# Towards Elastic and Resilient In-Network Computing

Daehyeok Kim

October 2021

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Thesis Committee:**
Srinivasan Seshan, Co-chair
Vyas Sekar, Co-chair
Justine Sherry, Carnegie Mellon University
Jennifer Rexford, Princeton University
Jitendra Padhye, Microsoft

*Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.*

**Keywords:** In-network computing, Programmable networks, Programmable data planes, Fault Tolerance

## Abstract

Recent technology advances in programmable network data plane devices such as programmable switches and smart network interface cards creates a new computing paradigm called in-network computing. While many recent works have demonstrated the promise of in-network computing, we observe that there is a huge gap between what in-network computing offers today and evolving application demands. In particular, we believe that resource elasticity and fault resilience are essential for any practical computing platform but missing in today's view of in-network computing. While the application workloads are evolving, today's in-network computing only supports a simple deployment model where a single application runs on a single device equipped with limited resources. Also, although device failures becomes prevalent and are critical for correctness and performance of applications, it has gained little attention.

While resource elasticity and fault resilience have been extensively studied for traditional server-based computing, enabling them on programmable data plane devices is challenging, especially due to their hardware constraints. First, unlike servers equipped with a hierarchy of resources, the data plane devices only have limited resources and capabilities, which are fixed at a hardware design time, making it hard to utilize resources in an elastic way. Second, limited programmability with low-level primitives such as bit-level packet manipulation makes it hard to implement existing mechanisms designed for servers in the data plane. Lastly, even if we can somehow implement the mechanisms in the data plane, since the device needs to process a stream of packets at a very high rate (*e.g.,* a few tens Tbps in a switch), it is challenging to run the mechanisms without affecting the performance.

In this thesis, we argue that by designing abstractions that effectively expose resources and capabilities available *outside* a single type of device while hiding the complexities of dealing with heterogeneity, we can make in-network computing more elastic and resilient without any hardware modifications. In particular, we design three systems for in-network applications written for programmable switches, each of which consists of key abstraction, P4-based programming API, and runtime environment. TEA provides a virtual table abstraction that gives an illusion of large match-action tables by effectively utilizing DRAM available on remote servers, ExoPlane offers an augmented switch resource architecture to support concurrent applications with an one big switch with large resource abstraction, and RedPlane provides an one big fault-tolerant switch abstraction for an application deployed on multiple switches in the network. Putting the three pieces all together, with our abstractions and runtime environments, developers can easily enable resource elasticity and fault resilience for their applications without worrying about underlying complexities.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The traditional view of the network as a "dumb pipe" that merely transmits bits of data from one application end-point to the other is long gone. Today, the network, especially its data plane, becomes more programmable, which allows us to implement sophisticated functionality beyond packet forwarding inside the network. The key enabler for this is recent technology advances in programmable network data plane devices. For example, programmable switching Application-Specific Integrated Chips (ASICs) provide limited data plane programmability while ensuring packet processing rates of 10s of Tbps or a few billion packets per second [32, 44, 46, 47], and programmable Network Interface Cards (NICs) (or Smart NICs) with a Network Processing Units (NPUs) or Field Programmable Gate Arrays (FPGAs) also support data plane programmability [35, 36, 37].

This technology advance creates a new computing paradigm called *in-network computing*, which enables to run various functionalities that traditionally have been served by CPU servers or proprietary hardware devices, ranging from network middleboxes to distributed systems (*e.g.,* [63, 90, 91, 115, 120, 129, 149, 160]) on network data plane devices. We observe that this new form of computing becomes promising today because user demands and the technology advances have converged in a timely manner. Today, while network bandwidth is growing rapidly up to several hundreds of Gbps, the speed of CPU performance improvement has decreased with the slowing down of Moore's law. This can cause low and unpredictable application performance such as high tail latency. On the other hand, programmable data plane devices can process data at high rate while providing predictable performance, in-network computing can provide opportunities for improving the performance of applications, as well as reduce operational costs.

