TEA: Enabling State-Intensive Network Functions on Programmable Switches

Daehyeok Kim
Carnegie Mellon University and Microsoft Research

Zaoxing Liu
Carnegie Mellon University and Boston University

Yibo Zhu
ByteDance Inc.

Changhoon Kim
Intel, Barefoot Switch Division

Jeongkeun Lee
Intel, Barefoot Switch Division

Vyas Sekar
Carnegie Mellon University

Srinivasan Seshan
Carnegie Mellon University

Abstract

Programmable switches have been touted as an attractive alternative for deploying network functions (NFs) such as network address translators (NATs), load balancers, and firewalls. However, their limited memory capacity has been a major stumbling block that has stymied their adoption for supporting state-intensive NFs such as cloud-scale NATs and load balancers that maintain millions of flow-table entries. In this paper, we explore a new approach that leverages DRAM on servers available in typical NFV clusters. Our new system architecture, called TEA (Table Extension Architecture), provides a virtual table abstraction that allows NFs on programmable switches to look up large virtual tables built on external DRAM. Our approach enables switch ASICs to access external DRAM purely in the data plane without involving CPUs on servers. We address key design and implementation challenges in realizing this idea. We demonstrate its feasibility and practicality with our implementation on a Tofino-based programmable switch. Our evaluation shows that NFs built with TEA can look up table entries on external DRAM with low and predictable latency (1.8–2.2 μs) and the lookup throughput can be linearly scaled with additional servers (138 million lookups per seconds with 8 servers).

CCS Concepts

• **Networks** → **Programmable networks**; **In-network processing**; • **Hardware** → **Emerging technologies**.

Keywords

Programmable switches, Programmable networks, Data centers, Remote Direct Memory Access, Network Function Virtualization

ACM Reference Format:
Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. 2020. TEA: Enabling State-Intensive Network Functions on Programmable Switches. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, NY, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3387514.3405855

1 Introduction

Network functions (NFs) are an essential component in today's online service infrastructure. They are deployed on the critical path of the infrastructure (e.g., at the front-end) where a large volume of traffic with many concurrent flows needs to be handled. This requires NFs to be scaled for overall network operations.

NFs have been traditionally deployed either using standalone hardware appliances or a cluster of commodity servers (also known as network function virtualization (NFV)) [29, 59]. More recently, another approach has been gaining attention in the community: NFs implemented on programmable switch ASICs (e.g., [5, 16, 53]).

However, we find that none of these approaches can handle NFs when there is a combination of a large number of concurrent flows (e.g., $O(10M)$) and a very high traffic rate (e.g., > 1 Tbps). A programmable switch ASIC cannot serve a large number of concurrent flows that requires a large flow table due to its small on-chip SRAM space although it has enough capacity to process a very high traffic rate. Similarly, it requires several tens of hardware appliances or hundreds of servers to handle the high-traffic rate, which significantly increases operational cost.

We observe that the limited on-chip SRAM space is a key bottleneck for programmable switch ASICs. If we could enable the switch ASICs to store lookup tables on cheaper DRAM in a scalable way, it could be a new enabler to serve a broader set of operating regimes, which are defined by workloads and operating conditions (i.e., traffic rate and the number of concurrent flows that NFs have to process), cost-efficiently. In this paper, we envision a new system architecture called TEA (Table Extension Architecture) that enables the switch ASICs on the top of racks in an NFV cluster to leverage DRAM on commodity servers.

While using server DRAM is an appealing low-cost and scalable solution, accessing server DRAM is inherently slower than accessing on-chip SRAM. As we discuss in §3.1, without careful design, this can significantly degrade processing performance and availability of NFs. Indeed there are several technical challenges in realizing this vision in practice:[1]

- First, for external DRAM access, while RDMA (Remote Directly Memory Access) looks a promising solution, it is unclear how to do RDMA from the switch ASIC without modifying it. Our insight is that by leveraging the programmability of ASIC, we

[1]Our recent position paper proposes this high-level idea [47]. However, that work fails to tackle these technical challenges and falls short of providing a concrete proof-of-concept realizing the architecture.

can implement a subset of the RDMA protocol that suffices for our rack-scale deployment model in NFV clusters.

- Second, since each external DRAM access incurs high latency (a few μs), TEA must complete table lookups in a single-round trip to DRAM and must continue processing other packets. At first glance, it would seem that conventional cuckoo hashing [58] would suffice. However, cuckoo hashing is not suitable for external DRAM because it can require multiple memory accesses at times. Fortuitously, we find that bounded linear probing [67], a design originally created for improving cache hit rates, can be a basis for enabling table lookups guaranteed to complete in a single round trip. In addition, we adapt this data structure to provide temporary storage to support our deferred packet processing needs.
- Third, to support NFs that require several hundred million lookups per second, we need mechanisms to leverage the available DRAM and DRAM-access bandwidth across multiple servers. While traditional distributed hashing schemes (e.g., consistent hashing [43]) help scale out the lookup throughput by distributing table entries and balancing lookup request load across servers, we observe that they consume too many ASIC resources. We show that simpler, resource-efficient hashing schemes, combined with a small on-chip SRAM cache, can address both the load balancing and scaling requirements.
- Lastly, for high availability, one may detect servers' availability changes (due to server failures or congested link) in the control plane, but it could take several milliseconds to make the data plane react to it, degrading overall performance. We demonstrate that it is possible to repurpose existing ASIC's features to support rapid failure detection and fail-over in the data plane.

TEA provides a *virtual table abstraction* for lookup tables stored across the combination of on-chip SRAM and external DRAM, creating the illusion of large, high-performance tables to NFs. Our focus is on NFs such as L4 load balancers, firewalls, NATs, VXLAN or VPN gateways that are *compute-light* and *state-heavy*. Developers can write such NFs using a library of TEA APIs implemented in P4 [17] which is a programming language for programmable switches. We expose the APIs as modularized P4 codes so that developers can easily integrate TEA with their NF implementations.

We implement a prototype of TEA in P4 and four canonical NFs using the TEA API. We evaluate it with microbenchmarks as well as NF benchmarks in our testbed consisting of a Tofino-based programmable switch and 12 commodity servers. Our evaluations show that TEA allows NFs running on the switch to look up table entries with low and predictable latency (1.8–2.2 μs), and the throughput can be scaled linearly by recruiting more servers (138 million lookups per second with 8 servers in our testbed). Compared to server-based NFs with a single server, TEA-based NFs achieve up to 9.6× higher throughput and 3.1× lower latency without consuming the CPUs and many ASIC resources. We also show that TEA can react to server availability changes within a few microseconds.

2 Background and Motivation

NFs are deployed in many network settings, including inside the cloud and at the edge. They perform a wide range of tasks, ranging from packet filtering and load balancing to encryption and deep packet inspection. In this paper, we focus on *compute-light* and *state-heavy* NFs, such as L4 load balancers, firewalls, NATs, VXLAN or VPN gateways. Even though NFs in this category are not compute intensive, they still need to support a large volume of traffic and concurrent flows on the critical path (e.g., at the front-end of the cloud). Thus, their performance and scalability are the key for overall network operations.

	Hardware appliance	**Commodity Server**	**Programmable Switch**
Performance	40 Gbps	10 Gbps	3.3 Tbps
Memory	O(10GB) DRAM	O(10GB) DRAM	O(10MB) SRAM
Price	>\$40K	\$3K	\$10K
Energy consumption	480W	200W	620W

Table 1: Comparison of NF deployment options. We excerpt the information from product briefs [4, 7, 13] and prior work [53, 59].

There are three typical options to realize such NFs today: (1) using standalone hardware middlebox appliances, (2) implementing them on a cluster of commodity servers (i.e., NFV cluster) [19, 29, 59], and (3) implementing them on emerging programmable switches [7, 9]. We note that while there are other options such as implementing NFs on FPGA boards attached to servers (e.g., [31]), we consider the above three options that have been widely studied and deployed today.

Network operators may choose different options by considering the performance, memory size, cost, and energy efficiency of each option based on their workloads and operating conditions (i.e., traffic rate and the number of concurrent flow that NF instances have to process). To understand which option is better in which scenario, we analyze a canonical NF, load balancers, in four operational regimes.[2] Table 1 compares these options in terms of performance, memory size, price, and energy consumption, and we use these numbers in our analysis below.

Regime 1: Low traffic rate (<100 Gbps) / Small number of concurrent flows (e.g., 100K flows and ≈1 MB per-flow state). This regime can be served by using any of three options. While supporting 100 Gbps traffic would require 3 hardware appliances (~\$120K), or 10 servers (~\$30K), a single programmable switch can support it with on-chip SRAM which is large enough to serve the small flow state. Thus, using a programmable switch would be the most cost and energy-efficient solution for this regime.

Regime 2: Low traffic rate (<100 Gbps) / Large number of concurrent flows (e.g., 10M flows and ≈100 MB per-flow state). A programmable switch cannot handle this workload since it does not have enough SRAM to store the flow state. As mentioned above, supporting 100 Gbps traffic would require 3 hardware appliances or 10 servers. In both these options, the systems can easily store the relevant flow state.

Regime 3: High traffic rate (>1 Tbps) / Small number of concurrent flows (e.g., 100K flow and ≈1 MB per-flow state). In

[2]While our analysis focuses on a specific case of load balancers, these observations also apply to other NFs such as firewalls, gateway functions, NATs, and ACLs.

this regime, using a programmable switch would be the most cost and energy-efficient solution because the per-flow state can fit in its SRAM space and it can easily serve the traffic. Hardware appliances and commodity servers would require many nodes to support this traffic rate making them very expensive (25 × \$40K appliances vs. 100 × \$3K servers vs. 1 × \$10K switch).

