenter applications demand high throughput, low latency network access. The CPU overhead of providing lossless, in-order high-performance network transport in software is increasingly prohibitive. Particularly in Remote Procedure Call (RPC) workloads, short median packet sizes exacerbate the problem. However, TCP remains the default protocol in data centers, enterprise, and wide area networks, even as the widening gap between network and processor speeds has fueled innovation across the stack. Despite substantial performance and efficiency improvements, mainly in kernel-bypass systems [41, 3, 15], the overhead is still significant. As a result, the adoption of purely hardware-based network stacks [5, 44], specialized protocols [21, 34] and APIs [11, 17, 44], and combinations thereof is growing in prevalence. Unfortunately, these pose significant large-scale deployment challenges [10, 27], limiting adoption to niche deployments, such as in backend service implementations [27]. The vast majority of applications continue to rely on conventional TCP stacks, eschewing performance benefits of specialized APIs and protocols for portability and interoperability.

A long line of improvements to software TCP stack architectures has reduced overheads: Careful packet steering improves cache-locality for multi-cores [40, 24, 15], kernel-bypass enables safe direct NIC access from user-space [3, 41], application libraries avoid system calls for common socket operations [15], and fast-paths drastically reduce TCP processing overheads [20]. Yet, even with these optimizations, communication intensive applications continue to devote as much as 74% of CPU cycles to network packet handling [20].

Alternatively, co-designing applications with accelerators and network transports is an avenue to extract high performance. For example, numerous key-value stores [18, 30], transaction processing systems [8, 42], and RPC frameworks [19] specialized for RDMA transports have been proposed. Recent works take this a step further by implementing parts of the application in-network using programmable network devices [14, 16, 23, 22]. Even purely software-based tight application protocol couplings, such as eRPC [17] and R2P2 [21], require developers to take transport constraints into account, such as weakened delivery semantics,

buffer management, and execution models. This kind of co-design breaks the clean abstraction barriers between applications, the operating system, and the hardware and ties applications to specific hardware capabilities and performance characteristics, causing software reuse and management to become a significant challenge.

Hardware protocol offload is a promising avenue for reducing software overheads. Moving parts of network processing, such as checksums and segmentation, from software into specialized hardware improves efficiency [47]. However, TCP offload engines (TOEs) [5, 4] that offload most [29] or all protocol processing have so far failed to find adoption. We argue that this is because of the rigidity of fixed hardware offloads. This type of offload precludes customization [48] of the transport logic and limits protocol evolution after deployment [33, 9].

Finally, FPGA-based offload approaches, such as ToNIC [2], can achieve programmable transport protocol offload at 100Gbps line rates. However, the near-hardware development model forced the ToNIC developers to abandon TCP's stream abstraction and instead requires users to choose a fixed segment size for data transfers. Further, when offloads have to be programmed in a low level hardware description language, it hampers development velocity.

We present TASNIC, a high-performance, yet flexible implementation of the widely-used TCP protocol that maintains application compatibility, but is offloaded to a programmable SmartNIC. Our solution alleviates all the aforementioned issues and represents a sweet spot on the performance-efficiency-generality trade-off.

TASNIC offloads the TCP datapath to Netronome Agilio CX40 SmartNIC [36] to eliminate the CPU overhead due to packet processing. Applications interface directly with the hardware datapath through the libTASNIC library that implements fully backwards compatible POSIX sockets. Unmodified applications can take advantage of TASNIC by simply linking with the TASNIC library.TASNIC thereby removes kernel-crossings and data copies for data transfers, eliminating almost all host CPU packet processing overhead (§5.3). Furthermore, TASNIC places no demands on the underlying data center network. It operates well, even under loss and congestion (§5.2). TASNIC allows complete customization of transport logic and flexibility to implement new features, such as congestion control algorithms and traffic monitoring and auditing (§5.4). Additionally, as SmartNICs are much

less power-hungry as compared to server cores, significant energy savings is a practical benefit of our design.

Stateful TCP packet processing on SmartNICs is challenging. SmartNICs are heavily resource-constrained and support restrictive programming models [25]. Achieving line-rate packet processing imposes stringent per-packet time budgets. Furthermore, SmartNICs are geared towards massively parallel processing (60×8-way multi-threaded cores in our case) of packets optimized for stateless workloads such as checksum and segmentation offloads. To operate within the tight energy budget of a PCIe expansion card, SmartNICs have a complex hierarchy of extremely small fast memories that have to be managed by software [37]. Dealing with multiple layers of non-uniform memory (5 memory types for our SmartNIC) with differing access characteristics and capacity is challenging. Finally, SmartNICs have a myriad of hardware accelerators. How best to utilize them is largely unclear for transport protocol offload. On the other hand, TCP is a complex protocol that is not amenable to parallelization. It is extremely sensitive to packet reordering and losses. Also, TCP requires a large amount of per-flow state to track in-flight segments for segment reassembly and retransmission. Lack of support for timers and floating-point operations presents a challenge to congestion control enforcement. Furthermore, limiting transfers across the high-latency PCIe interface is also a requirement.

Resolving the gap between TCP's requirements and SmartNIC hardware capabilities requires careful design of the data-path to maximize parallelism and to efficiently utilize available fast memory. TASNIC leverages fast-path processing, fine-grained parallelism, and near-memory computing for high performance, while remaining flexible via a modular design. We separate common TCP code paths from exceptions and focus the SmartNIC offload only on the common case (*fast-path processing* [20]). Connection management, retransmissions, and congestion control policy are executed on a separate slow-path running on the host. We organize per-packet processing tasks into fine-grained, reusable modules that keep private state and communicate explicitly (*modularity*), simplifying development and integration. We organize these packet processing modules as a *fine-grained, data-parallel computation pipeline* and minimize co-ordination among modules by leveraging private state. We accelerate each pipeline stage using the *near-memory computing*

facilities of the SmartNIC architecture. We also leverage hardware *sequencing and reordering* to support parallel, out-of-order processing of pipeline stages, even for the same TCP connection, while enforcing in-order segment delivery where necessary.

We make the following contributions:

- We present TASNIC, a flexible, high-performance TCP offload engine. TASNIC leverages fast-path processing, fine-grained parallelism, and near-memory computation for performance, but remains flexible via a modular design.
- We implement TASNIC on the Netronome Agilio-CX40 40Gbps SmartNIC architecture. The Agilio-CX40 is a many-core architecture with near-memory computing capabilities providing multiple layers of memory that accelerate concurrent data structures and synchronization. SmartNICs are powerful, yet flexible, and, as we show, support the offload of complex transport protocols, such as TCP.
- We evaluate TASNIC on a range of workloads and compare to Linux, the high-performance kernel-bypass TAS [20] network stack, and a Chelsio Terminator NIC [4] with a TCP offload engine. We find that TASNIC cuts 99th-percentile latency to 42% and provides 70% higher throughput for 64B RPCs versus TAS, 20% higher throughput than Chelsio for long flows,and an order of magnitude higher throughput under 5% packet loss than Chelsio. The Memcached [28] key-value store scales throughput up to 38% better on TASNIC than using TAS, while saving up to 80% more host CPU cycles than Chelsio. We add debugging and auditing functionality to the TASNIC data-path in < 2 person-hours. TASNIC maintains high performance when interoperating with the other evaluated network stacks.

Chapter 2

Background

Packet processing offload is by no means a new idea. Numerous existing projects have provided solutions for various points in the design space. We start by providing an overview of the transport protocol implementation space and place our approach in context of the related work. We then provide detail on the SmartNIC we use for TASNIC.

2.1 Related Work

Table 2.1 provides an overview of the features of existing transport protocol implementations and TASNIC.

In-kernel. The Linux TCP stack is the most widely used solution despite its well-known overheads. As the stack resides completely inside the Linux kernel, every interaction with the application requires a switch to privileged execution mode and data copies for security. The complex design of the stack leads to high code and memory footprints. It is found to be severely deficient in performance especially for small-packet sizes and large number of connections [20]. Stateless offloads [47] such as Large Segmentation Offload (LSO) and Generic Receive Offload (GRO) [12], aid Linux, but its CPU overhead remains high [20]. The Linux stack has minimal operational complexity and most applications are designed for the POSIX sockets interface. With TASNIC we aim for binary API and protocol compatibility

Design	Flexibility	Hardware	Performance	CPU overhead
In-kernel	Kernel update	None	Poor	Very High
Kernel-bypass	C code	None	Excellent	High
TOE	No	TOE	Moderate	None
RDMA	No	RDMA	Excellent	None
ToNIC	Verilog	FPGA	Excellent	None
TASNIC	C code	SoC	Excellent	None

Table 2.1: Features of transport protocol implementations.

with Linux, leveraging prior insights on scalable implementations of those APIs in Linux [40, 24], while drastically reducing CPU overheads via offload.

Kernel bypass. Kernel-bypass stacks such as mTCP [15] and Arrakis [41] eliminate kernel overheads by entrusting the TCP stack to the application, which has security implications. IX [3] utilizes hardware virtualization features to move the TCP stack to a different protection domain. TAS [20] and Snap [27] improve upon the previous systems by leveraging a protected fast-path on dedicated cores to handle common case packet processing, for TCP and RDMA respectively, achieving better performance, scalability, and modularity. TAS is a state-of-the-art kernel-bypass system. It improves upon the previous kernel bypass systems by leveraging a unique split - a dedicated fast-path to handle common case packet processing and a slow-path for exception code paths. TAS [20] achieves higher throughput and lower latency for RPC workloads than mTCP and IX. It also demonstrates much better connection scalability. Further, unlike other kernel-bypass systems, it is workload proportional. Even with all the efficiency optimizations, TAS still occupies a significant number of CPU cores. TASNIC also leverages kernel-bypass, and uses a similar fast-path architecture, but with the fast-path executing on a SmartNIC, thereby freeing up host CPU cores for applications.

Specialized APIs and protocols. Another step towards lower CPU utilization is specialization. R2P2 [21] is an UDP-based protocol for remote procedure calls (RPCs) optimized for efficient and parallel processing, both at the end-hosts and in the network. eRPC [17] goes a step further and co-designs an RPC protocol, a custom API, and a complete kernel-bypass network stack, to minimize CPU overhead per RPC. eRPC combines transport API with a threading model. To use eRPC, applications need a redesign to comply with the API, the threading model and use DMA-capable message buffers allocated by the eRPC framework. Developers must also decide whether to run the RPC handlers in a dispatch thread or in separate worker threads. Additionally, eRPC implements a client-driven protocol. It splits the transport logic between the server and the client, and thus it is incompatible with other network stacks. Furthermore, eRPC still uses server cores to implement the transport logic.