Despite many recent efforts that have demonstrated the promise of in-network computing, we observe that there are still many missing pieces to make the programmable data plane as a practical and future-proof computing platform, especially to support evolving application workloads. In particular, as we will describe later in this chapter, while *resource elasticity* and *resilience against device failures* are essential for any computing platforms, they have considered in an ad-hoc manner or gained little attention in today's in-network computing. Although the resource elasticity and resilience have been extensively studied in the context of traditional server-based computing, we find that it is uniquely challenging to support them for in-network computing, due to fixed and limited resource and capability within a device, limited programmability with low-level primitives, and high operation speed of the device.

In this thesis, we argue that by designing abstractions that effectively expose resources and capabilities available *outside* a single type of data plane device (*e.g.,* a single switch box) while hiding the complexities of dealing with heterogeneity, we can make in-network computing more elastic and resilient without any hardware modifications. This is based on our observation that although each type of device has certain limitations in terms of available resources and capabilities, different types of devices can be complementary with each other. For example, while a programmable switch is capable of processing packets at the rate of Tbps with only a few 10s MB of SRAM, a smart NIC can provide lower processing capacity with a few GB of DRAM.

We particularly focus on enabling resource elasticity and fault resilience for in-network applications running on programmable switches, also called *in-switch applications*, which have the most tight constraint on resource and capability among programmable data plane devices available today, while leveraging resources on other types of *external* devices (*e.g.,* smart NICs and commodity CPU servers). In particular, we design three new architectures for in-switch applications, each of which consists of key abstraction, P4-based programming API, and runtime environment. At a high-level, TEA (Table Extension Architecture) provides a virtual table abstraction that gives an illusion of large match-action tables by effectively utilizing DRAM available on remote servers (*elastic memory*), ExoPlane proposes an augmented switch resource architecture to support multiple concurrent applications with a one big switch with large resource abstraction (*elastic compute and memory*), and RedPlane provides an one big fault-tolerant switch abstraction for an application deployed on multiple switches in the network (*fault resilience*). Putting the three pieces all together, it realizes elastic and resilient in-network computing.

---

**Thesis statement:** *With the right abstractions and runtime environments for programmable data planes, in-network computing can become more elastic and resilient.*

---

In the remainder of this chapter, we describe today's view of in-network computing, our vision of elastic and resilient in-network computing, challenges in realizing the vision, and then highlight our key results.

## 1.1 Today's Myopic View of In-Network Computing

With the technology advance in programmable data planes, many recent efforts have shown the feasibility and efficiency of various applications running on programmable data plane devices, including network monitoring and telemetry [63, 116, 149], DDoS defenses [115, 160], key-value store caches [90, 113], network middlebox functions [120, 129, 134], consensus protocols [91, 107], and others [140, 152]. Today, network administrators or application developers program the data plane devices like in early days in server-based computing; they program and deploy *a single program* on *a single device* within a given fixed and limited physical compute and memory resources. If the program requires more than available resources, it will fail to compile and run.

While the current way of enabling in-network computing might be enough for running a single program with fixed and small-sized workloads, we argue that it can significantly limit the potential of in-network computing. We observe that there is a huge gap between evolving application workload demands and the current *myopic* view of in-network computing. Although the number of in-network applications that can potentially run concurrently keep increasing [85, 99] and the amount of traffic workload also grows [34, 65], today, only a single program can running on a single device under severe resource constraints (*e.g.,* 10s MB SRAM on programmable switches) which is hard to be extended without significant hardware modifications. Thus, if applications' workload demands exceed the constrained resource capacity on a single device, the applications will fail to run. In addition to the resource constraint, while several measurement studies in production networks have shown the prevalence of networking hardware failures [79, 112, 119], which can affect performance and correctness of applications, handling such failures has received little attention or considered in an ad-hoc manner.