Regime 4: High traffic rate (>1 Tbps) / Large number of concurrent flows (e.g., 10M flows and ≈100 MB per-flow state). Many servers or appliances are required as the traffic rate increases (e.g., 10 Tbps requires 1000 high-end servers, which costs \$3M). Although programmable switches can handle the traffic rate [7], their limited memory makes it infeasible to support the needed flow state. One could add more on-chip SRAM (\$2-5K per GB) with chip modification or more switches to address the memory limitation, but costs would rise significantly.

In summary, our analysis suggests that: (1) servers and appliances can handle the low-bandwidth regime effectively, (2) programmable switches are great when flow-state fits in the limited SRAM space, and (3) nothing handles the most demanding workloads well. Ideally, if we could build an architecture that enables switches to utilize more memory with cheaper DRAM (like servers) in a scalable way, it would make programmable switches more broadly applicable and serve the extreme regime cost-efficiently.

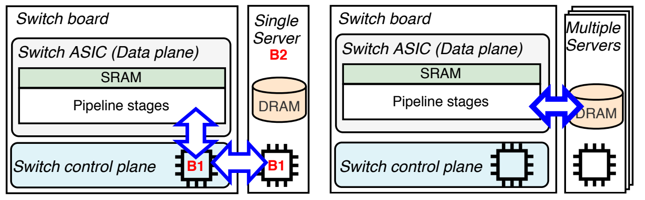
3 Design Space and Challenges

Building on the above analysis, we explore if and how we can potentially leverage external DRAM that already exists in the network. Now, there are two places where we can naturally find available DRAM *near* the switch ASIC:

(1) Switch's control plane. The control plane has a few GB of DRAM to manage the control plane data. An ASIC could access the DRAM via the PCIe channel between the ASIC and the control plane CPU. Note that the PCIe channel has a limited bandwidth which is lower than the ASIC's per-port bandwidth. While this low and fixed bandwidth is enough to process occasional control plane traffic, it cannot support higher traffic rates (which can cause high memory access rate) without significant hardware modifications. Also, although in theory, it is possible to add additional DRAM to the control plane, in practice, the size is fixed at design time. (e.g., 8 GB in the switch in our testbed [14]).

(2) On-board off-chip DRAM. Some switch ASIC vendors have added custom off-chip DRAM on the switch board [8]. This DRAM is used for custom tasks such as buffering packets or storing specific lookup tables. Similar to the control plane case, the memory access bandwidth and size is fixed at design time, which makes it very hard to scale without chip modification. Note that while a future switch ASIC architecture might provide on-board off-chip DRAM with larger size and higher bandwidth, it requires new interfaces and mechanisms to access DRAM from a programmable pipeline. We discuss this further in §7.

We observe that two options above do not scale in terms of memory access bandwidth and capacity today, which are typically fixed at hardware design time. We believe that support for scaling becomes more critical as the total amount of traffic (both in terms of traffic volume and number of concurrent flows) each switch needs to process increases [12, 24].



(a) Naïve design and performance bottlenecks (B1 and B2). **(b) TEA enabling to access external DRAM in the data plane without CPU involvement.**

Figure 1: Comparison between RPC-based naïve design and TEA to access external DRAM.

Our vision. In this paper, we take an alternative approach that leverages *DRAM in commodity servers in NFV clusters* in a scalable way. A typical NFV cluster (either inside the cloud or at the edge) consisting of multiple racks of servers [19, 29, 32, 59] already has several tens of GB of DRAM on each server. If we can reserve some portion of DRAM and let the switch ASIC located at the top-of-rack (ToR) access it, the ASIC could make use a large per-flow table, which would not be possible with on-chip SRAM today.

Using a single server could still limit the access bandwidth, i.e., minimum of network bandwidth between the ASIC and the server, and PCIe bandwidth in the server. However, we can leverage multiple servers to increase the aggregate bandwidth. Also, while the ASIC uses DRAM in servers, CPUs on the servers can simultaneously serve other tasks such as compute-intensive NFs, including traffic en/decryption or payload inspection, which cannot be supported by switches today.

If this can be realized, programmable switches can become an effective way to serve *high traffic rate involving a large number of concurrent flows*, and thus work for all the regimes we considered earlier. However, realizing this vision has key design and implementation challenges, as we describe next.

3.1 Challenges

To understand why it is challenging to realize this vision, let us consider a natural starting point based on prior work using traditional Remote Procedure Call (RPC) mechanisms [41, 57] (Figure 1a). Specifically, the switch ASIC sends and receives RPC requests and responses via the switch control plane to avoid adding complexity (e.g., state management for reliable transport) to the data plane. While this is functionally correct, there are three fundamental bottlenecks:

(1) High and unpredictable latency. A table lookup can result in high latencies because of the latency between the ASIC, the control plane CPU, and the server CPU (over the network), which can take a few hundred microseconds. Moreover, the uncertainty introduced by the scheduling logic on the switch control plane and server CPU can introduce jitter and high variability [46].

(2) Limited memory access bandwidth. The lookup throughput is constrained by the minimum of the bandwidth between ASIC-to-the-control-plane-CPU and control-plane-CPU-to-server-CPU. Both bandwidths are typically very limited (e.g., PCIe bandwidth between the ASIC and the control plane is a few tens of Gbps which is much lower than a few hundreds of Gbps of ASIC's per-port bandwidth available today) and fixed at hardware design time.

(3) Availability. If the server fails or the network link between the control plane and the server becomes unavailable, the switch cannot lookup tables on external DRAM, degrading NF performance.

We observe that the root causes of these problems are (1) the involvement of CPUs at the control plane and the server and (2) the use of the single server (Figure 1a). This motivates us to ask: Is it possible to allow the switch ASIC to access external DRAM *purely in the data plane* and *without servers' and the control plane's CPU involvement* in a scalable way *across multiple servers*? To answer this question, we must address the following challenges:

C-1. Data-Plane External DRAM Access. Switch ASICs typically do not have direct external DRAM access capability. Is it possible to enable it without hardware modifications?

Even if the ASIC can somehow directly access external DRAM, it can incur a few microseconds of latency which is an order of magnitude slower than its packet processing speed. This long latency creates the following two challenges:

C-2. Single Round-Trip Table Lookups. If we use conventional hashing (e.g., cuckoo hashing [58]) for storing and locating table entries in external DRAM, multiple DRAM accesses may be required to lookup an entry. Is it possible to make the ASIC do a table lookup in a single round-trip to DRAM without involving server CPUs and hardware modifications?

C-3. Packet Processing. The ASIC must be able to continue processing the packet (e.g., modifying header fields) after completing the lookup from external DRAM. In the meantime, it also needs to keep processing subsequent packets in the pipeline. How can we manage the packet until the lookup completes?

C-4. Load-Balanced Bandwidth Use. Although using multiple servers (i.e., adding network links) increases external DRAM access bandwidth, a subset of links could become overloaded due to the access locality (i.e., most of memory accesses are destined to the subset of servers' DRAM). This makes it hard to utilize available link bandwidth. How can we ensure that memory access loads are balanced across servers?

C-5. Tolerating Server Churn. Access to external DRAM becomes unavailable when a server fails or the network becomes congested (causing packet drops). How can we detect and react to these events quickly to minimize performance degradation?

4 TEA Design

To address the above challenges, we design TEA, a virtual table abstraction for tables stored across local SRAM and external DRAM. Using the abstraction, NFs running on a ToR programmable switch can perform key-based (e.g., 5-tuple of an IP packet) table lookups, and TEA fetches the corresponding entries either from switch-local SRAM or remote DRAM. When it accesses DRAM, it delays the processing of the packet corresponding to the lookup request without blocking the rest of the packet processing pipeline. TEA's lookup response handler resumes the delayed packet's processing when DRAM lookup completes.

Figure 2 illustrates this workflow. TEA provides a set of APIs implemented in P4, a language to program NFs on programmable switches, and exposes each component as a module in P4 [17, §13].

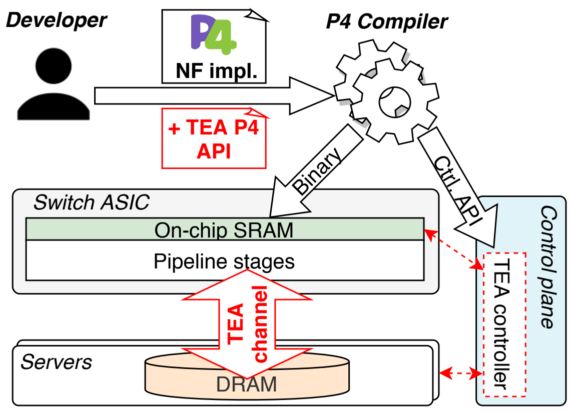


Figure 2: NFs implemented in P4 can be extended with TEA P4 API to look up tables across external DRAM and on-chip SRAM. The control plane is (dotted lines) involved when establishing a TEA channel.

This enables developers to easily integrate TEA with their NF implementations in P4. Once developers write their NFs using TEA components, the unmodified P4 compiler generates a binary of TEA-enabled NFs that can be loaded to the data plane and control plane APIs that can be used for configuring TEA components in the data plane.

TEA builds on the following five key ideas to address the challenges described in §3.1:

1. *Leveraging ASIC programmability* to enable simplified RDMA in the data plane (§4.1).
2. *Repurposing bounded linear probing* to guarantee hash table lookups in a single-round trip to external DRAM (§4.2.1).
3. *Offloading packet store to external DRAM* to enable asynchronous lookups (§4.2.2).
4. *Leveraging the small-cache theory [30]* to scale out the throughput (§4.3).
5. *Repurposing ASIC's hardware capabilities* to detect and react to sever availability changes in the data plane (§4.4).