RDMA is a popular hardware-offload solution. Applications use the RDMA verbs interface to perform data transfer. Hence, often applications must be re-designed for this purpose. An RDMA-capable NIC manages the connection state and implements the transport logic. However, large-scale RDMA deployments have seen significant issues such as Head-of-Line (HoL) blocking, livelocks owing to the lack of flexibility and hardwired transport logic. Furthermore, RDMAs implement a simple reliable transport algorithm that performs poorly under losses. Thus, they rely on Datacenter Bridging (DCB) features for lossless networks such as Priority Flow Control (PFC) to function [50, 32].

These approaches improve processing efficiency, but at the cost of requiring application re-designs, all-or-nothing deployments, and operational issues at scale [10]. TASNIC instead implements the TCP protocol compatible with existing datacenter infrastructure and POSIX sockets.

Fixed offload. Specialization at the hardware level also drastically improves processing efficiency. TCP offload engines [5], such as the Chelsio Terminator [4], implement the TCP protocol in custom fixed hardware. TOEs retain compatibility by hooking into the kernel, but as a result still incur system call overheads for each operation (§5.1). Another popular offload is Microsoft's TCP chimney [29], which retains connection control within the OS kernel and executes data exchange on the NIC. RDMA is also usually implemented in hardware, but with support for kernel-bypass. While fixed offloads can drastically improve efficiency, they are inherently inflexible, leading to operational challenges [10, 33, 48, 49] including the inability to update protocols or fix incompatibilities with other protocol implementations. TASNIC instead relies on SoC-based SmartNICs programmable in C, improving flexibility.

Flexible offload. SmartNICs, based on network processors (NPUs) or FPGAs, provide a programmable substrate for flexible offload. iPipe [25] is an actor-based framework for offloading applications to NPU SmartNICs. AccelTCP [35] focuses on offloading TCP connection management and TCP connection splicing [26] for proxies to Netronome SmartNICs, while retaining data transfer operations in software. FPGAs generally offer higher performance, but their near-hardware pro-

gramming model makes development a challenge. ToNIC [2] provides building blocks for flexible transport protocol offload to FPGA-SmartNICs, but its architecture forces it to depart from TCP's byte-stream abstraction, requiring application developers to choose fixed segment sizes. Similarly, Catapult's primary use-case is network management offload [43]. TASNIC relies on an NPU-SmartNIC to offload TCP data transfers, the most common operation in data centers, while retaining the flexiblity of a software programming.

Parallel packet processing RouteBricks [6] parallelizes across cores and cluster nodes for high-performance routing, achieving high line-rates, but remaining flexible via software programmability. Routing relies on read-mostly state and is simple compared to TCP. TASNIC applies fine-grained parallelism to complex, stateful code paths.

2.2 Programable SmartNIC architecture

In this work, we use the Netronome Agilio-CX40 system-on-chip (SoC) SmartNIC platform to offload the TCP datapath. SoC-based SmartNICs support massively parallel processing of packets with a large pool of flow processing cores (FPCs). For example, with hardware-assisted multi-threading, the Agilio-CX40 can process up to 480 packets simultaneously, while hiding the latency of data movement. Multiple layers of non-uniform memory can store state for a large number of flows. With support for programming in a C dialect, called Micro-C, and open-source firmware for a number of built-in hardware accelerators, SoC-based SmartNICs are powerful, yet flexible, and support the offload of complex transport protocols, such as TCP.

The Netronome Agilio-CX40 SmartNIC belongs to the NFP-4000 device family [36, 37]. The NFP-4000 incorporates multiple functional blocks referred to as *islands*. We show the islands relevant to TASNIC as rectangular blocks in Figure 2.1. Islands are connected in a mesh via a high-bandwidth interconnect (arrows in Figure 2.1).

The *PCIe* island has two PCIe Gen3 x8 interfaces. FPCs can enqueue DMA descriptors to the DMA engine on the PCIe island to perform asynchronous bulk

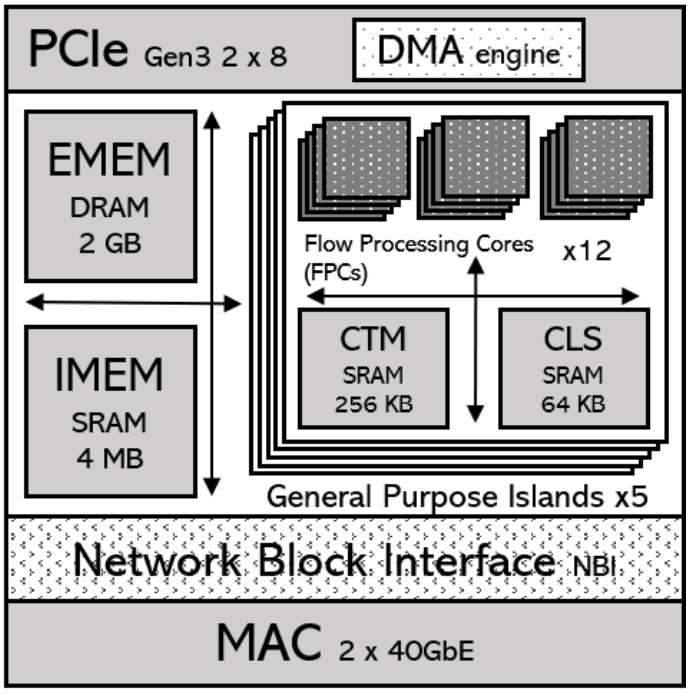

Figure 2.1: NFP-4000 functional islands.

transfers between host and device memories. The *MAC* island supports two 40Gbps Ethernet interfaces, accessed via a *network block interface* (NBI) that provides additional receive and transmit packet processing capabilities (described below).

2.2.1 Flow Processing Cores (FPCs)

NFP-4000 has 60 FPCs grouped into five islands, each containing 12 FPCs and island local memory. Each FPC is an independent 800MHz 32-bit RISC core with 32KB instruction memory, 4KB data memory, 256 general-purpose registers, hardware accelerators for CRC computation, and a 16-entry content addressable memory (CAM). The CAM tracks the LRU order of accesses and can evict entries based on it. PCs support up to eight non-preemptible threads designed as co-operative processing with hardware-assisted context switching (2-cycle overhead). FPCs do not support a stack due to which the program variables are accessed either from registers or local memory. Each thread has a private program counter. Typically, the register pool and local memory are partitioned among threads and are accessed

in a context-relative way to minimize context switching overhead. There is no native support for a program stack.

2.2.2 Command push-pull (CPP) bus

FPCs interact with other functional units by issuing commands over a *command push-pull* (CPP) bus. Instructions accessing the CPP bus may take several hundred cycles to complete. *Transfer registers* serve as the data buffers for asynchronous commands over the CPP bus. Data read from a remote unit is transferred into *read-transfer registers*. Similarly, data transfer out of FPCs goes through *write-transfer registers*. *Signals* notify the FPCs once the asynchronous command execution completes. Until notification is received, the associated transfer registers must not be modified/accessed. Each FPC provides 32 read and 32 write transfer registers and 15 signals per thread. FPCs typically hide the latency associated with CPP commands by switching over to a different thread context until the operation is complete. Threads can voluntarily swap out while waiting for one or more signals to be asserted. A hardware arbitrator on the FPC maintains a list of threads and their activations and schedules them accordingly.

2.2.3 Memory

The NFP-4000 is a *near-memory compute* architecture, where FPCs send asynchronous memory commands to memory units via the CPP bus. Memory units support bulk access of up to 128B and atomic access of up to 32B. A *lookup engine* exposes a CAM/TCAM, hash table, and trie for fast matching. A *queue memory engine* supports 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 may be used to coordinate threads.

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

local FPCs for quick command execution and data transfers. 4KB of FPC-local memory has access latency of 1–3 cycles.

Typically, LM is used to store data needed for every packet. CLS stores data required for most packets, small shared tables, and intra-island queues. CTM stores packet headers & metadata. IMEM stores packet bodies, medium-sized shared tables, global queues, and medium-sized shared tables. Finally, EMEM is both the largest and slowest memory. EMEM can store large tables to support a large number of flows.

While there are multiple levels of memory in the NFP-4000 architecture, they are disjoint. They operate in different address spaces, maintain a single copy of data as there is no hardware support for caching or coherence between the different levels in the hierarchy. Furthermore, memory units and the CPP bus operate in parallel. Commands issued to a memory unit even from the same thread may be reordered and execute in parallel. Memory accesses may be synchronized using primitives such as atomics, signals, and hardware-assisted concurrent data structures (§ 2.2.6).

2.2.4 Network Block Interface (NBI)

The NBI can perform certain line-rate packet processing via 48 *packet-processing cores (PPCs)*. On packet ingress, *packet characterizers* perform header parsing, assign sequence numbers, allocate buffers for headers and payload on any general-purpose island (while load balancing among islands), and copy headers and payload to the allocated buffers. On egress, a *packet modification engine* accepts instructions to modify per-packet data at programmed offsets. A *packet sequencer* maintains packet order based on assigned sequence numbers. The *traffic manager* performs traffic shaping and scheduling based on transmission queue priorities. PPCs are configured statically and via metadata appended to each packet.

We configure the NBI to use the entire CTM memory in the general-purpose islands to buffer packets. We set allocation limits, such that available memory is partitioned equally between packets received from the network and packets buffered for transmission. This ensures that neither RX nor TX overwhelms the other.

2.2.5 Buffer list manager (BLM)

The BLM is a software module that provides EMEM and IMEM buffer allocation services to other FPCs. The BLM interacts with the NBI to automatically recycle buffers of transmitted packets. It is required for proper function of the NBI.

2.2.6 Synchronization

With 480 threads running concurrently, it is challenging to coordinate them to preserve consistency and order. To this end, the NFP-4000 architecture supports several synchronization primitives.

Threads on an FPC synchronize registers and local memory variables by only yielding the processor outside of a critical section. Atomic memory engines present in island, internal and external memory units support atomic arithmetic, test-and-add, and compare-write operations. Using these primitives, semaphores and spinlocks can be constructed. Intra-FPC and inter-FPC signaling can orchestrate threads in a specific communication topology similar to MPI-style programming. Concurrent data structures such as linked lists, ring buffers, journals, work-stealing queues, and hash tables are supported natively on transactional memories. Transactional memories also support event generation. With this feature, when certain memory regions are modified, memory engines automatically push events to a configured FPC. Finally, threads use 96-entry ticket locks in the memory unit to reorder packets (or events) based on the sequence number. NBI assigns a sequence number to every packet in the order of ingress. FPCs may process the packets out-of-order and synchronize at the end using the sequence number assigned by NBI. Sequencers in the NBI output packets on the MAC interface roughly in order of their sequence numbers.