Although the resource elasticity and fault resilience are fundamental building blocks for any practical and future-proof computing platform, they have not been considered in today's in-network computing, significantly limiting its potential.

## 1.2 A Vision for Elastic and Resilient In-Network Computing

In this thesis, we envision an in-network computing platform that natively supports resource elasticity and fault resilience for in-network applications, rather than letting network administrators and developers constraint their workload sizes or manually handle failures in application-specific ways. In fact, supporting resource elasticity and fault resiliency for a computing platform is not a new problem; they have been extensively studied for traditional server-based computing, especially in the context of operating systems and distributed systems. For example, while in early days in server-based computing, people have to write a program within limited memory space, with the advent of virtual memory abstraction provided by operating systems, they can program more easily with an illusion of large memory space. Also, handling failures is typically very complicated and error-prone, abstractions such as replicated state machine provides an illusion of a single, high available system. Such abstractions enables elastic and resilient computing while hiding the complexities of how it actually works. Then, the natural question is: *Can we directly apply these existing techniques designed for server-based computing to in-network computing?*

Unfortunately, we find that there are many unique challenges in supporting the resource elasticity and fault resilience for in-network computing, which makes it difficult to apply the existing techniques. First, unlike CPU servers which have a hierarchy of resources (*e.g.,* CPU caches, main memory, and disks for memory), the current programmable data plane devices, especially programmable switches which is the context of this thesis, only have limited resource and capability within a device, which are fixed at a hardware design time. Thus, there seems no way to utilize resources in an elastic way. Second, limited programmability with low-level primitives such as bit-level packet header manipulation in P4 language makes it hard to implement existing complicated mechanisms in the device data plane, which are designed for more capable CPU servers with high-level primitives. Lastly, even if we can somehow implement the mechanisms in the device data plane, since the data plane needs to process a stream of packets at a very high rate (*e.g.,* a few tens Tbps in the switch data plane), it is challenging to run the mechanisms without affecting the performance.

Fortunately, we find an opportunity for addressing the first challenge with the fixed and limited amount of resources on a single device, which seems the most fundamental obstacle to realizing our vision. We observe that although each type of device has certain limitations in terms of available resources and capabilities, if we look at outside a single device, there are other types of data plane devices accessible through the network, and different types of devices can be complementary with each other. For example, while a programmable switch is capable of processing packets at the rate of Tbps with only a few 10s MB of SRAM, a smart NIC provides lower processing capacity with a few GB of DRAM. Thus, if we can somehow let applications running on one type of device (*e.g.,* a switch) utilize larger resources on another type of device (*e.g.,* DRAM on a smart NIC), that can be a potential solution to enable elastic resource and potentially provide a way of supporting resilience as well.

Although this approach sounds promising, realizing it entails several practical challenges. In particular, since applications running on a switch now needs to access resources on different types of devices (*e.g.,* switches, NICs, and CPU servers) connected through the network, achieving high performance while not affecting behavior of applications and hiding the complexities from programmers is challenging. [TODO: Expand challenges]

## 1.3 Summary of Contributions

This thesis presents three novel systems that realize resource elasticity and fault resilience by providing new abstraction and runtime environments that expose resources on external devices for in-network applications running on the switches:

**TEA** is a system designed to offer *elastic memory* for state-intensive applications. It consists of a programmable switch and multiple servers offering DRAM and provides an illusion of large match-action tables for applications running on programmable switches via a new abtraction called virtual table abstraction. TEA demonstrates:

- The feasibility of table lookups on available DRAM on remote servers entirely in the data plane with low and predictable latency (1.8–2.2 $\mu s$) while not involving CPUs on servers.
- The feasibility of providing resource elasticity in terms of table size and access bandwidth that can be scaled linearly by recruiting more servers (*e.g.,* 138 million lookups per second with 8 servers in our testbed).
- The cost and performance benefits of TEA-enabled applications compared to the same applications running on servers.

We present TEA in Chapter 3.