4.1 DRAM Access in the Data Plane

To access external DRAM, we choose RDMA, which is quite common in service provider deployments [34, 55]. In comparison to RPC, RDMA is an attractive option because it is designed specifically for predictable performance memory access. It provides hardware support for a set of low-level memory operations such as read, write, and a few atomic operations (e.g., fetch-and-add). Since it does not involve the server CPU for either the memory access or the reliable transport of messages, RDMA reduces both memory access latency down to $\approx 2\ \mu s$, and delay jitter, and allows the use of the CPU for other compute-intensive tasks.

Challenges of using RDMA from switch ASICs. However, we still need to address two practical problems: (1) Is it feasible to generate RDMA packets purely in the switch data plane when DRAM access is needed? (2) Can we support reliable RDMA transport

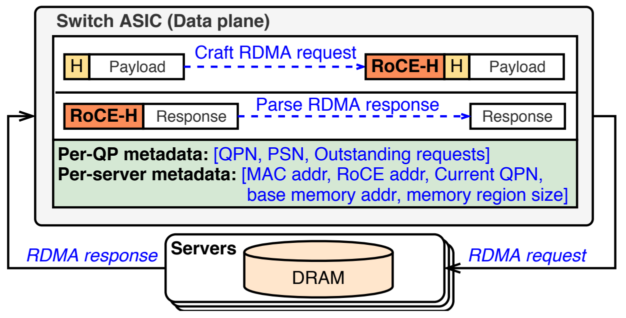

Switch ASIC (Data plane)
H
Payload
Craft RDMA request
RoCE-H
H
Payload
RoCE-H
Response
Parse RDMA response
Response
Per-QP metadata: [QPN, PSN, Outstanding requests]
Per-server metadata: [MAC addr, RoCE addr, Current QPN, base memory addr, memory region size]
RDMA response
Servers
DRAM
RDMA request

Figure 3: Switch ASIC generates RDMA requests by adding RoCE headers on incoming packets and parse RDMA responses without specialized capabilities for RDMA. To maintain reliable channels, the ASIC maintains per-QP and per-server metadata.

within the switch data plane? (i.e., can switch ASICs maintain the necessary per-connection RDMA context and protocols?)

Our approach. While it may be hard to implement reliable RDMA in general on a programmable switch, we observe that we do not need fully functional RDMA for our use case. Our key insight here is that the programmable features of modern switch ASICs together with the scoped deployment model of TEA enable us to implement a small but sufficient subset of RDMA features we need.

1) Generating RDMA packets: With respect to the first sub challenge, we note that the most popular RDMA technology today is RoCE (RDMA over Converged Ethernet) protocol [37, 38], where RDMA requests and responses are regular Ethernet packets with RoCE headers. This means that ASICs can generate valid RDMA requests by crafting RoCE packets without needing any RDMA-specific hardware components.

Figure 3 illustrates this high-level idea. When the data plane needs to access DRAM, it crafts an appropriate RDMA packet by adding a series of specific RoCE headers to the incoming packet. This include Ethernet headers, global route headers, base transport headers, and RDMA extended transport header with RDMA metadata such as a queue-pair number (QPN), a packet sequence number (PSN), a remote access key (Rkey), a remote memory address, and a length of data to be written or read from the DRAM.[3] The needed metadata is provided via the control plane in advance.

2) Reliable RDMA: To address the second question of reliable RDMA, we leverage the assumption that in TEA, DRAM servers are directly connected to the ToR switch. This means that if we can make RDMA request and response packet not be dropped at the switch or NICs, the RDMA channel becomes reliable. Thus, we can simplify the RoCE protocol with two possible options. One is by ensuring the underlying Ethernet network is lossless via Priority Flow Control [2]. In this option, a NIC sends a PAUSE request to the switch when RDMA requests are buffered more than its threshold to prevent packet drops due to buffer overflow. When the switch receives a PAUSE request, it has to buffer packets until the NIC allows to send packets. We adopt this option in our prototype implementation in addition to our simple switch-side flow control to cope with the current NIC configuration as we describe in §5.3.[4] Alternatively, we can also configure a higher QoS-level for our RDMA traffic over lossy fabric [18]. These options allow us to enable RDMA between the ASIC and DRAM servers with a minimal amount of RDMA context metadata and without complex retransmission schemes. Specifically, it only needs to maintain a QPN (4 bytes) and tracks a packet sequence number (4 bytes) and the number of outstanding requests (2 bytes) for each queue-pair, which are used when crafting RDMA requests for the QP. Maintaining such metadata in the data plane requires only up to a few KBs of SRAM in total.

4.2 TEA-Table: Lookup Table Structure

The design of TEA's table data structure, TEA-Table, addresses two key issues: (1) how to complete a lookup in a single round-trip to external DRAM and (2) how to defer processing of the current packet until the lookup completes and continue processing other packets without blocking. TEA-Table repurposes a data structure that was originally designed for improving cache hit rates in software switches [67] to achieve single RTT lookups and incorporates remote packet buffers within the data structure to accommodate deferred packet processing.

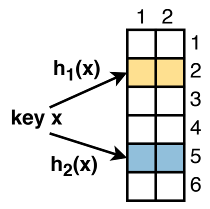
4.2.1 Single Round-trip Lookups: RDMA only provides low-level memory operations such as read and write, using virtual memory addresses. However, NFs require *richer key-based lookup interface* to retrieve table entries with keys (e.g., an IP 5-tuple for an address mapping table in NAT) from DRAM. Thus, TEA must map a *key* to a *virtual memory address*. The challenge is that due to relatively large DRAM access latency ($\approx 2\ \mu s$), we must be able to locate and fetch the entry in a single DRAM read.

Strawman solutions. At first glance, it appears we can use traditional hashing techniques. Indeed, many modern switch ASICs adopt variations of cuckoo hashing [58] for exact-match lookups in SRAM as it guarantees constant-time lookup. A caveat, however, is that each lookup requires multiple memory accesses. This means, with two-way cuckoo hashing, each lookup requires two independent memory reads. While this is feasible with fast parallel lookups on SRAM, our experience suggests that extending it to external DRAM via RDMA channel would either significantly degrade the performance of NFs or make the data plane logic complicated. To reduce multiple DRAM accesses in cuckoo hashing, we need to know precisely which of the two hash tables to access for a given key. Recent work, EMOMA [60], uses additional Bloom filters [21] in SRAM to address this issue. By checking for membership, the query can be directed to the appropriate hash table. Since there is a risk of false positives in the filter, EMOMA has a more complex item insertion that checks if inserting a new entry causes false positives. Unfortunately, this makes it impractical.[5]

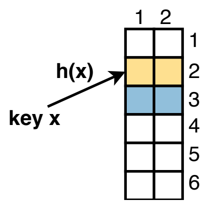
[3]QP is the connection abstraction used in RDMA communications (similar to the socket) and QPN is a unique identifier assigned for each QP. RKey is assigned to each memory protection domain where allocated memory region is registered.

[4]In our experiments, we observe that our switch-side flow control mechanism prevents a NIC buffer from being overflowed before the NIC generates PAUSE frames.

[5]In our simulation, it takes several hours to insert just a few tens of million entries and implementing BFs for such a scale consumes other resources across multiple packet processing stages in the ASIC. Since such a slow insertion speed with a non-negligible amount of resource consumption makes this approach impractical, we do not consider this design.

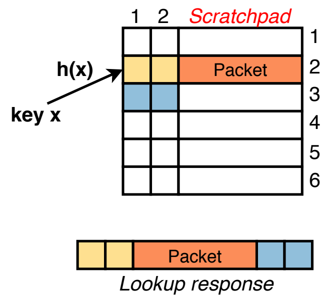


(a) Cuckoo hashing.

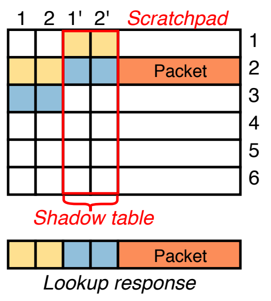


(b) Bounded linear probing.

Figure 4: Cuckoo hashing and bounded linear probing. In this example, there are 6 buckets and 2 cells per bucket. The numbers on the top and right side indicate cell and bucket indices, respectively.



(a) Incorrect design: switch cannot parse entries in blue cells.



(b) Corrected design with shadow table.

Figure 5: Design of TEA-Table with scratchpads. Scratchpads temporarily store original packets during lookups. i^{th} bucket of the shadow table has a copy of $((i+1)\ mod\ n)^{th}$ bucket of the original table ($n = 6$ in this figure).

Our approach. We build on a recent approach called Bounded Linear Probing (BLP) [67]. BLP was originally designed for improving cache hit rates and reducing lookup latency in software switches. Somewhat serendipitously, we find that it can also be used in our setting. Figure 4 illustrates the differences between cuckoo hashing and BLP. When placing and looking up a table entry, instead of using two hash functions as in cuckoo hashing (Figure 4a), BLP uses one hash function and lets the second bucket be placed right next to the first bucket (Figure 4b).

We find that BLP's design lends itself to fetching both hash buckets in a single RDMA read. However, since BLP is designed for caching, we need to handle colliding entries differently. In BLP, when hash collisions happen, it evicts colliding entries and puts them to the main memory region (i.e., DRAM). In contrast, in TEA, since the table is already located in DRAM, we put colliding entries to switch SRAM, making all entries exist in either SRAM or DRAM. Although it consumes some amount of SRAM space, we empirically prove that the collision rate is only 0.1% for the same size of the hash table as the cuckoo hash table and the same number of keys inserted. For example, when the total number of table entries is 80 million, 80K colliding entries are stored in SRAM, which takes around 4MB in the NAT mapping table with IPv6 addresses. This design is much simpler than the cuckoo hash-based approaches and requires fewer resources in the ASIC while guaranteeing at most one RDMA read per lookup.