2.2.7 Global Reordering Block (GRO)

The GRO is a another software block shipped with the Netronome SDK that utilizes the NFP-4000 near-memory primitives to provide light-weight event ordering. It can execute on any of the general-purpose FPCs, accepting commands

via asynchronous messages from other FPCs. The GRO can reorder up to 40 Mpps. We use the GRO to reorder packets between parallel, out-of-order stages and in-order stages of TASNIC's TCP data-path processing pipeline.

2.2.8 Programming in Micro-C

Netronome Micro-C [38] is a version of C89 with added support for FPC-specific constructs such as register and memory annotations, signals, and CPP commands. Micro-C lacks dynamic memory allocation and other standard library features. The Micro-C compiler aggressively inlines function calls, performing register allocation for the whole program. Recursion is not supported. Furthermore, each variable requires a storage annotation specifying the memory location and a scope annotation specifying which threads or FPCs share it. CPP commands enter as assembler inserts. Explicit declaration of transfer registers and signals for CPP commands is necessary. Programmers must also carefully choose points in the code where a thread can yield. Besides, floating-point data types and division operation are not part of the instruction set. Finally, due to the access properties of NFP-4000 memories, data-structures that require multiple indirections or pointer chasing must be avoided as it would almost certainly lead to poor performance.

Chapter 3

Design

3.1 Objectives

TASNIC has the following goals:

- **Low tail latency and high throughput**. Modern datacenter network loads consist of short and long flows with wildly different performance requirements. Short flows, driven by remote procedure calls, require low latency completion time, even in the tail, while long flows benefit from high throughput. TASNIC shall provide flows what they need, regardless of the mix of active flows.
- **Scalability**. The number of network flows that servers must handle simultaneously is increasing. TASNIC shall scale with this demand and handle thousands of flows. The number of host processor cores is still increasing, as well. TASNIC shall scale to many host processor cores.
- **Efficiency**. The main motivation for offload is to make available precious host CPU cycles for applications. At the same time, SmartNIC resources are precious and should not be wasted. TASNIC shall maximize host CPU cycles available for application-level packet processing, while avoiding SmartNIC resource use for tasks that do not significantly reduce host CPU utilization. This implies that only sporadic control and exception processing may execute on host CPUs.
- **Compatiblity**. For deployment velocity, TASNIC should work with existing network infrastructure and applications. TASNIC shall support unmodified applications using POSIX sockets. It shall also inter-operate well with other TCP stacks and support standard TCP + congestion control protocols.
- **Flexibility**. The world around TASNIC is changing rapidly. Hence, TASNIC shall maximize customizability. Transport logic, application interfaces, congestion control, packet scheduling, application-specific offloads, auditing and debugging extensions should all be replaceable and customizable with minimum implementation effort.

3.2 Design Principles

To achieve these goals, we propose a set of design principles that can be used to implement a range of complex transport protocol offloads, such as TCP:

- **Fast-path processing.** To be efficient, we separate common-case code paths from exceptions and focus the SmartNIC offload only on the common case. In particular, fast-path processing is one-shot for each network packet. Packets are never queued in NIC memory. This leads to vastly simpler buffer management as we free buffers immediately after processing.
- **Modularity.** To provide flexibility, we organize per-packet processing tasks into fine-grained, reusable modules that keep private state and communicate explicitly. New functionality and offloads can be implemented as new packet processing modules, simplifying development and integration.
- **Leverage fine-grained parallelism.** For high performance, we organize per-packet processing modules as a data-parallel computation pipeline and minimize co-ordination among components by leveraging private state. We specialize FPCs to each stage, allowing us to fully utilize each FPC's hardware acceleration and processing capabilities. We employ hardware sequencing and reordering to support parallel, out-of-order processing of some pipeline stages, even for the same TCP connection, while enforcing in-order segment delivery where necessary.
- **Near-memory computing.** To support connection scalability and to minimize latency, we use the various near-memory compute facilities of the SmartNIC architecture, placing state in FPC-local memory nodes to balance memory bandwidth and capacity utilization, as well as access latency.

3.3 Offload Architecture

TASNIC splits the network stack into three components: fast-path offloaded to the NIC, slow-path on the host, and a POSIX network sockets library (libTASNIC) dynamically linked to unmodified applications. Fast-path implements the core of the transport logic and is responsible for segmentation and transmission, loss detection and recovery, rate control, payload transfer to/from socket buffers, and application notification. The slow-path handles connection and application

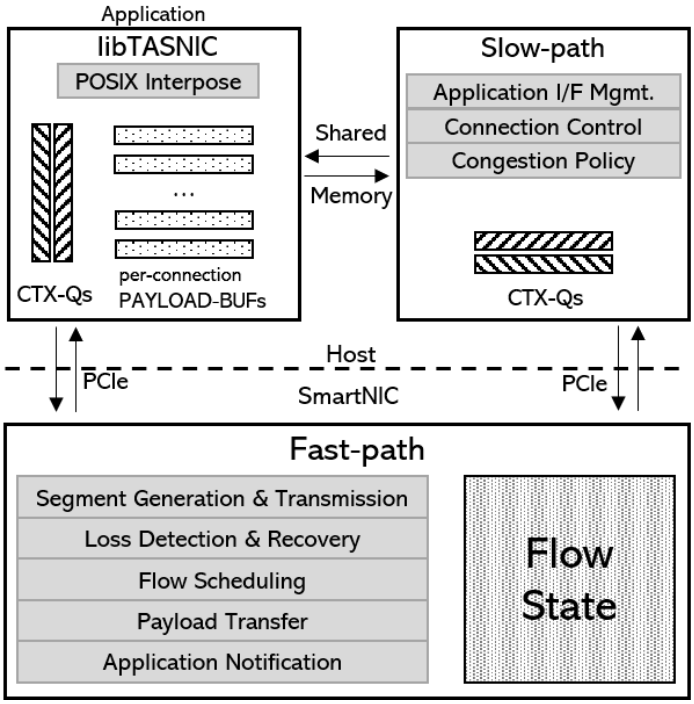

Figure 3.1: TASNIC overview.

management, timeouts, congestion control policies, and uncommon recovery scenarios. Figure 3.1 shows the high-level design of TASNIC.

3.3.1 Interfaces

libTASNIC, slow-path, and fast-path use pairs of *context queues* to communicate, one for each communication direction. Context queues always use shared memory on the host, but communication among fast-path and host components naturally requires traversing the PCIe interface. We use scalable PCIe communication techniques [39] to avoid excessive PCIe traffic by polling 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 memory-mapped IO to a NIC doorbell. On receiving the doorbell, the fast-path fetches new entries from the context queue using DMA and frees context queue entries by writing back null-descriptors using DMA. Applications reclaim context queue entries by polling for null descriptors. In the other direction, fast-path adds new descriptors

to the context queue using DMA. Applications poll for new entries by testing for non-null entries at the queue head. Applications replace consumed entries with null-descriptors. They also ring a doorbell using MMIO to notify the fast-path of freed entries. Slow-path context queues also use the mechanism described above.

libTASNIC is free to register several context queue pairs with the slow-path and fast-path. While it is possible to share a context queue among cores via locks, scalability is achieved by registering per-core queues. libTASNIC interacts with the slow-path for connection management. The slow-path handles the three-way handshake and sets up the flow state in the fast-path. Likewise, the slow-path notifies the application of any pending connection requests in the backlog. Each connection has its private receive and transmit payload buffers. libTASNIC appends data for transmission into the per-connection TX buffer and notifies the fast-path using a context queue. Incoming data is appended to the RX buffer by TASNIC and the application is notified via an associated context queue.

3.4 Slow-path

The slow-path is primarily responsible for control and management mechanisms including connection control, application interface management and congestion control policy. These policy mechanisms have non-constant per-packet overheads or complex code paths that occur infrequently or require utilities unsupported by the NIC hardware. Broadly, our slow-path design closely follows the structure of TAS [20].

3.4.1 Congestion Control

Congestion control protocols routinely use floating point and division operations to compute the updated transmission rate based on observed congestion indicators. For example, calculating ECN-Ratio in DCTCP [1] and RTT-gradient in Timely [31] requires both division and floating point computation. As noted previously (§2.2.8), NFP-4000 instruction set does not support them. Thus, we choose locate the congestion control policy in the slow-path running on the host. This is also important for supporting flexibility in modifying existing or implementing

new congestion control policies. As a result, TASNIC provides a generic framework to implement different congestion control algorithms. The slow-path runs a control loop over the set of active flows to compute a new transmission rate periodically. The interval between each iteration of the loop is determined by the RTT of the flow. In each iteration, the slow-path reads per-flow congestion control statistics from the NIC to calculate a new rate for the flow. The rate is then forwarded to the slow-path via the context queues for enforcement. We also monitor restransmission timeouts in the control iteration. TASNIC implements both DCTCP and Timely, but DCTCP is configured as the default congestion control algorithm.

3.4.2 Retransmissions

We handle retransmission timeouts in the slow-path. When collecting congestion control statistics in the post-processing block in the fast-path, we also track if the flow has unacknowledged data in the transmission buffers. If the flow has unacknowledged data for more than a constant multiple of round-trip times and does not receive any acknowledgment in this duration, the slow-path adds a retransmission event for this flow on the context queues.

3.4.3 Connection Control

Connection setup and tear-down involve complex code paths with multiple stages such as ARP resolution, port allocation, payload buffer allocation and TCP handshake. Fast-path forwards packets with the SYN flag to the slow-path. The slow-path notifies the application of incoming connections on listening ports. If the application decides to accept the connection, the slow-path executes the three-way handshake, allocates the payload buffers and sets up the flow-state in fast-path. Similarly, applications issue `connect()` calls to the slow-path to establish new connections. On connection tear-down, the slow-path disables the connection in the fast-path and removes the corresponding state.

3.4.4 Application Interface Management

When new applications connect to TASNIC, they first communicate with slow-path to setup the context queue interfaces. The slow-path is responsible for memory allocation and setting up the doorbells in the fast-path for the context queues.

3.5 Application Library

The user-space application library interposes on the POSIX socket API calls from the application. It forward the control and management calls to the slow-path and data transfer calls to the fast-path on the context queues. We ensure that the overheads of the application library are minimal and keep the operations as simple as possible. Finally, applications do not have control on any of the core TCP operations such as sending the acknowledgment or congestion control for reasons of security and performance isolation.

3.5.1 Low-level interface

TASNIC also offers a low-level interface that is more economical than the sockets interface. If application compatibility is not a priority, applications can utilize the low-level interface to extract more performance. In the low-level interface, applications directly access data from the DMA-capable payload buffers and interact with the fast-path by adding entries directly on the context queues.