**ExoPlane** presents an augmented switch resource architecture designed to offer *elastic compute and memory* for multiple concurrent applications. It consists of a programmable switch and multiple external data plane devices, and provides an OS-like abstraction that effectively multiplexes applications across the switch and external devices. ExoPlane shows:

- The feasibility of multiplexing concurrent applications targeting for a switch across the multiple devices with minimal impacts on performance while satisfying multiple per-app and cross-app requirements.

- The effectiveness of abstractions for resource and state management that allow to hide the complexities of dealing with heterogeneity from network administrators and application developers.
- The cost benefit of the rack-level resource augmentation architecture compared to other alternatives.

We present ExoPlane in Chapter 4.

**RedPlane** is a system designed to offer *fault resilience* for applications running on programmable switches. It consists of multiple programmable switches where the same application is running and servers offering in-memory state store. It provides a one big fault-tolerant switch abstraction that gives an illusion of a single, highly-available switch. RedPlane illustrates:

- The definition of a new correctness model based on the traditional notion of *linearizability* for in-switch applications and its realization with RedPlane protocol.
- The feasibility of implementing RedPlane protocol entirely in the data plane, which correctly replicating state to external state store for different types of applications (read-centric, write-centric, read/write-mixed).
- The performance benefit compared to alternative approaches by showing negligible per-packet latency overhead for read-centric applications and less than 8 $\mu s$ overhead even for the worst case and fast recovery of end-to-end TCP throughput within a second.

We present RedPlane in Chapter 5.

## 1.4 Dissertation Plan

This thesis proceeds as follows. In Chapter 2, we discuss background and related work, including the history of programmable networks, programmable data plane technology, and in-network computing and applications. In Chapter 3, we present TEA for state-intensive in-network applications, which provides a virtual table abstraction. In Chapter 4, we discuss ExoPlane, an operating system for an augmented resource architecture to support multiple concurrent stateful in-network applications. In Chapter 5, we discuss RedPlane, a framework that makes in-network applications tolerant to switch failures by providing one big fault tolerant switch abstraction. Finally, in Chapter 6 we discuss and conclude.

# Chapter 2

# Background and Related Work

In this chapter, we give an overview and background for in-network computing, including a history of network programmability, key technology enablers for programmable data planes, and the concept of in-network computing and its popular applications. Readers familiar with the basics of programmable networks and relevant hardware technologies such as such as programmable switch architectures can skip ahead to Chapter 3.

## 2.1 From Active Networks to Programmable Networks

Over the past two decades, both the research community and computer networking industry have made various efforts to make computer networks more programmable. With the advent of the concept of control-data plane separation [154] and OpenFlow-based software-defined networking (SDN), network operators can modify fixed-function data plane's policies such as packet forwarding rules through the control plane interface. However, despite the Active networks [150] and the following research [50, 62, 158] on the data plane programmability, it has not been practical to program the functionality of the data plane. The critical roadblock was that the hardware technology for the data plane was not able to achieve efficient simultaneous programmability and performance.

The vision of the programmable data plane, which seemed impossible, has been made possible by recent advances in networking hardware technology. For example, programmable switch Application-Specific Integrated Chips (ASICs) now can provide limited data plane programmability while ensuring packet processing rates of 10s of Tbps [32, 46]. Also, programmable Network Interface Cards (NICs) (or Smart NICs) with a network processing units (NPUs) or Field Programmable Gate Arrays (FPGAs) also support data plane programmability [35, 36, 37]. These hardware technologies become key enablers that allow the data plane to perform computation directly on the packet beyond its traditional role, packet forwarding, in the network. We call this form of computation *in-network computing*. It creates many new opportunities to improve the performance of networking and distributed systems applications. Because of this, cloud service providers such as Microsoft [74] and Amazon [33] have also deployed these devices in their data centers which now become a collection of heterogeneous programmable data plane devices.