4.2.2 Deferred Packet Processing: Another key challenge is storing the packet while DRAM is accessed. This is especially critical since the $\approx 2\ \mu s$ DRAM access time is very long in the context of high-speed switching where a packet is processed every nanosecond. A naïve solution would be to buffer the packet using on-chip SRAM. However, it is undesirable to use scarce SRAM for buffering a large number of packets during DRAM access.

We address this issue by storing packets to DRAM and reading back the packet along with retrieving the table entry. Specifically, we propose TEA-Table which extends our hash table structure by employing *scratchpads*. In each scratchpad, we temporarily store a packet during lookups. As shown in Figure 5, in TEA-Table, we allocate a scratchpad for each bucket large enough to hold an MTU size packet. Note that our design requires the path MTU between the switch and the DRAM servers to be larger than the end-to-end MTU. In our prototype implementation, we set the path MTU size to 9000 bytes and the end-to-end MTU to 1500 bytes.

Hardware constraints of current RDMA NIC and switch ASIC impose another challenge. Since the NIC allows an RDMA read operation to read only a continuous memory region, with a naïve design of TEA-Table, an original packet is placed between two buckets in a lookup response, as illustrated in Figure 5a. While we need to parse both buckets, with this format of a lookup response, the ASIC often cannot parse the second bucket (blue-colored) when the original packet (orange-colored) is large. This is because high-speed switching ASICs usually can parse only the first few hundreds of bytes in each packet.

To address this issue, we put a shadow table whose i^{th} bucket contains a copy of the $((i+1)\ mod\ n)^{th}$ bucket of the original table, where n is the number of buckets in the table. As shown in Figure 5b, the shadow table allows placing two buckets consecutively before the scratchpad in the lookup response packet. In this way, the switch can parse two buckets. Although the shadow table incurs additional DRAM consumption, given a small bucket size (<150 B) and a large available DRAM size ($> O(1\ GB)$), the cost is reasonable to achieve our goal.

4.2.3 TEA-Table operations: Given these building blocks, we now describe operations in TEA-Table.[6]

- *Inserting an entry*: Since it takes some time to complete an insertion operation, new entries are first inserted in to an *SRAM stash*, which is a small SRAM space to keep the pending entries. When there is no room in both buckets, our insertion logic running on the control plane chooses a victim cell and replaces it with the new key. In the next iteration, the logic tries to insert the key from the victim cell. If there still exists a key that fails to be inserted after *MaxTries* iterations, it remains in the SRAM Stash. Once the insertion is completed, the entry will be removed from the stash.
- *Deleting an entry:* Deletion is a simple operation which takes a key of a target entry as a parameter. To delete the entry, our

[6]The pseudocode for each operation can be found in Appendix A.

deletion logic running on the control plane locates the cell of the entry using the same logic as in the insertion operation and overwrites the cell with zeros.

- *Lookup an entry:* When an NF requests a lookup for an entry, our lookup logic first checks whether it exists in SRAM Stash or Cache (we explain the cache in §4.3), and if it does, the entry in SRAM is returned. Otherwise, after retrieving the DRAM address of the bucket, it uses RDMA to write the packet to the scratchpad of the bucket and then performs an RDMA read of the entire bucket including the packet stored in the scratchpad.
- *Lookup response handler:* Upon receiving the RDMA read request, the NIC sends an RDMA read response containing a lookup response back to the switch. To handle the lookup response at the switch, we introduce *Lookup response handler*, which is a similar concept as the callback handler in other programming languages. Upon receiving a lookup response, the handler returns an entry and the original packet parsed from the response. TEA allows developers to define custom actions in the handler (e.g., modifying header fields with the fetched entry).

Note that as the insertion and deletion operations are relatively complex compared to the lookup operation, the control plane has to execute them. Due to this constraint, our current design does not support NFs that add and delete table entries in the data plane.

4.3 Multiple DRAM Servers

Recall from §3, we can achieve higher lookup throughput using multiple servers. To utilize the available access bandwidth effectively, we need to answer the following questions: (1) How to partition and distribute a TEA-TABLE across multiple servers? (2) How to balance memory access load across the servers?

Strawman solution. To partition the table and provide load balancing, we can consider conventional distributed hashing schemes such as consistent hashing [43] and rendezvous hashing [64] as they can achieve good load balance among servers by partitioning hash tables. However, in these algorithms, each server is in charge of many non-contiguous parts (i.e., buckets) of the table. In turn, this causes the switch ASIC to maintain a large number of ⟨bucket range, server ID⟩ mappings, consuming a non-negligible amount of TCAM space. For example, if one wants to implement consistent hashing, supporting N servers with $100N$ virtual nodes[7] can use up to $(100N - 1)$ range-matching rules.

Our approach. Instead, we apply a simpler, resource-efficient hashing scheme to partition the table. We split the entire hash table into N sub-tables that contain buckets in a contiguous hash space and distribute them to N servers. The size of each sub-table can be different depending on the available DRAM provided by each server. This design requires only N range-matching rules in TCAM to locate a server for a key.

While this simple design reduces the TCAM usage, it may not guarantee the same load balance as the traditional distributed hashing approaches. Fortunately, we find that adding a small cache to the switch SRAM is helpful for load balancing across the servers. In particular, we leverage the theoretical results that caching at least $O(N \log N)$ popular entries where N is the number of *servers*, not the number of entries, can provide uniform load balancing across N servers regardless of traffic patterns or skewness [30]. For example, for NFs using per-flow table entries, the popularity can be defined as the number of packets in each flow. Specifically, we keep track of the popular entries within the data plane using a count-min sketch [25], for which efficient switch data plane implementations are already available [40, 50].

[7]In consistent hashing, multiple virtual nodes are assigned to each physical node for better load balancing [43].

As an additional benefit, this cache also reduces the total DRAM access traffic in TEA. When an NF looks up the cached entries, the requests are absorbed by the switch without consuming DRAM access link bandwidth, thus reducing the number of lookup requests that need to be served by the NICs. In practice, the small cache can help achieve near switch line-rate throughput since only a few popular entries are frequently requested and consume a significant portion of throughput [20, 26, 62]. We show the effectiveness of caching for load balancing and throughput improvement in §6.1.

4.4 High Availability

As mentioned in §3, TEA needs to detect and react to lookup failures to ensure high availability. We consider the following two lookup failure modes: (1) *high link utilization* due to regular network traffic (i.e., other than lookup requests) could cause table lookup requests be dropped. (2) When *a server fails*, lookup requests destined to the server cannot be completed.

Strawman solution. Failures could be detected by periodically checking the port counters (to estimate link utilization) and port status (as an indicator of server failures) from the control plane. However, it could take a few tens of milliseconds from detecting an event to updating the state in the data plane. The delay can result in: (1) dropping many lookup requests due to the out-of-date state and (2) overlooking short-duration events (e.g., microbursts).

Our solution. To reduce the delay, we repurpose the meter and packet generator engine of the switch ASIC to estimate port utilization and port status, respectively. Typically, the meter, which implements the RFC 2698 [36], is used for enforcing QoS policies (e.g., rate limiting). When it is executed, it returns a color (red, yellow, or green) based on pre-configured rates (i.e., if the utilization exceeds the rate, the meter returns red). The packet generator engine is typically configured to inject packets into a switch pipeline when a certain event happens mainly for diagnosis purposes.

To detect high port utilization, we set a threshold (link bandwidth in bps) for the per-port meter and get colors for ports where a lookup request can be routed. To detect a port down event, we configure the packet generator engine to generate a packet when ports go down. By processing the generated packet, TEA updates the port status table in the data plane. Based on these two per-port state information (utilization and status), TEA decides an egress port for a lookup request (i.e., an active port that is not overutilized). Note that since the meter is updated after a packet is completely received, it can lag behind less than a microsecond. We show that the gap is small enough to make it useful to react to high link utilization in §6.1.

In our prototype, we replicate hash tables in TEA-TABLE to two servers and let TEA choose a server based on the availability.

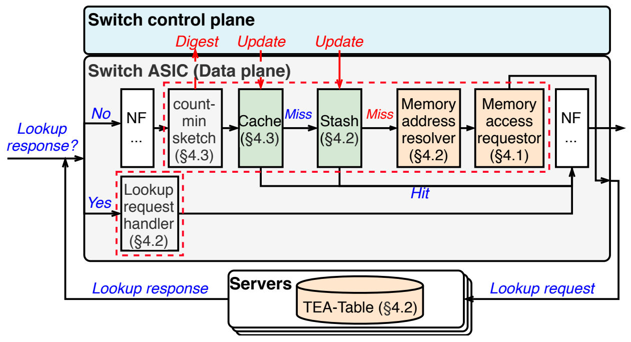


Figure 6: Summary of key components in TEA. The components form one logical TEA component (dotted-red box) used by an NF pipeline.

4.5 Putting It All Together

Figure 6 illustrates the key components of TEA on the switch data plane and servers, and how an NF uses it for packet processing. When the NF performs a lookup with a key using the TEA APIs, TEA first updates the count-min sketch of the key. Then, it checks whether an entry for the key exists in SRAM Stash or Cache (green-colored). If it exists, it directly passes the entry to the NF. Otherwise, it resolves a memory address and server ID using the memory address resolver. It then generates an RDMA write of the packet contents to the scratchpad and an RDMA read of the table row using the memory access requester (orange-colored). This design guarantees that RDMA write and read requests are always destined to the same server, and with our flow control mechanism described in §5.3, both requests are not issued and a packet is dropped when the destination server is overloaded. Upon receiving an RDMA request from the switch, RDMA NICs on servers fetch entries from DRAM and send them back to the switch. Then, the lookup response handler extracts matched entries and the original packet contents to pass them to the NF.