3.6 Fast-path

In-order packet delivery is necessary for good TCP performance. Out of order deliveries lead to packet drops, complex reassembly code paths, retransmissions, and unnecessary acknowledgments. High-performance parallel packet processing is at odds with TCP's ordering requirements as parallel processing often makes no ordering guarantees.

To provide high performance while hiding the overhead of high-latency packet processing and memory operations, TASNIC's fast-path has to leverage available

parallelism within the TCP data-path. However, TCP's in-order delivery requirement and associated connection state management requires at least some in-order processing. To minimize it, we make use of three design principles: 1) Fine-grained parallelism, 2) event sequencing and reordering, and 3) near-memory processing. The first requires us to partition the TCP data-path and associated connection state such that parallel sections do not need to synchronize to process packets. The second requires us to carefully determine when to sequence events, when to allow out of order processing, and when to reorder events according to the previously determined sequence. We also need efficient sequencing and reordering mechanisms. The third requires us to localize state within the SmartNIC's memory topology and to use near-memory hardware primitives, such that critical sections can finish with minimal execution latency.

3.6.1 Fine-grained parallelism

We decompose the TCP data-path into a three-stage parallel pipeline: *pre-processing*, *protocol*, and *post-processing*. To parallelize across connections, we replicate the entire pipeline on four islands and use a hash function to distribute load. FPCs dedicated to a pipeline stage can utilize available local memory and its near-memory hardware primitives to operate on their portion of the connection state faster (§3.6.4).

3.6.2 Fine-grained state partitioning

To enable fine-grained parallelism, we partition connection state across pipeline stages (cf. §3.1). 15B of pre-processor state contains MAC, IP, and TCP port numbers used for connection identification. 43B of protocol state contains TCP windows, sequence and acknowledgment numbers, and host payload buffer state. 51B of post-processor state contains host payload buffer and context queue addresses, and fast-path congestion control state. In aggregate, each TCP connection has 108 bytes of state, allowing us to manage thousands of connections in the fast-path. In particular, we can manage 16 connections per protocol FPC, 512 connections per flow-group, and 16K connections in the EMEM cache. Using all of EMEM, we can support up to 8M connections. The slow-path interacts with this state through connection management events that traverse the pipeline.

Table 3.1: Per-flow state (108 bytes).

Field	Bits	Description
Connection:		
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
Application-interface:		
opaque	64	App-defined flow id
context	16	Context-queue id
rx|tx_base	128	RX/TX buffer base
rx|tx_size	64	RX/TX buffer size
Protocol:		
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
Congestion Control:		
cnt_ackb|ecnb	64	ACK'd and ECN bytes
cnt_fretx	8	Fast-retransmits count
rtt_est	32	RTT estimate
rate	32	TX rate

3.6.3 Event sequencing and reordering

TCP requires that receive (RX) and transmit (TX) events of the same connection are delivered in order. Existing TCP stacks satisfy this requirement via in-order processing of these events, limiting single-connection throughput to the speed of the processor executing the TCP state machine.

To improve throughput with limited FPC compute speed, we sequence and reorder packets to extract more packet processing parallelism, while delivering packets in order. In particular, we can assume that packets are delivered in-order to the MAC interface and that applications request payload delivery on the PCIe interface in order in the common case. This allows us to assign a sequence number to each incoming RX and TX event, process them out of order, and re-order them later. We use the NBI sequencer to assign a sequence number to each incoming RX and TX packet. The pre-processing and post-processing stages can operate on each packet independently and out-of-order. Only the protocol stage requires in-order processing. We use the GRO to re-order packets entering the protocol stage. The GRO uses the NBI sequence number to deposit the event descriptor to the appropriate flow-group work-queue in order.

Within the protocol stage, FPC threads must synchronize to respect the order when dequeuing events and ensure strict serialization of their connection state. We use signals to orchestrate FPC threads to ensure sequential execution of events while also retaining parallelism by overlapping thread execution with asynchronous fetches from the work-queue. Other TCP state machine transitions, such retransmissions and socket operations, are mutually commutative and may also be intermingled with packet transmission and reception. We process them in any order.

3.6.4 Near-memory processing

An order of magnitude difference exists in the access latencies of different memory levels. For performance, it is critical to maximize access to local memory and to use near-memory hardware primitives. To do so, we build specialized caches at multiple levels in the different pipeline stages.

Caching. We use each FPC's CAM to build 16-entry fully-associative local memory caches for all pipeline stages. We evict entries based on least-recent-use (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 storage is in EMEM. The protocol stage is responsible for the management of the caches. Having a single module manage the cache index eliminates synchronization overheads. When a pipeline stage receives an event associated with a connection, it fetches the relevant state into its local memory either from CLS or from EMEM. If necessary, it also evicts entries from the caches to make space. Note that the local memory and the CLS caches are mutually exclusive. We allocate connection identifiers in such a way that we minimize collisions on the direct-mapped CLS cache.

Active connection database. 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 relies on the database to lookup connection state on packet ingress. It computes a CRC-32 hash on the 4-tuple to locate the connection identifier 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 and island mapping. Figure 3.2 shows how pipeline stages and other relevant software modules map onto FPCs on the NFP-4000 islands. We use four of the five general-purpose islands for the protocol-processing pipeline (called *protocol islands*). Each island manages a *flow-group*. While the protocol and post-processing FPCs are local to the flow-group, the pre-processors handle any incoming traffic. Pre- and post-processing involve multiple high-latency CPP commands. To hide their latency and increase processing bandwidth, we assign 4 FPCs to each of these stages per flow-group. To avoid inter-FPC synchronization on connection state, there is a single protocol FPC per flow-group and pre-processors determine and forward traffic to the appropriate protocol FPC. Each island retains 3 unassigned FPCs that can run flexible user-programmed policies, tracing, and accounting applications (§5.4). On the remaining general-purpose island (called *service island*), we host data-path functionality that is not organized by flow-group.

We describe the functionality in §3.6.4. The GRO and BLM FPCs are located on a special-purpose island that contains miscellaneous hardware-accelerated functionality. For island-local interactions among modules, we use CLS ring buffers. CLS supports the fastest intra-island producer-consumer mechanisms. Among modules on different islands, we rely on work-queues in IMEM and EMEM.

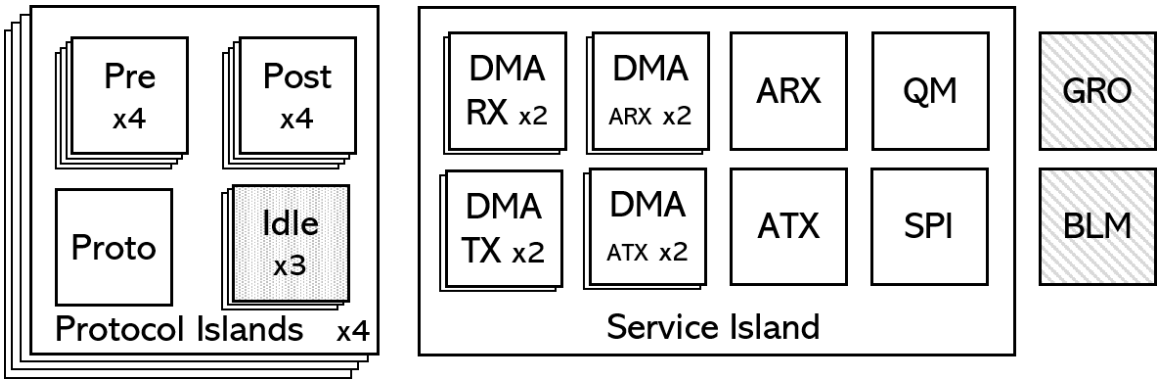

Figure 3.2: Block layout on NFP-4000 architecture.

Service Island

The service island contains application context managers (ARX, ATX), a queue manager (QM), a slow-path interface (SPI), and DMA managers for RX, TX, ARX, and ATX. DMA managers are replicated to hide various host communication latencies. The number of FPCs assigned to each functionality is determined such that no functionality may become a bandwidth bottleneck. We describe each in more detail in this section.

Application context managers (ATX/ARX). To hide the latency of PCIe transactions when interacting with applications, we separate this functionality into application context manager FPCs. ARX manages the application RX context queues. When the post-processing stage produces a new descriptor to enqueue, ARX allocates an entry on the context queue and increments the tail. ARX serves as the serialization point for the RX context queues and performs DMA to transfer the descriptor to the host. ARX also periodically monitors the doorbell updated by the application to obtain the most recent queue head. ATX polls the application TX

context queue tail doorbells to detect bumps to the payload buffer. Doorbells are updated by the application to indicate new context queue entries. ATX performs DMA to read the descriptor and to write-back a null descriptor in its place. Finally, ATX generates an AC event for the pipeline.

Queue manager (QM). The queue manager is a work-conserving flow scheduler that obeys transmission rate-limits configured by the slow-path's congestion control policy. The QM keeps track of how much data is available for transmission and the configured rate for each connection. Transmission rates are stored in EMEM and are directly updated by the slow-path using memory-mapped IO. The QM generates TX events for flows with a non-zero transmit window.

We implement our flow scheduler based on Carousel [46]. Carousel schedules a large number of flows using a time wheel. Based on the next transmission time, as computed from rate limits, we enqueue flows into corresponding slots in the time wheel. As the time slot deadline passes, the QM schedules each flow in the slot for transmission (§3.6.5). To conserve work, the QM only adds flows with non-zero transmit window into the time wheel and bypasses the rate limiter for uncongested flows. These flows are scheduled round-robin.

We implement Carousel using hardware queues in EMEM. Each slot is allocated a hardware queue. To add a flow to the time wheel, we simply 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 part of the NFP-4000 instruction set. Thus, the slow-path computes transmission intervals in cycles/byte units from rates and programs them to NIC memory. This enables the QM to compute the time slot using only multiplication.

Slow-path interface (SPI). The SPI manages context queues to the slow-path. Exception packets filtered by the pre-processor are copied to the host slow-path by SPI. The SPI also issues retransmissions computed by the congestion control policy received on the slow-path context queue.

DMA managers. The PCIe island on the NFP-4000 exposes a pair of DMA transaction queues, one for each direction. FPCs can issue 128 asynchronous DMA operations on each queue. We design the DMA managers to be highly parallel using hand-crafted assembly code to maximize the number of in-flight DMA operations. Each type of DMA manager (RX, TX, ATX, ARX) maintains its own quota of outstanding DMA operations, ensuring that different types of DMA commands receive adequate PCIe resources.