[TODO: Restructure it]

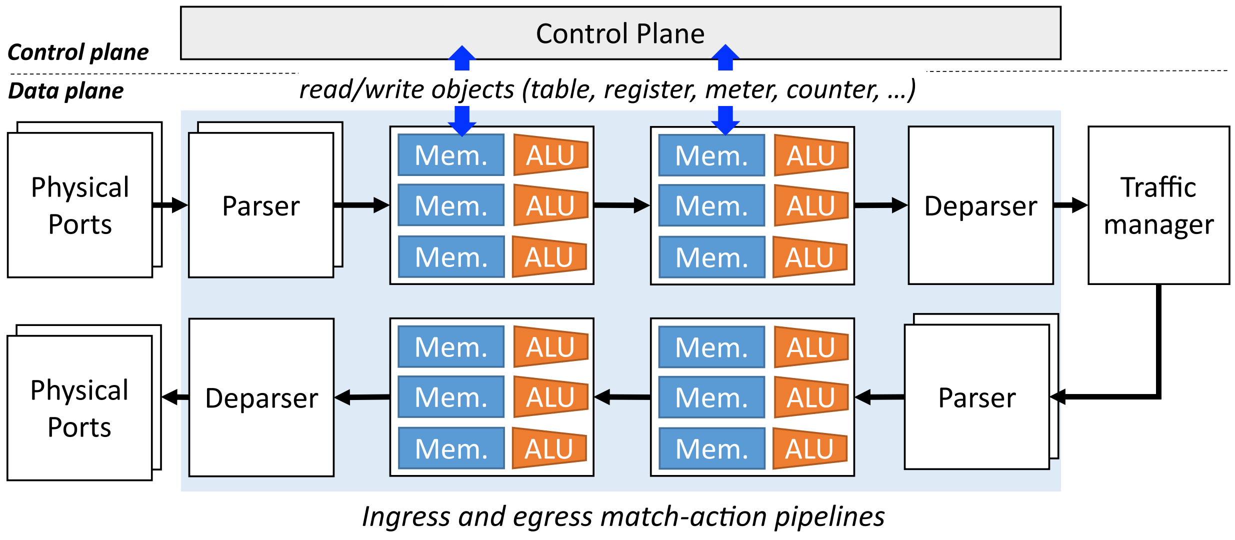


**Figure 2.1:** Conceptual view of programmable switch architecture.

## 2.2 Programmable Data Plane Technology

[TODO: Preamble]

**Programmable switches.** Programmable switch architectures used today, *e.g.,* Intel Tofino [46], use a limited amount of on-chip memory (*e.g.,* SRAM and TCAM) to provide a variety of stateful object abstractions, including tables, registers, meters, and counters. Applications can use these to keep state across multiple packets, such as the address translation table in the NAT example above. In the ingress and egress match-action pipeline, objects are allocated in each stage and accessed by packets via ALUs. These objects are also accessible by the switch control plane through the ASIC-to-CPU PCIe channel which has a limited bandwidth ($O$(10 Gbps)) compared to the ASIC's per-port bandwidth ($O$(100 Gbps)). In addition, the ASIC provides other built-in functionality such as packet replication, recirculation, and mirroring for more advanced packet processing.

**Smart NICs.**

**Packet processing on commodity CPU servers.**

## 2.3 In-Network Computing and Applications

# Chapter 3

# Table Extension Architecture for State-Intensive In-Network Applications

Network functions (NFs) are an essential component in today's on-line 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)) [72, 130]. More recently, another approach has been gaining attention in the community: NFs implemented on programmable switch ASICs (e.g., [3, 27, 120]).

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 §3.2.1, 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]

[1]Our recent position paper proposes this high-level idea [101]. However, that work fails to tackle these technical challenges and falls short of providing a concrete proof-of-concept realizing the architecture.

- 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 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 $\mu s$), 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 [128] 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 [164], 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 [94]) 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 [38] 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 $\mu s$), 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$\times$ higher throughput and 3.1$\times$ 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.

| | **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 3.1:** Comparison of NF deployment options. We excerpt the information from product briefs [6, 9, 46] and prior work [120, 130].