Overhead of TEA. When an NF accesses external DRAM for table lookups using TEA, it incurs some amount of latency and bandwidth overheads for packet processing. For latency, as we show in §6.1, it adds up to around 2 μs per-packet latency depending on the packet size. For bandwidth, since TEA generates additional RDMA packets for external DRAM lookups, it affects both the switch pipeline and link bandwidth consumption. Within the switch pipeline, as it replicates an incoming packet to generate RDMA write and read packets, it doubles the bandwidth usage of the *egress* pipeline. It also consumes the same amount of link bandwidth between the switch and a server where a target entry is located. On the server side, while TEA does not involve CPUs, it consumes some amount of servers' memory bandwidth, which may affect performance of memory-intensive applications running on servers, especially when the memory bandwidth is fully utilized. Note that if an entry for the packet is already cached, there is no overhead.

Network function	State	Table size (MB)
NAT	Per-flow address mapping	525
Stateful firewall	Per-flow connection state	353
Load balancer	Per-flow connection mapping	525
VPN gateway	Ext.-to-int. tunnel mapping	343

Table 2: The NFs we developed with TEA. Table sizes are estimated by assuming 10 million entries with IPv6 addresses.

5 Implementation

5.1 Data and Control Plane

We implement TEA's data plane in P4 [22] and compile it to Barefoot Tofino ASIC [7] with P4 Studio [6]. In the memory address resolver, we use Tofino-embedded crc64 as a hash function to locate a bucket in TEA-Table. We implement the server ID resolution using a range-matching table. In the memory access requestor, to craft lookup request packets, we make the packet replication engine in the ASIC replicate an incoming packet into two packets. The engine ensures that there is no interleaved packet between two replicas. Based on the replicas, it generates RoCE packets (i.e., an RDMA write and read) by adding RoCE headers on top of the packets based on the metadata resolved by the memory address resolver.

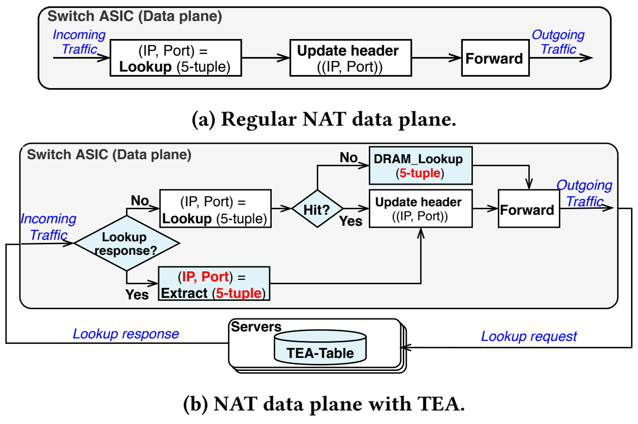
We implement the count-min sketch [25] for collecting the statistics and determining popular entries, similar to that of prior work [40, 51]. We use 4 register arrays and 64K 16-bit slots per array to implement sketches. When the sketches detect a popular key (i.e., counts of the key exceed a threshold), it reports the key to the control plane by using the digest feature in the ASIC. The digest internally maintains a Bloom filter that prevents duplicate keys from being reported. The control plane populates popular entries to the cache which is implemented as a regular exact-matching table. We use a cache of size N=1024 in our prototype which consumes approximately 55 KB of SRAM in NAT for IPv6 addresses.

Switch control plane and server agent. We implement the switch control plane in Python and C. It manages the ASIC via the ASIC driver using a runtime API generated by the P4 compiler. The server agent running on servers is written in C, which initializes an RDMA NICs on the servers and communicates with the switch control plane when it establishes RDMA connections.

5.2 Programming NFs with TEA

Our prototype implements TEA APIs as a library of modularized P4 codes using the concept of control block in P4 [17, §13]. Each control block implements key modules such as the lookup response handler, memory address resolver, and memory access requestor. Developers provide TEA with a definition of key (e.g., 5-tuple) used of a lookup table, a structure of the table stored in DRAM (e.g., using `struct` in C), and where to store the lookup response for further packet processing. Figure 7 shows how these blocks would be used to implement NAT.

To demonstrate the applicability of TEA, we implement four NFs in P4 using TEA: a NAT, a stateful firewall, a load balancer, and a VPN gateway. Table 2 describes the state each NF maintains using TEA and its estimated size. Brief descriptions of each are below, and simplified P4 codes are in Appendix B.



(a) Regular NAT data plane.

(b) NAT data plane with TEA.

Figure 7: Comparison of simplified NAT data plane with and without TEA. To use TEA, in addition to the original logic (white-boxes), developers need to add TEA modules (blue-boxes) and provide basic information necessary for lookup (red-colored).

NAT. The NAT implementation uses the TEA to store NAT translation tables, to lookup a ⟨private IP, Port⟩ pair for a given 5-tuple. It modifies the IP address and port header fields using lookup results.

Firewall. The firewall stores the connection state to external DRAM using TEA. For an external connection, the firewall looks up a connection state and uses it to determine how to handle packets.

Load balancer. The load balancer stores the per-flow server mapping table to external DRAM using TEA. For each incoming packet, it looks up a ⟨Backend server's IP address, Port⟩ from the table.

VPN gateway. We implement a VPN gateway (e.g., [11]) based on the details described in prior work [19]. It manages the external-to-internal tunnel mapping table consisting of a ⟨customer's external tunnel ID, VM IP⟩ pair as a key and a ⟨Server IP, internal tunnel ID⟩ pair as a value. For incoming packets from customers, the gateway looks up the table to retrieve corresponding server IPs and internal tunnel IDs, and translates packets.

5.3 Limitations

The NICs in our testbed limit the maximum number of outstanding RDMA read requests to 16, and if there are more requests than the limit (i.e., overloaded), they drop the requests and the QP state becomes invalid. To prevent the NICs on servers from being overloaded, we implement a simple flow control in the switch data plane, which counts and limits the number of outstanding read requests. If there is a lookup request and the number of outstanding requests has already reached to the limit, it drops the request (i.e., not generating both RDMA read and write requests), causing a packet drop. This may affect the end-to-end performance. We plan to design a mechanism that routes lookup requests to an alternative DRAM server in such a case, instead of dropping packets. Also, currently, we assume that there exists at least one server that is not overloaded, and if there is no available server, TEA does not generate lookup requests and drops the packets as above.

While our NF implementations (§5.2) access one large table, some NFs may require multiple large tables. Although the current design of TEA can support multiple tables through multiple external DRAM accesses, we plan to improve its efficiency as future work.

6 Evaluation

We evaluate TEA on a testbed consisting of a programmable switch and commodity servers using both real data center network packet traces and synthetic packet traces. Our key findings are:

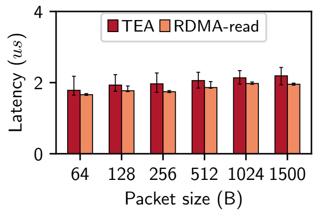
- With a single server, TEA provides a predictable lookup latency (1.8–2.2 μs) and throughput (7.3–10.9 million lookups per second) for different sizes of packets. With multiple servers, a small cache helps balance loads across servers across different skewness parameters. With the cache, adding servers scales the throughput effectively and 8 servers can perform 138 million lookups per second under a skewed workload. (§6.1).
- Compared to server-based NFs, TEA-enabled NFs are cost effective. TEA shows up to 9.6× higher throughput and 3.1× lower latency under the same hardware configuration. Even under an optimal setting for server-based NFs, TEA still shows ≈2.3× higher throughput without requiring costly hardware (§6.2).
- TEA-enabled NFs can serve traffic with latency and throughput that is comparable to the switch-only implementation (i.e., NFs running on a switch without accessing external DRAM) in the common case (§6.2).
- TEA provides these benefits without incurring much ASIC resource overhead. It consumes on-chip resources, including SRAM, TCAM, and hash bits, all less than 9% (§6.3).

Experimental setup. Our testbed consists of a Wedge 100BF-32X 32-ports programmable switch [14] with a Tofino ASIC and 12 servers equipped with two Intel Xeon E5-2609 CPUs (8 logical cores in total), 64 GB RAM, and a 40 Gbps Mellanox CX-3 Pro RDMA NIC. The servers run Ubuntu 18.04 with the kernel version 4.4.0. All servers are directly connected to the switch. We use 4 servers as packet generators and 8 as DRAM servers.

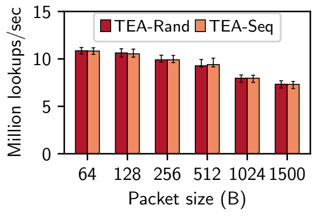
Traffic workloads. We use both packet traces collected from a real data center network [1] and synthetically generated ones. The packet sizes vary (64–1500 B) in the real trace. The synthetic traces are based on the observations from several data center measurement studies [20, 26, 62]. We generate packet traces with the flow size distribution in terms of the number of packets per flow that follows Zipf distribution with the skewness parameter (α=0.99, 0.95, 0.90). We use a keyspace of 1 million randomly generated IPv4 5-tuples when creating packet traces. We generate multiple packet traces with different packet sizes and skewness parameters. We replay the traces using DPDK-pktgen [3] on packet generator nodes. In our testbed, each traffic generator node can generate 64 B packets at around 34.54 Mpps and 1500 B packets at 40 Gbps.