3.6.5 Data-path processing

The TASNIC data-path pipeline processes four kinds of events:

- **RX**: Each incoming packet on the MAC is processed by the NBI, which triggers an RX event. The pipeline must perform segment reassembly—advance the relevant connection state, determine the segment's position in the host receive payload buffer, generate an acknowledgment to the sender, and, finally, notify libTASNIC. The incoming segment often also acknowledges previously transmitted data. In this case, the pipeline must inform libTASNIC to free the data in the transmit payload buffer.
- **AC**: libTASNIC informs the fast-path through context queue bumps when new data is available in the transmit payload buffer and when space in the receive payload buffer is freed. These trigger AC events. The pipeline must update the connection's receive and transmit windows accordingly.
- **TX**: TX events are generated by the queue manager (§3.6.4), based on per-connection rate limits determined by the TCP congestion control policy. In response, the pipeline prepares a TCP segment for transmission, fetching its payload from a host transmit payload buffer and sending it out the MAC.
- **RETX**: The slow-path generates RETX events upon retransmission timeouts. The fast-path resets the connection's transmission state to start retransmitting from the earliest unacknowledged sequence number.

We now detail how events traverse the pipeline (cf. Figure 3.3).

Receive (RX)

Red in Figure 3.3.

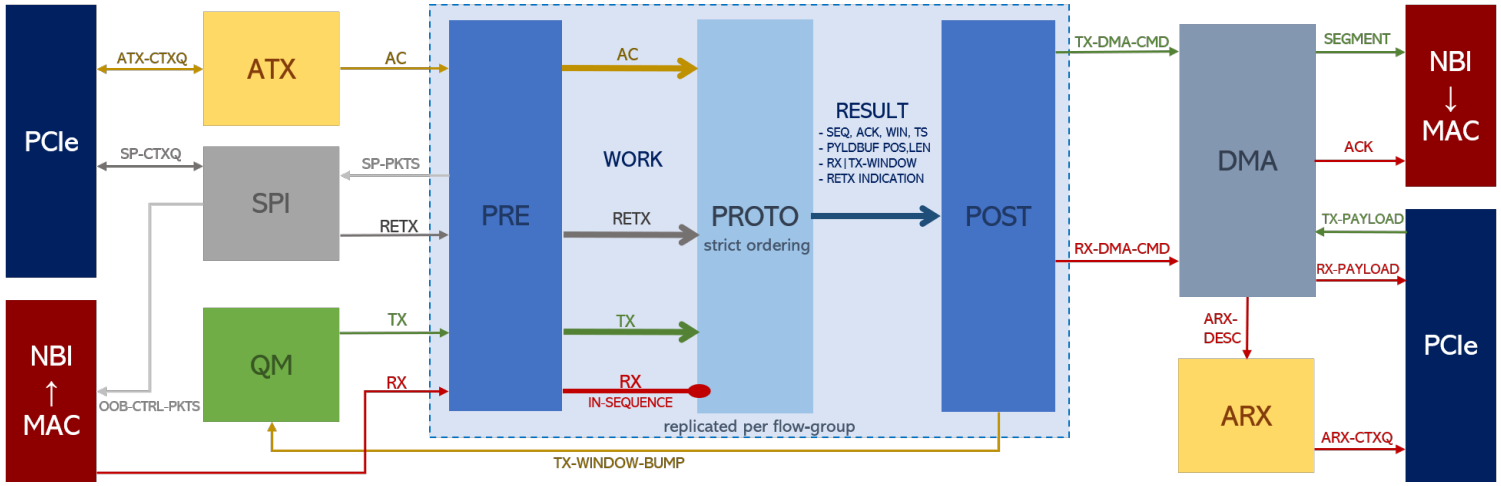

Figure 3.3: Fast-path parallel pipeline architecture.

Common data and control packets belonging to established connections can be processed by the fast-path. These *fast-path packets* may have any of the ACK, FIN, PSH, ECE, and CWR flags and they may have the timestamp option. Packets that are not fast-path packets are filtered and forwarded to the slow-path (SP-PKTS).

Pre-processing. Pre-processor FPCs poll the NBI for incoming packets and internally generate RX events when a packet is received (RX). The pre-processor FPC first validates the packet header. Non-fast-path packets are filtered and forwarded to the slow-path. For fast-path packets, the pre-processor generates a *header summary*—a digest of packet header fields required by later pipeline stages. In anticipation of sending a TCP acknowledgment, the pre-processor also swaps source and destination header fields in the buffered packet for reuse in post-processing.

Next, the pre-processor fetches the connection identifier based on the packet's 4-tuple using the active connection database (§3.6.4). Connection state is identified by the connection identifier throughout the pipeline. A CRC32 hash on the 4-tuple is used to assign the connection to a flow-group. The pre-processor deposits the RX event descriptor containing the header summary and connection identifier on the work-queue of the protocol stage for the flow-group.

Protocol. Protocol FPCs receive RX events on their flow-group work-queue, which are deposited there via the GRO in order of NBI arrival. The protocol FPC updates the connection's sequence and acknowledgment numbers, as well as the transmit window, based on the header summary. The protocol FPC determines the segment's position in the connection's host receive payload buffer, trimming the payload to fit the receive window if necessary. It prepares a receive context queue descriptor for libTASNIC notification. Finally, it generates a snapshot of relevant local connection state (RESULT) and places it on the post-processing work-queue.

Out-of-order arrivals and packet drops need special treatment. We track one out-of-order interval in the receive window, allowing the protocol stage to perform reassembly directly within the host payload buffer. We attempt to merge out-of-order data with the interval when possible and write the data to the payload buffer. If the merge fails, we drop the packet and generate acknowledgments with the expected sequence number to trigger retransmissions at the sender. When

a segment fills a hole, the fast-path resets the out-of-order interval and notifies the application of the payload bump.

Post-processing. The post-processor prepares the acknowledgment packet from the RESULT. Acknowledgments also provide explicit congestion notification (ECN) feedback and accurate timestamps for RTT estimation, which the post-processor determines and inserts. It collects congestion control and transmit window statistics for use by the slow-path and queue manager, which it forwards there.

The post-processor generates a DMA command to the RX DMA manager that describes the physical address of the host buffer, payload offset, and payload length (RX-DMA-CMD). If libTASNIC is to be notified, the post-processor allocates a context queue descriptor with the appropriate notification and forwards it along. The RX DMA manager first enqueues payload DMA descriptors to the PCIe block (RX-PAYLOAD). After payload DMA completes, the RX DMA engine sends the context descriptor to the ARX FPC (ARX-DESC), which issues the context queue descriptor to the host to notify libTASNIC of new payload (ARX-CTXQ) and returns the internal descriptor buffer to the pool. Simultaneously, the RX DMA manager sends the prepared acknowledgment to the NBI (ACK). We utilize the NBI transmission sequencers to send acknowledgments in sequence of packet arrival. The NBI frees packet buffers after transmission. This ordering is necessary to prevent the application and the peer from receiving notifications before the data transfer to the receive payload buffer is complete.

Application Event (AC)

The AC path is shown in Yellow in Figure 3.3.

ATX manages a buffer pool in NIC memory for context queue descriptors. Context queue entries (ATX-CTXQ) are read into buffers allocated from this pool. We limit the pool size to flow-control the application context queue. Without flow-control, the application may flood the context queue to overwhelm the pipeline.

Pre-processing. The pre-processor reads AC events forwarded by the ATX module, determines the flow-group, and routes it to the appropriate flow-group work-queue.

Protocol. In response to an AC event, the protocol stage updates connection receive and transmit windows. Finally, if the AC event contains a connection-close indication, the protocol stage marks the connection as FIN.

Post-processing. When the transmit window expands due to the application adding data for transmission, the post-processor updates the queue manager (TX-WINDOW-BUMP). It then returns the context descriptor to the ATX's pool after processing the AC event.

Transmit (TX)

The TX path is shown in Green in Figure 3.3.

Pre-processing. The pre-processor receives TX events from the queue manager schedule queue. The pre-processor allocates a packet buffer and prepares Ethernet and IP headers. Then, it enqueues the event to the flow-group work-queue.

Protocol. Upon TX, the protocol stage assigns a TCP sequence number based on connection state and determines the transmit offset in libTASNIC's TX payload buffer.

Post-processing. The post-processor issues a DMA command and a context queue descriptor with the libTASNIC transmission notification to the TX DMA manager (TX-DMA-CMD) and updates the QM (TX-WINDOW-BUMP). The TX DMA manager first enqueues payload DMA descriptors to the PCIe block (TX-PAYLOAD), then issues the segment to the NBI (SEGMENT). After payload DMA completes, the TX DMA engine sends the context descriptor to the ARX FPC (ARX-DESC), which issues the context queue descriptor to the host to notify libTASNIC of transmission (ATX-CTXQ) and returns the internal descriptor buffer to the pool.

Retransmit (RETX)

The RETX path is shown in Gray in Figure 3.3.

Retransmissions are usually triggered by the host slow-path in response to timeouts to the SPI (SP-CTXQ) and forwarded via the pre-processing stage to the

appropriate flow-group. We opt for a simple go-back-N retransmission strategy. The protocol stage resets the transmission state to the last acknowledged sequence number. The protocol stage also tracks incoming duplicate ACKs to trigger fast retransmissions. The remainder of the RETX path is identical to the TX path for the segment to retransmit.

Chapter 4

Implementation

TASNIC is implemented in 18008 lines of C code. The fast-path primarily comprises 5801 lines of MicroC code. The TASNIC application library is 4620 lines whereas the slow path is 5549 lines of C code. We use the NFP compiler toolchain version 6.0.4.1 to compile the firmware.

4.1 Fast-path

As mentioned previously, the TASNIC fast-path is offloaded as a firmware to the Netronome Agilio CX40. We implement the fast-path primarily in MicroC - a variant of C customized for the NFP-4000 architecture. However, to gain performance we handcraft parts of the protocol and the DMA block in assembly. Furthermore, we extensively annotate the program to ensure that no variables get spilled to higher latency memories during compilation.

4.2 Slow-path

The slow-path runs as a process on the host. We implement a user-mode driver on top of the `igb_uio` driver shipped by DPDK [7]. In order for the fast-path to DMA payload directly into application memory, we allocate physically contiguous buffer using 1G hugepages. As hugepages cannot be swapped out, the physical to virtual address mapping remains valid in the lifetime of TASNIC. The slow-path maps a small pool of 1G hugepages during startup and allocates the payload buffer and context queue allocations as fragments of individual hugepages.