## 3.1 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.

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) [54, 72, 130], and (3) implementing them on emerging programmable switches [32, 46]. We note that while there are other options such as implementing NFs on FPGA boards attached to servers (e.g., [74]), 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 3.1 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 $\approx$1 MB per-flow state).** This regime can be served by using any of three options. While supporting 100 Gbps traffic would require 3 hardware appliances ($\sim$\$120K), or 10 servers ($\sim$\$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 $\approx$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

[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.

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 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 [46], 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.

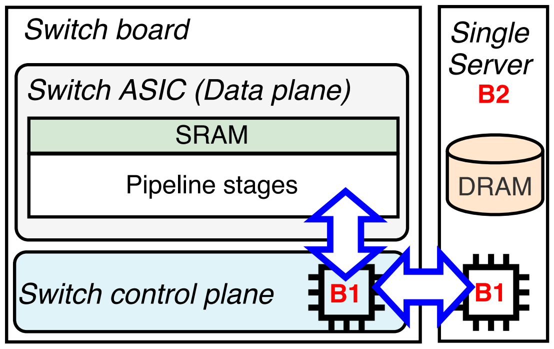
In summary, our analysis suggests that: (1) servers and appliances can handle the low-bandwidth regime effectively, (2) programmable switches are great when flow-state fits in the limited SRAM space, and (3) nothing handles the most demanding workloads well. Ideally, if we could build an architecture 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.

## 3.2 Design Space and Challenges

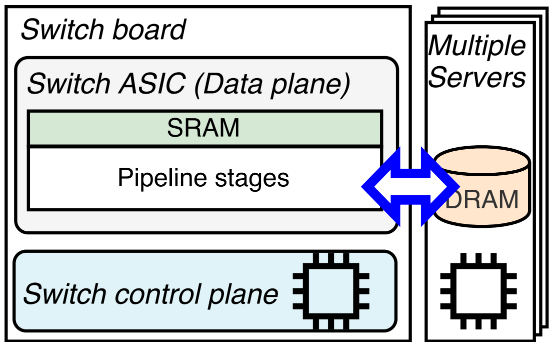
Building on the above analysis, we explore if and how we can potentially leverage external DRAM that already exists in the network. Now, there are two places where we can naturally find available DRAM *near* the switch ASIC:

**(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 [39]).

**(2) On-board off-chip DRAM.** Some switch ASIC vendors have added custom off-chip DRAM on the switch board [29]. 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 §3.6.



(a) Naïve design and performance bottlenecks (B1 and B2).



**(b)** TEA enabling to access external DRAM in the data plane without CPU involvement.

**Figure 3.1:** Comparison between RPC-based naïve design and TEA to access external DRAM.

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 [34, 65].

**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 [54, 72, 75, 130] 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.

### 3.2.1 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 [92, 127] (Figure 3.1a). 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 [100].

**(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 3.1a). 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 [128]) 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?