6.1 Microbenchmarks

Single-server lookup latency and throughput. First, we evaluate the performance of the DRAM access channel with a single server. For this experiment, we disable the SRAM cache. For latency, we inject 10,000 packets of different sizes (64–1500 B) to measure the lookup time. As a baseline, we setup two servers directly connected and run `ib_read_lat` in perftest [10] to measure RDMA

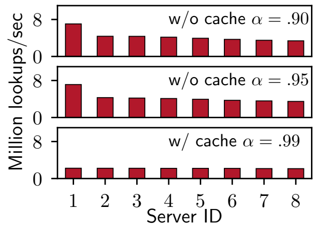


(a) Lookup latency

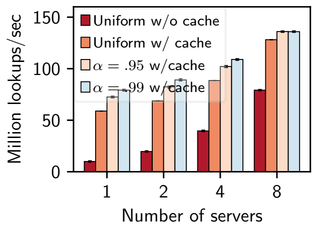


(b) Lookup throughput

Figure 8: Lookup performance of TEA via an RDMA channel with a single server.



(a) Small cache balances memory access loads.



(b) Lookup throughput scales with more servers and cache.

Figure 9: Scalable lookup throughput of TEA with multiple servers and cache.

read latencies for different message sizes. For throughput, we replay the trace for 30 seconds and measure the number of lookups completed during the period. Since the memory access pattern might affect the throughput, we force TEA to access buckets sequentially or randomly in this measurement.

Figure 8a shows the median, 10^{th} and 90^{th} percentile of lookup time. We see that each lookup takes 1.8–2.2 μs and the latency grows with the packet size, which is higher than raw RDMA reads (0.1–0.2 μs). This is mainly because our RDMA read request and response packets are larger than raw RDMA read packets. First, due to switch ASIC limitation, we are not able to remove an original packet from each replicated packet. This makes each RDMA read request packet have the original packet as a trailer. Second, in TEA, each RDMA read response packet consists of a bucket and the original packet, as illustrated in Figure 5b.

Figure 8b shows the lookup throughput with different packet sizes. At the maximum traffic rate we can generate in our testbed, the server NIC can handle 7.3–10.9 million lookups per second, and there is the only negligible difference (up to 0.02 million lookups per second) between sequential and random memory access patterns.

Overall, our evaluation shows TEA's remote DRAM access channel can provide *predictable* performance which is close to the raw RDMA performance.

Throughput scaling with multiple servers. Next, we evaluate the effectiveness of using multiple servers and a small cache to scale up the lookup throughput. Here, we replay synthetic packet traces consisting of 64 B packets with the different skewness parameter (α) for the flow size distribution and measure the number of lookups served by each server with/without the cache enabled.

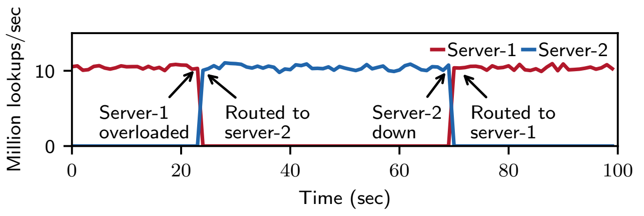


Figure 10: Lookup throughput changes during failover events.

Figure 9a shows that the lookup load distribution is skewed across servers without the cache. We also observe that such a skewed access pattern limits the aggregate when the lookup request rate is high, even if there is available link bandwidth to servers. Finally, we see that with cache, even with the most skewed access pattern (α=0.99), the load is evenly spread across servers and 49% of requests are served by the cache.

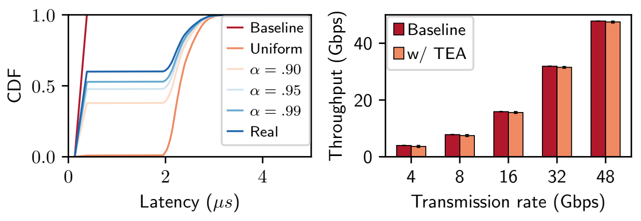
Next, we measure the aggregate lookup throughput varying the number of servers with different α values. As shown in Figure 9b, in all four cases, while the aggregate throughput scales linearly as we add more server, there is the difference in achievable maximum throughput depending on the skewness and the existence of the cache. We see that the more skewed the load distribution, the higher aggregate throughput TEA can support with the cache. When α=0.99 or 0.95, TEA can process 138 million lookups per second with 8 servers and the cache. Note that this performance is limited by the maximum packet generation rate we can achieve in our testbed.

One natural question regarding the throughput would be *what is the maximum throughput an NF with TEA can achieve with N servers in a rack?* The evaluation result shows that with 8 servers TEA can support up to 138 million lookups per second. If we extrapolate this result, it means that the NF can process up to $138/8 \times N$ million packets per seconds, which is not high enough to support very high traffic rate with small size packets, especially when skewness is not high, and this is a limitation of our current design. For example, to support a few billion packets per second traffic rate, TEA requires more than a hundred servers, which is way more than a number of servers typically existing in a rack and a number of switch ports. Note that this analysis may not be perfectly accurate because as mentioned above, the measured maximum throughput is capped by the packet generation rate in our testbed. We plan to analyze the system throughput by injecting packets at higher rates with more servers.

Availability. Next, we evaluate how TEA reacts to server churn by setting up 2 servers, loading the same table entries using server-1 as a primary and server-2 as a secondary server. We replay the 64 B packet trace and measure the lookup throughput by disabling the cache. For the result in Figure 10, we inject the background traffic from packet generators to server-1 to emulate link utilization increase. We see that TEA starts sending lookup requests to server-2, and the throughput reaches the maximum within a second (at around 24 sec.). At this point, server-2 becomes primary. We then stop injecting the background traffic and disconnect server-2 to emulate a server failure. We can see that TEA starts routing lookup

Network func.	TEA w/o cache		TEA w/ cache		Server-based	
	Lat. (μs)	Tput. (Mpps)	Lat. (μs)	Tput. (Mpps)	Lat. (μs)	Tput. (Mpps)
NAT	2.34	10.64	1.93	79.37	5.62	8.49
Stateful firewall	2.35	10.58	1.91	79.23	5.59	8.37
Load balancer	2.33	10.61	1.91	79.34	5.64	8.37
VPN gateway	2.30	10.80	1.92	79.45	5.99	8.25

Table 3: Throughput and latency of NFs implemented using TEA with a single server and corresponding software implementations running on a single server (4 CPU cores). Note that TEA does not involve the CPU on the server.



(a) NAT processing latency distribution. **(b) NAT throughput for real data center traces.**

Figure 11: Performance of NAT using TEA.

requests to server-1 as soon as it detects the event (at around 71 sec.). We observe that TEA can react to the changes in the link and server availability quickly despite a slight throughput drop at the time of failure.

6.2 NF Performance

Comparison with server-based NFs. We note that many factors including hardware configurations (e.g., number of CPU cores) and software optimizations can affect the performance of software-based NFs. Our goal here is to show the cost benefit of TEA by comparing the performance with the same hardware configuration (i.e., a server connected to a switch). For the evaluation, we implement NFs described in Table 2 using Click-DPDK [15] which is one of popular ways to implement high-performance NFs. We run them on the server described above.

For a fair comparison, we focus on a per-packet processing latency and throughput for 64 B packets with a single server for TEA and server-based NFs. We inject packets using 4 traffic generator nodes (max. traffic rate is ≈138 Mpps). Table 3 summarizes the results with median values for each experiment. Within each implementation option, there is no significant differences between NFs. Between TEA and server-based NFs, TEA shows up to 1.3× and 9.6× higher throughput, without and with the cache, respectively. For latency, TEA is up to 2.6× faster without cache and 3.1× faster with cache. TEA does not involve the server's CPU at all during the experiments while server-based NFs fully utilize 4 CPU cores. Note that with more CPU cores, the server-based implementations could achieve higher throughput, ideally, close to the NIC's raw performance (≈34 Mpps). Even compared to that case, TEA with cache can still achieve ≈2.3× higher throughput with much lower hardware cost since it does not involve the CPU.

Resource	Additional usage
Match Crossbar	12.6%
SRAM	8.5%
TCAM	0.4%
VLIW Instruction	4.2%
Hash Bits	6.3%

Table 4: Additional switch ASIC resources used by TEA.

Comparison with switch-based NFs. To understand the overhead that TEA incurs, we compare the performance of a specific NF, NAT, running on a programmable switch, when using TEA and when using local SRAM tables (referred as baseline). The results for other NFs are similar.

To measure latency, we replay both synthetic and real data center packet traces [20] consisting of 64 B packets. Note that since the real traces consist of varying sizes of packets, we make the payload size of each packet be 64 B with the original headers (i.e., the flow information is maintained). To measure the per-packet latency, we record two timestamps when packets come into the switch and leave the switch after the NAT processes the packet. Figure 11a shows the CDF of the latency distribution. The baseline and uniform represent the best and worst possible performance, respectively. We see that the more skewed the flow size distribution is, the lower the median latency is. Interestingly, we observe that the real traces show a skewness even higher than α=0.99. In the traces, top 95 popular flows take more than 50% of total flows), so the cache can serve more packets, lowering the median latency. Regardless of the skewness, we see that the variance is small (no long tail), resulting in the predictable latency.

To measure throughput, we replay real data center packet traces at the rate which is higher than the original rate at which it was captured. Since the packet sizes vary, we measure the throughput in Gbps rather than Mpps. A single packet generator node can replay the trace at 14.48 Gbps, thus the maximum transmission rate we could achieve is around 57.92 Gbps with our four packet generator nodes. Figure 11b shows the throughput of NAT with varied transmission rates. We see that NAT with TEA can serve the traffic at the incoming rate for all cases.