4.3 Limitations

Interrupts. libTASNIC and Slow-path operate in polling mode to read entries from the context queues. MSI-X interrupts may be used to send interrupts to applications from the fast-path to notify applications. This will enable the applications to sleep when they are inactive.

Memory allocation. We allocate physically contiguous context queues and payload buffers on 1G hugepages. In the future, we can use IOMMU to eliminate the allocation of physically contiguous memory.

Congestion Control in Slow-path. Presently, the congestion control policy runs on the host as a part of the slow-path. As mentioned previously, this is primarily because of the absence of floating-point and division operations in the NFP-4000 instruction set. The Netronome Agilio CX40 SmartNIC also features an ARM11 core which boots Linux. We plan to move the congestion control and retransmission control loop to the ARM processor.

Delayed ACK. We do not implement delayed acknowledgement protocols in the fast-path. Delayed ACK will have a significant impact on the throughput of large flows as the number of packets as it reduces the number of packet to process in the fast-path.

Chapter 5

Evaluation

To understand the performance of TASNIC, we evaluate it on the following benchmarks:

- **RPCs.** How does TASNIC's throughput and latency for short RPCs fare against state-of-the-art systems? Does TASNIC provide high throughput for long RPCs? To how many simultaneous connections can TASNIC scale (§5.1)?
- **Loss and Congestion.** How does TASNIC perform under loss and congestion (§5.2)?
- **Key-Value stores.**Does TASNIC improve throughput and latency of Memcached? Does TASNIC interoperate well with clients using other network stacks (§5.3)?
- **CPU savings.** How many CPU cycles does offload save? What is the impact on performance (§5.3)?
- **Flexibility.** Is it simple to add offloads to TASNIC (§5.4)?

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 40Gbps and Chelsio Terminator T62100-LP-CR 100Gbps network adapters. We use one of the above 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 100Gbps NICs, whereas the Xeon E3 machines have 82599ES 10 Gbps NICs. The machines are connected to a 100Gbps Ethernet switch.

Baseline. We compare the performance of TASNIC against in-kernel Linux stack, Chelsio ToE full-offload solution and TAS kernel-bypass stack. In all cases, we use the same application binary to run the benchmarks.

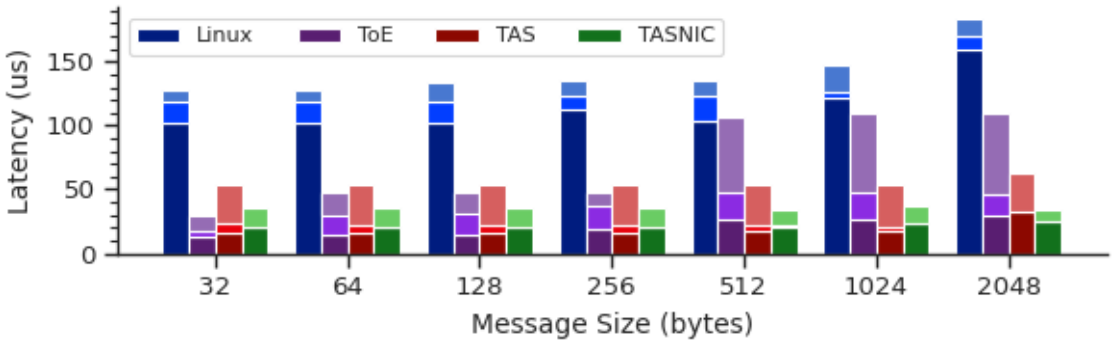

Figure 5.1: Median, 99p and 99.99p latency for Short RPCs.

5.1 Remote Procedure Calls (RPCs)

Latency: Short RPCs. To understand the latency of TASNIC, we run a simple event-based RPC echo server. The client establishes a single connection to the server, leaves a single RPC in-flight and waits from the response from the server before initiating the next RPC. Figure 5.1 shows the median, 99 and 99.99 percentile round-trip latency 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 < 256B, TASNIC's median latency (20 us) is 1.4× Chelsio's median latency (14 us) and 1.25× TAS's median latency (16 us). TASNIC's pipelined design and slow FPCs increases the latency for this case. However, TASNIC demonstrates predictable performance with up to 3.2× smaller tail compared to Chelsio and nearly constant per-packet overhead as the packet size increases. In case of a 2KB message which is higher than the TCP maximum segment size, TASNIC's latency distribution remains nearly unchanged. TASNIC's intra-connection parallelism is able to hide the processing latency of multiple transmissions, providing 0.78× the latency and 0.5× the tail-latency of TAS for this case.

Throughput: Short RPCs. We now run 20 server threads with 20 client connections. RPC size is 64B. Each connection keeps as many as 32 requests to the server in-flight. We show the RPC rate and the latency distribution in Table 5.1. Bidirectionally, TASNIC processes 26.38 million RPCs per second, achieving 1.73× TAS

Table 5.1: Throughput and Latency of Short RPCs.

Stack	Rate		Latency (us)		
	Gbps	mOps	50p	99p	99.99p
Linux	0.93	1.81	249	1014	3089
ToE	0.45	0.87	48	19177	36267
TAS	3.90	7.62	52	141	2305
TASNIC	6.75	13.19	46	59	72

throughput. TASNIC also retains at least two orders of magnitude lower 99.99p tail latency versus other stacks, demonstrating that TASNIC's parallel pipeline is scalable both across messages on the same connection and across connections. Chelsio performance drops significantly for this benchmark. While Chelsio offloads TCP processing to the NIC, it still retains a system call interface, reintroducing kernel overhead. Furthermore, we observe that the overhead of Chelsio's `epoll()` implementation increases by orders of magnitude with increasing connections and messages.

Fine-grained parallelism. To study the impact of our parallelism design principles, we repeat the echo benchmark with 64 connections, with each connection leaving a single 2KB RPC in-flight. Table 5.2 shows the performance impact as we progressively add parallelism. In the baseline, we run the entire TCP processing to completion on the SmartNIC before processing the next event. Pipelining improves the performance by 46× over the baseline. As we enable 8 threads on the FPCs (2.25× gain), we hide the latency of asynchronous operations and improve FPC utilization. Next, we replicate the pre-processing and post-processing blocks, leveraging sequencing and reordering for correctness, to extract 1.35× improvement and finally, with four flow-group islands, we see a further 2× improvement.

Pipelined RPC. We compare pipelined throughput for different RPC sizes by running a single-threaded server, consuming RPCs on 128 connections, produced by 16 client threads. The server waits for an artificial delay of 250 or 1,000 cycles after each RPC to simulate application processing. We quantify RX and TX performance separately, by switching RPC consumer and producer roles among clients

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

Table 5.2: Parallelism breakdown analysis.

and servers.

Figure 5.2 shows the results. TASNIC provides up to 4× better throughput over Linux and 5.3× better throughput over Chelsio when receiving for 250 cycles of processing overhead. For 2KB message size, both TAS and TASNIC reach 40G line rate, whereas Linux and Chelsio barely reach 10G and 7G, respectively. When sending packets, the difference in performance between Linux and TASNIC is starker. TASNIC 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/message. Performance of TAS and TASNIC track closely for all message sizes. This is expected as the single application server core is saturated by both network stacks.

Throughput: Large RPCs. In this setup, a single client transfers a large RPC message to the server which is then echoed back to the client. As shown in Figure 5.3, TASNIC performs 20% better than Chelsio, the next best performing stack. Other stacks cannot parallelize per-connection processing, leading to limited throughput. TASNIC currently acknowledges every incoming packet. For bidirectional flows, this quadruples the number of packets to be processed per second. Implementing delayed ACKs will improve TASNIC's performance further for bidirectional RPCs. We experimentally confirm that TASNIC without delayed ACKs is able to saturate the 40Gbps line rate for unidirectional flows for every message size shown in the figure.

Connection Scalability. We establish an increasing number of RPC client connections from 5 machines to a multi-threaded echo server. Each connection leaves

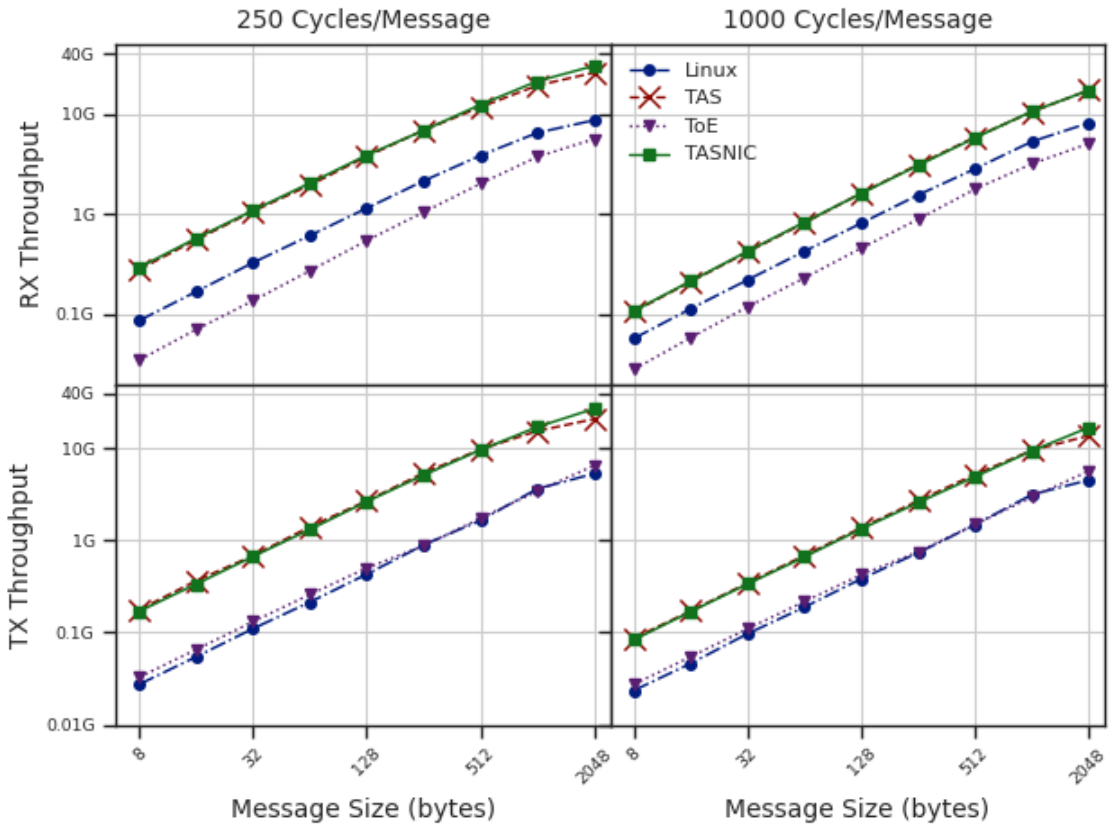

Figure 5.2: Pipelined RPC throughput, varying per-RPC delay and size, for a single-threaded server.

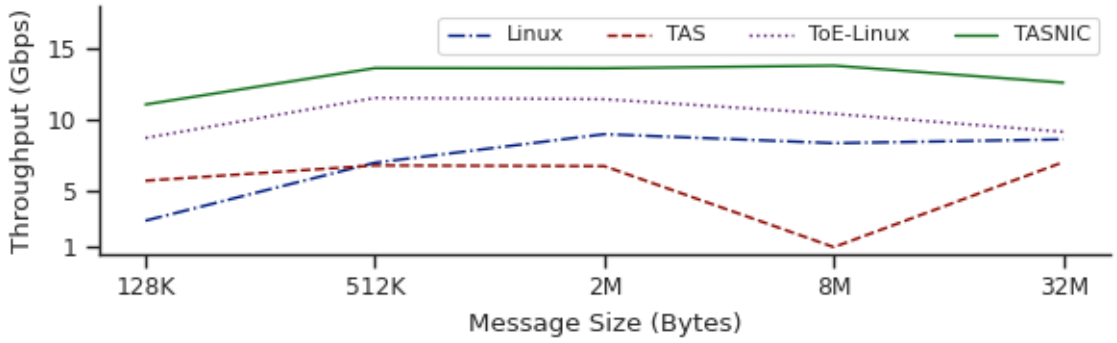

Figure 5.3: Throughput with varying message size for Large RPCs.

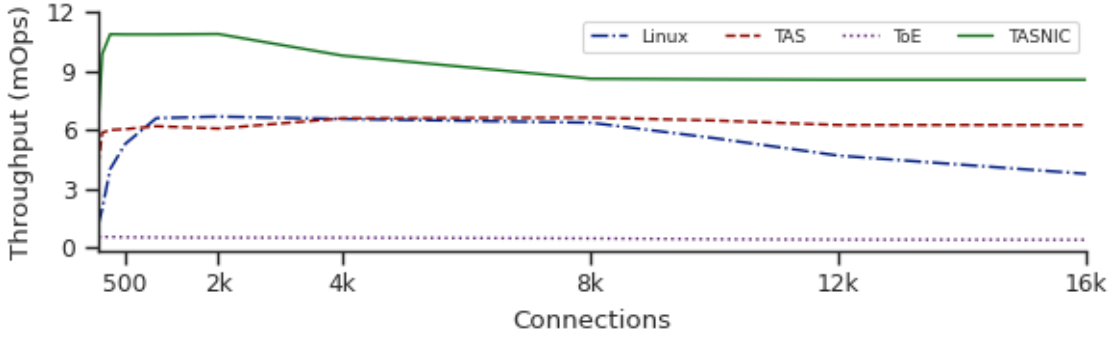

Figure 5.4: Connection scalability benchmark.

a single 64B RPC in-flight. Figure 5.4 shows the throughput as we vary the number of connections. This workload is very challenging for TASNIC as it exhausts fast memory, prevents per-connection batching, and causes a cache miss at every pipeline stage for every packet. Up to 2K connections, TASNIC shows a throughput of 1.6× Linux and 1.8× TAS. TASNIC is able to cache 2K connections in the CLS memory. Beyond this, the protocol stage must move state between local memory, CLS, and EMEM. EMEM's SRAM cache is increasingly strained as the number of connections increases. TASNIC's throughput declines by 20% as we hit 8k connections and plateaus beyond that. TAS's fast-path exhibits better connection scalability, as it is able to prefetch state efficiently, while Linux's throughput declines significantly. Chelsio displays poor performance for this workload, as `epoll()` overhead dominates again. Further, Chelsio is unable to distribute connections to server cores uniformly when using the `SO_REUSEPORT` option.

5.2 Packet Loss and Congestion Control

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 128 flows as we vary the loss probability, shown in Figure 5.5. TASNIC's throughput under 5% losses is at least an order of magnitude better than the other stacks. Linux and Chelsio are able to withstand higher loss rates as they implement more sophisticated reassembly and recovery algorithms, including selective acknowledgments. TASNIC and TAS implement go-

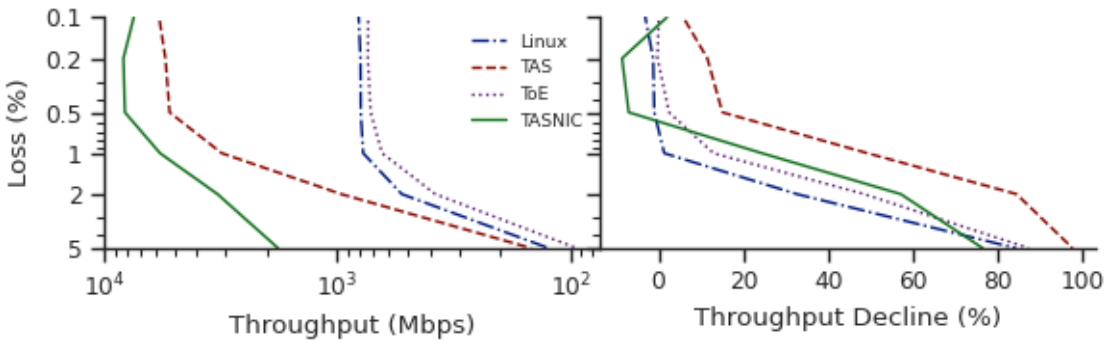

Figure 5.5: Throughput, varying packet loss rate.

Stack	# connections		
	128	**512**	**2048**
Linux	0.96	0.47	0.36
TASNIC	1.0	0.98	0.98

Table 5.3: Jain Fairness Index (JFI) at line rate.

back-n recovery. TASNIC's behavior under loss is still better than TAS. TASNIC processes acknowledgments on the NIC, triggering retransmissions sooner, and its predictable latency, even under load, helps TASNIC recover quicker from packet loss. We note that RDMA tolerates up to 0.1% losses [32], while eRPC falters at 0.01% loss rate [17].

Rate-limiting. To show the efficiency and connection scalability of TASNIC's Queue Manager (§3.6.4), we measure the distribution of connection throughputs of bulk flows between two nodes operating at line rate for 60 seconds. Figure 5.6 shows the median and 99th percentile throughput of TASNIC and Linux as we vary the number of connections. For TASNIC, 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 as shown by the Jain Fairness Index in Table 5.3. Above 1024 connections, the median throughput of Linux is worse than the 99th percentile of TASNIC.

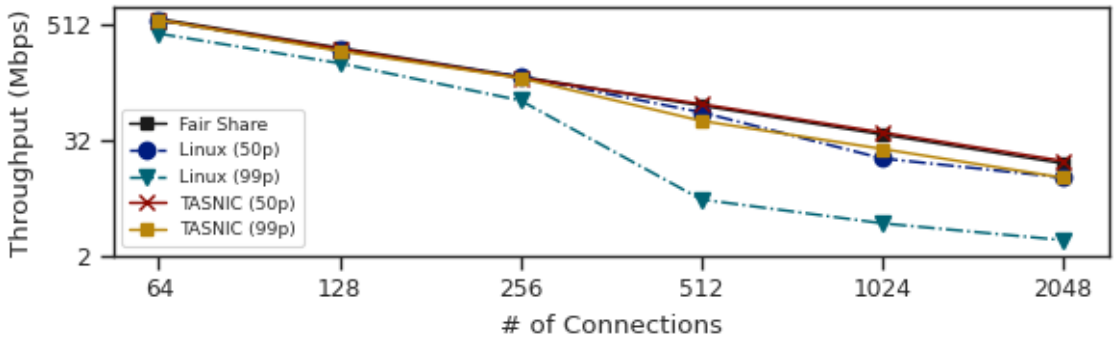

Figure 5.6: Distribution of throughputs at line rate.

# conns.	**Tpt.** (Gbps)	**Lat.** 99.99p (us)	**JFI**
16	9.51	5,998	0.98
16 (no cc)	9.47	11,586	0.95
64	9.51	10,753	0.96
64 (no cc)	9.23	44,397	0.73
128	9.48	13,746	0.99
128 (no cc)	8.96	64,250	0.53

Table 5.4: Congestion control under incast.

Congestion. We simulate incast by enabling traffic shaping on the switch to restrict server throughput to 10Gbps to build up the queues and configure WRED to perform tail drops when the switch buffer is exhausted. In this experiment, the client transfers 64KB RPCs and the server responds with a 32B response on each connection. As shown by table 5.4, TASNIC's is able to achieve line rate, maintain a low tail latency, and ensure fairness among flows under congestion. Disabling congestion control leads to bursty traffic causing excessive drops. This inflates tail latency by 4.9× and skews fairness by 2×.

5.3 Key-value Stores

Application scalability. We run a Memcached server, varying the number of server application cores. For each case, we run memtier_benchmark [45] on as

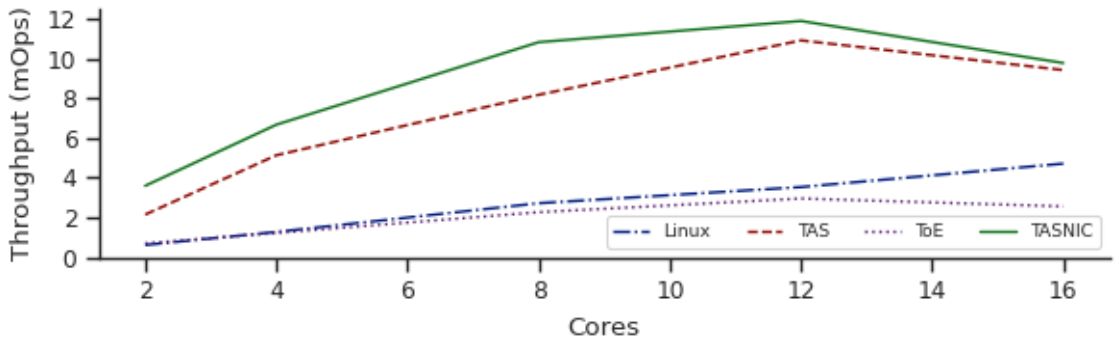

Figure 5.7: Throughput with varying cores on Memcached server.

many clients as necessary to arrive at the latency-throughput knee. TAS uses as many additional fast-path cores as necessary (not shown). Figure 5.7 shows that TASNIC and TAS perform similarly—both use per-core context queues for scalability. Memcached scales to 12 cores, where it is bottlenecked by application-level locks. Linux and Chelsio are slow for this workload due to system call overheads. Chelsio additionally suffers from locking overheads in the kernel and stops scaling at 12 cores.