6.3 TEA ASIC Resource Usage

We evaluate how much ASIC resource is consumed *only* by TEA based on the P4 compiler's output. Note that as mentioned in §4.2.1, the number of colliding entries in TEA-Table that are stored in the SRAM is 0.1% of the total number of entries. Thus, the SRAM space usage depends on the total number of inserted entries, and in this evaluation, we insert 10 million entries. Table 4 shows the resource consumption. We see that there are plenty of resources remaining to implement other functionality on the ASIC along with TEA. It consumes some amount of SRAM, TCAM, VLIW instruction, and hash bits, all less than 9%. Match crossbar is the most consumed resource. We observe that count-min sketch, cache, stash, and lookup response handler consume most of the match crossbar. Memory address resolver and access requestor modules consume SRAM and hash bits to store metadata for RDMA connections and resolve bucket and server IDs.

7 Discussion

Deployment locations. As a starting point, we focus on designing TEA for ToR switches in NFV clusters. However, TEA can be deployed in other locations. In data center racks, one can enable TEA at ToR switches with compute servers. For that, we need to make sure that there is unused DRAM space in servers and link bandwidth. Moreover, our design can be extended to non-ToR switches (e.g., aggregation-layer switches) in data centers, which do not have directly connected servers under it. Since it requires multi-hop routing for lookup requests, we need to have a careful design that deals with longer and (possibly) unpredictable lookup latencies and unreliability. For example, with RoCEv2 protocol [38], which runs on top of IP/UDP and supports multi-hop routing, external DRAM access requests from upper-level switches can be routed to servers.

Match types. In this paper, we mainly focus on exact-matching semantics. Other NFs may require other lookup types such as longest-prefix matching (LPM). Previous work emulates LPM using exact-matching [65] or converts an LPM table into a large exact-match table [45]. We can leverage such ideas to support other lookup types in TEA.

Use cases. Although the current design of TEA-Table provides a key-value based table abstraction, we can extend it to support other use cases. For example, by adopting the FIFO queue abstraction, TEA allows utilizing external DRAM as a large packet buffer which can be useful for handling packet drops due to congestion.

Other programmable switch ASICs. While we use Tofino-based programmable switches for our implementation, we believe our design can be implemented on other switch ASICs since hardware capabilities leveraged in TEA (i.e., packet manipulation, meter, packet generation engine, etc.) are general features supported by most switch ASICs available today.

TEA using on-board off-chip DRAM. As mentioned earlier, some switch ASICs support on-board off-chip DRAM for specific purposes such as packet buffers and select lookup tables [8]. As the traffic demand increases, programmable switch ASIC vendors may also consider to adopt such on-board DRAM. However, to use DRAM in a flexible manner, they need to address the same practical challenges as the ones described in this paper, including asynchronous and low-latency DRAM access without stalling the packet processing pipeline. Thus, we believe that our techniques designed for TEA can be extended for such a future programmable switch architecture.

8 Related Work

Hardware-accelerated NFs. NF tasks have been accelerated using programmable switch ASICs, FPGAs, or Smart NICs to outperform CPU-only designs. Examples include offloading load balancers [53] and network monitoring [5, 35, 56] to switches and IPSec gateway, load balancer, and other NFs to FPGA-based smart NICs [31, 48]. TEA makes it possible to accelerate a wider range of NFs on programmable switches and support more operating scenarios by addressing the memory constraint issue.

Using external memory from switches. Prior work has suggested system architectures that allow switches to utilize external memory on servers [19, 44]. Such architectures run packet processing logic on both a hardware switch and a software switch on the servers and use servers' memory (i.e., accessing lookup tables on servers' memory) by forwarding a subset of packets (i.e., offloading traffic in certain conditions) to the software switch. This involves CPUs, increasing both average and tail packet processing latencies. In contrast, TEA purely uses DRAM on servers without involving CPUs via RDMA while addressing practical challenges in using multiple servers.

NFV state management. Previous work on state management for stateful NFs in NFV utilizes the local or remote storage to manage NF state [33, 41, 61, 66]. For example, statelessNF [41] allows NFs to leverage a centralized storage to store and load states for NFs. Their focus is better scaling and failure handling in the NFV context. In contrast, TEA leverages external DRAM to enable state-heavy NFs on programmable switches.

Other applications on programmable switches. Recent work has shown that it can be useful to offload other applications or primitives to programmable switches to enhance their performance. For example, offloading the sequencer [49], key-value cache [40, 50], and coordination service [39] improves the performance of distributed systems, in terms of throughput, scalability, and load balancing. Such systems also suffer due to switch memory constraints. TEA-like techniques could help such applications as well.

Accessing remote memory via RDMA. RDMA has been used in applications such as key-value stores [27, 42, 54], distributed shared-memory [27], transactional systems [23, 28, 46], and distributed NVM systems [52, 63]. Our work demonstrates a novel use of RDMA, which allows a programmable switch to leverage external DRAM on such servers.

9 Conclusions

While emerging programmable switch ASIC designs make it possible for moving NFs from commodity servers to switches, the limited memory on these ASICs has been a significant impediment in their use for many NFs. To address this issue, we envision a new system architecture, called TEA (Table Extension Architecture), for top-of-rack switch ASICs in NFV clusters. TEA provides a performant virtual table abstraction for NFs on programmable switches so that they can make use of DRAM on servers connected to the switch in a cost-efficient and scalable manner. Our evaluation with microbenchmarks and NF implementations shows that TEA can provide NFs with low and predictable latency and scalable throughput for table lookups without servers' CPU involvement.

Ethics: This work does not raise any ethical issues.

Acknowledgments

We would like to thank the anonymous SIGCOMM reviewers and our shepherd for their insightful comments and constructive feedback. This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by NSF award 1700521. Daehyeok Kim was also supported by the Microsoft Research PhD Fellowship.

References

[1] 2010. Data Set for IMC 2010 Data Center Measurement. http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html.
[2] 2011. 802.1Qbb – Priority-based Flow Control. https://1.ieee802.org/dcb/802-1qbb/.
[3] 2011. pktgen-dpdk: Traffic generator powered by DPDK. https://git.dpdk.org/apps/pktgen-dpdk/.
[4] 2015. Intel Xeon Processor E5-2640 v3. https://ark.intel.com/content/www/us/en/ark/products/83359/intel-xeon-processor-e5-2640-v3-20m-cache-2-60-ghz.html.
[5] 2018. Advanced Network Telemetry. https://www.barefootnetworks.com/use-cases/ad-telemetry/.
[6] 2018. Barefoot P4 Studio. https://www.barefootnetworks.com/products/brief-p4-studio/.
[7] 2018. Barefoot Tofino. https://www.barefootnetworks.com/products/brief-tofino/.
[8] 2018. BCM88690–10 Tb/s StrataDNX Jericho2 Ethernet Switch Series. https://www.broadcom.com/products/ethernet-connectivity/switching/stratadnx/bcm88690.
[9] 2018. Cavium Xpliant Ethernet Switches. https://www.cavium.com/xpliant-ethernet-switch-product-family.html.
[10] 2018. Perftest package. https://github.com/linux-rdma/perftest.
[11] 2019. Azure VPN Gateway. https://docs.microsoft.com/en-us/azure/vpn-gateway/vpn-gateway-about-vpngateways.
[12] 2019. Cisco Visual Networking Index. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-738429.html.
[13] 2019. Compare Kemp LoadMaster, F5 Big-IP & Citrix Netscaler. https://kemptechnologies.com/compare-kemp-f5-big-ip-citrix-netscaler-hardware-load-balancers/.
[14] 2019. EdgeCore Wedge 100BF-32X. https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=181&id=335.
[15] 2019. FastClick. https://github.com/tbarbette/fastclick.
[16] 2019. In-network DDoS Detection. https://www.barefootnetworks.com/use-cases/in-nw-DDoS-detection/.
[17] 2019. P4$_{16}$ Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.2.0.html.
[18] 2020. Recommended Network Configuration Examples for RoCE Deployment. https://community.mellanox.com/s/article/recommended-network-configuration-examples-for-roce-deployment.
[19] Mina Tahmasbi Arashloo, Pavel Shirshov, Rohan Gandhi, Guohan Lu, Lihua Yuan, and Jennifer Rexford. 2018. A Scalable VPN Gateway for Multi-Tenant Cloud Services. *SIGCOMM Comput. Commun. Rev.* 48, 1 (April 2018), 49–55.
[20] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *ACM IMC* (2010).
[21] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426.
[22] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95.
[23] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and General Distributed Transactions Using RDMA and HTM. In *EuroSys* (2016).
[24] Cisco. 2018. Cisco Global Cloud Index: Forecast and Methodology, 2016–2021 White Paper.
[25] Graham Cormode and Marios Hadjieleftheriou. 2008. Finding Frequent Items in Data Streams. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1530–1541.
[26] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *USENIX NSDI* (2018).
[27] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *USENIX NSDI* (2014).
[28] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *ACM SOSP* (2015).
[29] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2015. Maglev: A Fast and Reliable Software Network Load Balancer. In *USENIX NSDI* (2015).
[30] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. 2011. Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services. In *ACM SOCC* (2011).
[31] 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. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX NSDI* (2018).
[32] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. 2014. Duet: Cloud Scale Load Balancing with Hardware and Software. In *ACM SIGCOMM* (2014).
[33] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling Innovation in Network Function Control. In *ACM SIGCOMM* (2014).
[34] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *ACM SIGCOMM* (2016).
[35] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven Streaming Network Telemetry. In *ACM SIGCOMM* (2018).
[36] J. Heinanen and R. Guerin. 1999. *A Two Rate Three Color Marker.* RFC 2698. RFC Editor.
[37] Infiniband Trace Association. 2010. Supplement to InfiniBand architecture specification volume 1 release 1.2.1 annex A16: RDMA over converged ethernet (RoCE).
[38] Infiniband Trace Association. 2010. Supplement to InfiniBand architecture specification volume 1 release 1.2.1 annex A17: RDMA over converged ethernet (RoCE).
[39] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *USENIX NSDI* (2018).
[40] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *ACM SOSP* (2017).
[41] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *USENIX NSDI* (2017).
[42] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-value Services. In *ACM SIGCOMM* (2014).
[43] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *ACM STOC* (1997).
[44] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. 2016. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *ACM SOSR* (2016).
[45] Changhoon Kim, Matthew Caesar, Alexandre Gerber, and Jennifer Rexford. 2009. Revisiting Route Caching: The World Should Be Flat. In *PAM* (2009).
[46] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. Hyperloop: Group-based NIC-offloading to Accelerate Replicated Transactions in Multi-tenant Storage Systems. In *ACM SIGCOMM* (2018).
[47] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan. 2018. Generic External Memory for Switch Data Planes. In *ACM HotNets* (2018).
[48] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *ACM SIGCOMM* (2016).
[49] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *USENIX OSDI* (2016).
[50] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *USENIX FAST* (2019).
[51] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *ACM SIGCOMM* (2016).
[52] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *USENIX ATC* (2017).
[53] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *ACM SIGCOMM* (2017).
[54] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX ATC* (2013).
[55] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *ACM SIGCOMM* (2015).

[56] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *ACM SIGCOMM* (2017).
[57] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. 2010. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review* 43, 4 (2010), 92–105.
[58] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo hashing. In *European Symposium on Algorithms* (2001).
[59] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. 2013. Ananta: Cloud Scale Load Balancing. In *ACM SIGCOMM* (2013).
[60] S. Pontarelli, P. Reviriego, and M. Mitzenmacher. 2018. EMOMA: Exact Match in One Memory Access. *IEEE Transactions on Knowledge and Data Engineering* 30, 11 (Nov 2018), 2120–2133.
[61] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *USENIX NSDI* (2013).
[62] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *ACM SIGCOMM* (2015).
[63] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed Shared Persistent Memory. In *ACM SoCC* (2017).
[64] David G. Thaler and Chinya V. Ravishankar. 1998. Using Name-based Mappings to Increase Hit Rates. *IEEE/ACM Trans. Netw.* 6, 1 (Feb. 1998), 1–14.
[65] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. 1997. Scalable High Speed IP Routing Lookups. In *ACM SIGCOMM* (1997).
[66] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. 2018. Elastic Scaling of Stateful Network Functions. In *USENIX NSDI* (2018).
[67] Dong Zhou. 2019. *Data Structure Engineering for High Performance Software Packet Processing.* Ph.D. Dissertation. Carnegie Mellon University.

Note: Appendices are supporting material that has not been peer-reviewed.

A Psuedocode for TEA-Table Operations

Algorithm 1: `Insert(key,value)` for TEA-Table (Control plane).

```
1  $tries$=0;
2  $entry$=($key$,$value$);
3  while $tries < MaxTries$ do
       /* Temporaily store the entry in SRAM during insertion */
4      insert $entry$ to SRAM Stash;
5      $i$=hash($key$);
6      if bucket[i] has an empty cell then
7          insert $entry$ to the cell;
8          remove $entry$ from SRAM;
9          copy the cell to the shadow table;
10         return Done;
11     $j$=($i$+1) % $n$;
12     if bucket[j] has an empty cell then
13         insert $entry$ to the cell;
14         remove $entry$ from SRAM;
15         copy the cell to the shadow table;
16         return Done;
17     select a random cell $c$ from bucket[$i$] $\cup$ bucket[$j$];
18     $victim$=$c.entry$;
19     insert $entry$ to $c$;
20     remove $entry$ from SRAM stash;
21     $entry$=$victim$;
22     $tries$++;
```

Algorithm 2: `Lookup(key)` for TEA-Table (Data plane).

```
1  if key exists in SRAM Stash or Cache then
2      return ($SRAM[key]$,$packet$);
3  $i$=hash($key$);
   /* Resolve memory address of the bucket */
4  $addr$=resolve_addr($i$);
   /* Write the packet to the scratchpad */
5  RDMA_Write ($addr$+$KV\_LEN$, $packet$, $packet\_length$);
6  $length$=$KV\_LEN$+$packet\_length$;
   /* Read the bucket and packet */
7  ($kv\_cells$, $packet$) = RDMA_Read ($addr$, $length$);
8  Lookup response handler:
9      Upon receive lookup response packet
10         return ($kv\_cells$[$key$], $packet$);
```

B Simplified P4 codes of NF implementations with TEA

As mentioned in §5, we implement the TEA APIs in P4 and expose them as modularized P4 codes [17, §13]. Figure 12 shows an example program written in P4 using the TEA APIs. Control blocks provided by TEA, including `LookupHandler`, `ServerResolver`, `MemResolver`, and `LookupRequestor` are used in the ingress or egress pipeline, along with the NF logic. Extending this template, developers can integrate TEA with their NF implementations. Based on the template, we implement NAT, stateful firewall, load balancer, and VPN gateway, described in §5 and below are the simplified P4 codes of the NFs.

```
#include "tea_core.p4"
control Ingress (headers hdr, metadata meta) {
    LookupHandler() lookup_handler;
    ServerResolver() server_resolver;
    MemResolver() mem_resolver;
    apply {
        lookup_handler.apply(hdr, meta);
        if (meta.lookup_md.found == true) {
            [Ingress NF logic]
        } else {
            server_resolver.apply(meta);
            mem_resolver.apply(meta);
        }
    }
}
control Egress (headers hdr, metadata meta) {
    LookupRequestor() lookup_req;
    apply {
        if (meta.lookup_md.found == true) {
            [Egress NF logic]
        } else {
            lookup_req.apply(hdr, meta);
        }
    }
}
```

Figure 12: A template of P4 program using TEA abstraction. TEA exposes as a library of P4 control functions (e.g., `lookup_response_handler`).

```
#include "tea_core.p4"
...
control Ingress (headers hdr, metadata meta) {
    ...
    action nat_ext_to_int () {
        hdr.ipv4.dstIP = meta.lookup_md.pip;
        hdr.ipv4.dstPort = meta.lookup_md.pport;
    }
    ...
    table nat {
        key = {
            meta.lookup_md.dir: exact;
        }
        actions = {
            nat_ext_to_int;
            nat_int_to_ext;
            drop;
        }
    }
    ...
    LookupHandler() lookup_handler;
    ServerResolver() server_resolver;
    MemResolver() mem_resolver;
    apply {
        ...
        lookup_handler.apply(hdr, meta);
        if (meta.lookup_md.found == true) {
            nat.apply();
            forward.apply();
        } else {
            server_resolver.apply(meta);
            mem_resolver.apply(meta);
        }
        ...
    }
}
control Egress (headers hdr, metadata meta) {
    LookupRequestor() lookup_req;
    apply {
        if (meta.lookup_md.found == false) {
            lookup_req.apply(hdr, meta);
        }
    }
}
```

Figure 13: NAT

```
#include "tea_core.p4"
...
control Ingress (headers hdr, metadata meta) {
    ...
    LookupHandler() lookup_handler;
    ServerResolver() server_resolver;
    MemResolver() mem_resolver;
    apply {
        ...
        lookup_handler.apply(hdr, meta);
        if (is_ext.apply().hit) { //packet from external?
            if (meta.lookup_md.found == true) {
                forward.apply();
            } else {
                if (meta.lookup_md.remote_miss == false) {
                    server_resolver.apply(meta);
                    mem_resolver.apply(meta);
                }
            }
        }
        ...
    }
}
control Egress (headers hdr, metadata meta) {
    LookupRequestor() lookup_req;
    apply {
        if (meta.lookup_md.found == false &&
            meta.lookup_md.remote_miss == false) {
            lookup_req.apply(hdr, meta);
        }
    }
}
```

Figure 14: Firewall

```
#include "tea_core.p4"
...
control Ingress (headers hdr, metadata meta) {
    ...
    action update_server_addr () {
        hdr.ipv4.dstIP = meta.lookup_md.serverIP;
    }
    ...
    LookupHandler() lookup_handler;
    ServerResolver() server_resolver;
    MemResolver() mem_resolver;
    apply {
        ...
        lookup_handler.apply(hdr, meta);
        if (meta.lookup_md.found == true) {
            update_server_addr();
            forward.apply();
        } else {
            server_resolver.apply(meta);
            mem_resolver.apply(meta);
        }
        ...
    }
}
control Egress (headers hdr, metadata meta) {
    LookupRequestor() lookup_req;
    apply {
        if (meta.lookup_md.found == false) {
            lookup_req.apply(hdr, meta);
        }
    }
}
```

Figure 15: Load balancer

```
#include "tea_core.p4"
...
control Ingress (headers hdr, metadata meta) {
    ...
    action encap_packet () {
        hdr.l3_tunnel.setValid();
        hdr.out_ipv4.setValid();
        ...
        hdr.l3_tunnel.id = meta.lookup_md.l3_tun_id;
        hdr.out_ipv4.dstIP = meta.lookup_md.serverIP;
        ...
    }
    ...
    LookupHandler() lookup_handler;
    ServerResolver() server_resolver;
    MemResolver() mem_resolver;
    apply {
        ...
        lookup_handler.apply(hdr, meta);
        if (is_ext.apply().hit) { //packet from external?
            if (meta.lookup_md.found == true) {
                encap_packet();
                forward.apply();
            } else {
                server_resolver.apply(meta);
                mem_resolver.apply(meta);
            }
        }
        ...
    }
}
control Egress (headers hdr, metadata meta) {
    LookupRequestor() lookup_req;
    apply {
        if (meta.lookup_md.found == false) {
            lookup_req.apply(hdr, meta);
        }
    }
}
```

Figure 16: VPN gateway