Network stack interoperability. To show TASNIC's interoperability with the other network stacks, we repeat a single-threaded version of the same Memcached benchmark for all server-client network stack combinations. Latency distributions are shown in Figure 5.8. We can see that TASNIC consistently provides the lowest latency across all stack combinations, except for the Chelsio server. Chelsio has lower latency with Chelsio and TAS clients up to the 60th percentile, beyond which a TASNIC client has lower latency. We show 99.99p latency for different combinations in Figure 5.9.

Offload. Offloaded CPU cycles may be used to do more application work. We quantify these savings by running Memcached on a server limited to 2 cores. We saturate the server cores using multiple clients and measure the maximum throughput achieved, shown in Figure 5.10. For TAS, we run two different configurations: 1. TAS shares a core with Memcached; 2. Single-threaded Memcached. TASNIC achieves 1.8× TAS (1 app + 1 FP), 2.5× Chelsio, and 3× Linux throughput.

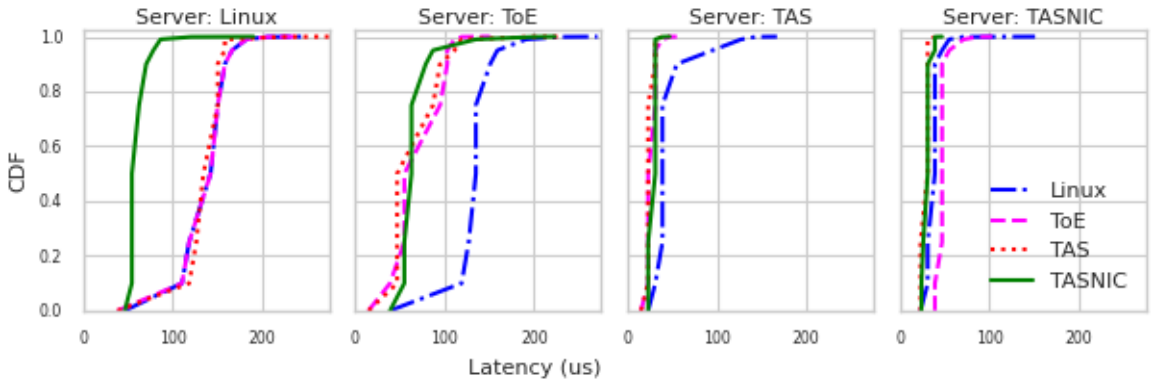

Figure 5.8: Latency of different server-client configurations.

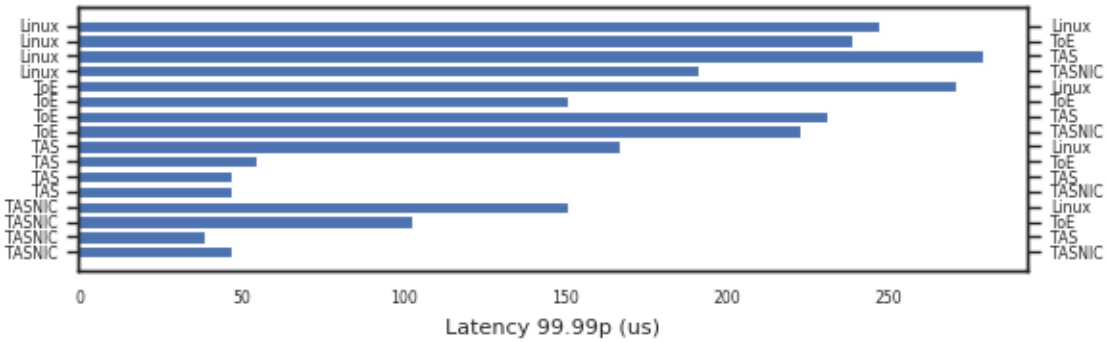

Figure 5.9: 99.99p Latency for different server-client configurations.

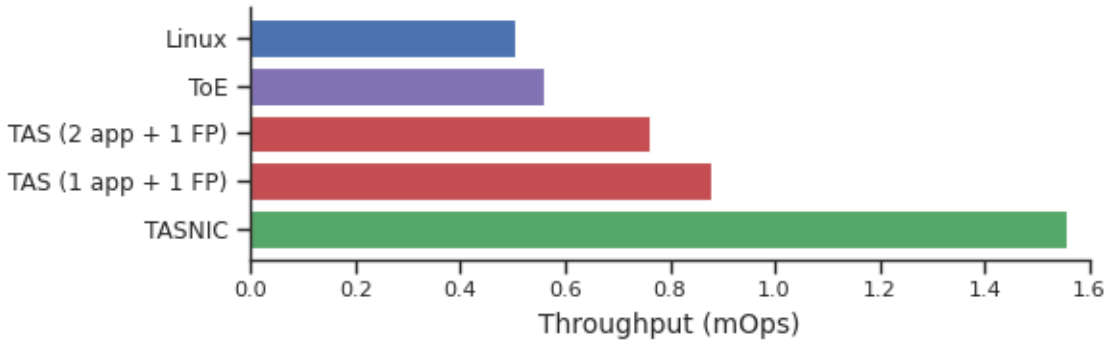

Figure 5.10: Throughput gains from TCP offload.

The kernel consumes 86% of CPU cycles for Linux and 80% of cycles for Chelsio. The TAS fast-path uses 27% of cycles. TASNIC eliminates these CPU overheads.

5.4 Flexibility

Data center networking evolves quickly, requiring stacks to be flexible. In particular, the ability to instrument new and existing code is crucial for development velocity. To demonstrate TASNIC's flexibility, we port the popular TCP tracing examples from bpftrace [13], tracking core transport events, such as per-connection drops and retransmissions. We also implement `tcpdump`, including the ability to filter packets based on header fields. Each TASNIC protocol island has 3 unassigned FPCs that we use to run these offloads. Finally, we implement new slow-path policy mechanisms, such as per-connection rate limits and per-application connection limits.

We leverage near-memory instructions to optimize these extensions. The `tcpdump` implementation comprises 283 LoC, rate and connection limits 60 LoC, and drop and retransmissions tracing 145 LoC. The above extensions were implemented in under 2 person-hours.

Chapter 6

Conclusion

TASNIC is a flexible, yet high-performance TCP offload engine to SmartNICs. TASNIC leverages fast-path processing, fine-grained parallelism, and near-memory computation for high performance, while remaining flexible via a modular design. We compare TASNIC to Linux, the TAS TCP accelerator, and the Chelsio Terminator TOE (ToE). We find that Memcached scales up to 38% better on TASNIC versus TAS, while saving up to 80% host CPU cycles versus ToE. For 64B RPCs, TASNIC cuts 99th-percentile latency to 42% and provides 70% higher throughput versus TAS, and an order of magnitude higher throughput under packet loss than ToE.

With TASNIC, we hope to make a strong case for SmartNIC offloads. The major hindrance to adoption of offloaded network stacks in the datacenters is the lack of flexibility. With TASNIC, we demonstrate that high-performance reliable ordered network transport is not at odds with efficieny and flexibility. Our system outperforms state-of-the-art TCP stacks across workloads without sacrificing application compatibility and inter-operability. We also demonstrate how applications gain extra CPU cycles by simply offloading the network stack to the hardware.

Furthermore, we outline general design principles to aid implementation of performant software on highly constrained network hardware. Our insights are valuable in understanding how best to integrate SmartNICs into the datacenter fabric. In the future, we consider how to generalize TASNIC design to other SmartNIC architectures. Particularly, we hope to implement TASNIC in higher-level programming languages such as P4. We also believe that our work will influence the design of new transport protocols that can be efficiently offloaded to programmable network hardware and also guide the evolution of SmartNIC architectures.

Bibliography

[1] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). *ACM SIGCOMM Computer Communication Review*, 40(4):63–74, August 2010.

[2] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'20, pages 93–109, 2020.

[3] 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 *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'14, pages 49–65, 2014.

[4] 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.

[5] 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.

[6] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, OSDI'09, pages 15–28, 2009.

[7] The Linux Foundation DPDK Project. DPDK. https://www.dpdk.org/.

[8] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th symposium on operating systems principles*, pages 54–70, 2015.

[9] 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 *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'18, pages 51–66, Renton, WA, April 2018. USENIX Association.

[10] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity Ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 202–215, 2016.

[11] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A new programming interface for scalable network I/O. In *10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI 12, pages 135–148, Hollywood, CA, October 2012. USENIX Association.

[12] 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.

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

[14] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *13th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '16, pages 425–438, 2016.

[15] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'14, pages 489–502, 2014.

[16] 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*, SOSP '17, pages 121–136, 2017.

[17] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'19, pages 1–16, 2019.

[18] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 295–306, 2014.

[19] 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.

[20] 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, pages 1–16, 2019.

[21] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference*, ATC'19, pages 863–880, 2019.

[22] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 104–120, 2017.

[23] 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.

[24] 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 Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 339–352, New York, NY, USA, 2016. Association for Computing Machinery.

[25] 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, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

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

[27] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP'19, pages 399–413, 2019.

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

[29] 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.

[30] 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*, ATC '13, pages 103–114, 2013.

[31] 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. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.

[32] 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, 2018.

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

[34] 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, 2018.

[35] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. Acceltcp: Accelerating network applications with stateful TCP offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'20, pages 77–92, 2020.

[36] Netronome. Netronome NFP-4000 flow processor. https://www.netronome.com/media/documents/PB_NFP-4000-7-20.pdf, 2020.

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

[38] Netronome. Programming NFP with P4 and C. https://www.netronome.com/media/documents/WP_Programming_with_P4_and_C.pdf, 2020.

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

[40] 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, page 337–350, New York, NY, USA, 2012. Association for Computing Machinery.

[41] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems*, 33(4):1–30, 2015.

[42] Marius Poke and Torsten Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 107–118, 2015.

[43] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture*, ISCA'14, pages 13–24. IEEE, 2014.

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

[45] Redis Labs. memtier_benchmark: Load generation and bechmarking NoSQL key-value databases. https://github.com/RedisLabs/memtier_benchmark, 2020.

[46] 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, 2017.

[47] 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.

[48] The Linux Foundation. toe. https://wiki.linuxfoundation.org/networking/toe.

[49] Brandon Wilson. Why are we deprecating network performance features (kb4014193)? https://techcommunity.microsoft.com/t5/core-infrastructure-

and-security/why-are-we-deprecating-network-performance-features-kb4014193/ba-p/259053, Mar 2021.

[50] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. *ACM SIGCOMM Computer Communication Review*, 45(4):523–536, 2015.