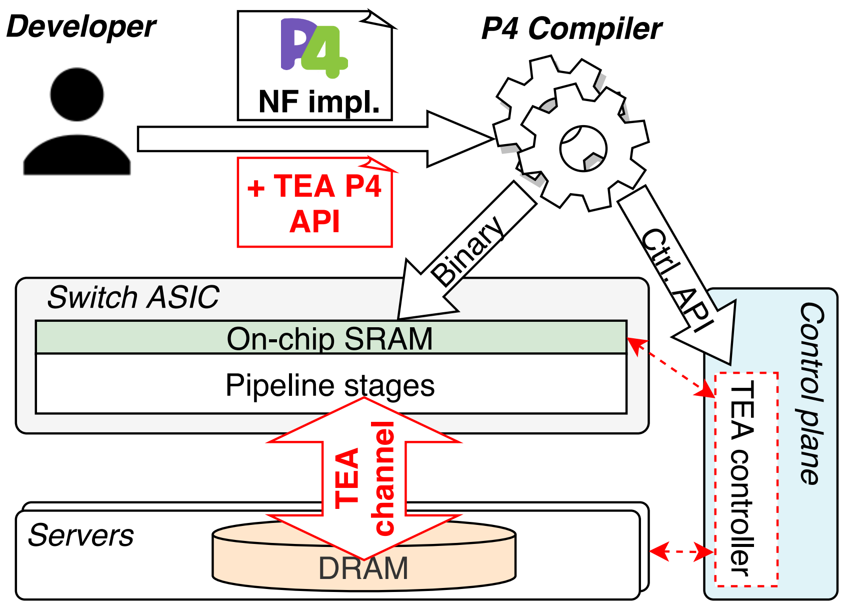


**Figure 3.2:** NFs implemented in P4 can be extended with TEA P4 API to look up tables across external DRAM and on-chip SRAM. The control plane is (dotted lines) involved when establishing a TEA channel.

## 3.3 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 3.2 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 [38, §13]. 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 §3.2.1:

1. *Leveraging ASIC programmability* to enable simplified RDMA in the data plane (§3.3.1).
2. *Repurposing bounded linear probing* to guarantee hash table lookups in a single-round trip to external DRAM (§3.3.2).
3. *Offloading packet store to external DRAM* to enable asynchronous lookups (§3.3.2).
4. *Leveraging the small-cache theory [73]* to scale out the throughput (§3.3.3).

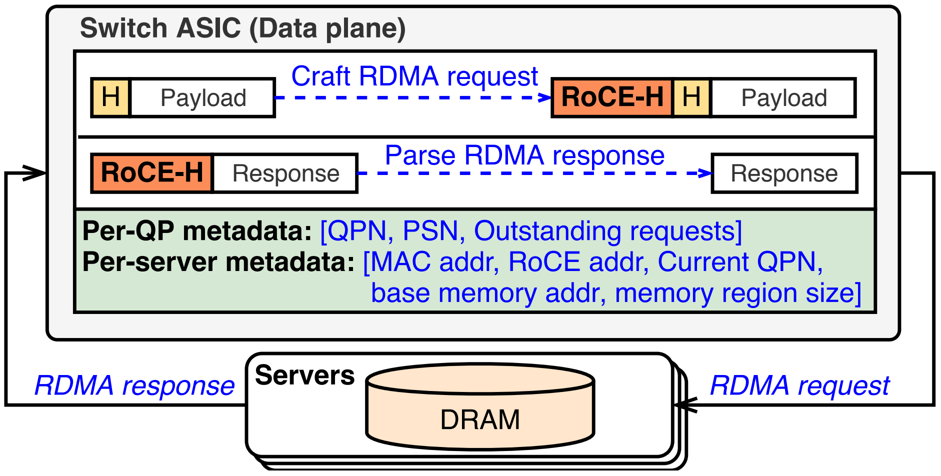

Switch ASIC (Data plane)
H
Payload
Craft RDMA request
RoCE-H
H
Payload
RoCE-H
Response
Parse RDMA response
Response
Per-QP metadata: [QPN, PSN, Outstanding requests]
Per-server metadata: [MAC addr, RoCE addr, Current QPN, base memory addr, memory region size]
RDMA response
Servers
DRAM
RDMA request

**Figure 3.3:** 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.

5. *Repurposing ASIC's hardware capabilities* to detect and react to sever availability changes in the data plane (§3.3.4).

### 3.3.1 DRAM Access in the Data Plane

To access external DRAM, we choose RDMA, which is quite common in service provider deployments [82, 122]. 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 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 [89**?** ], 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 3.3 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 [23]. 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 §3.4.3.[4] Alternatively, we can also configure a higher QoS-level for our RDMA traffic over lossy fabric [16]. 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.

### 3.3.2 TEA-TABLE: Lookup Table Structure

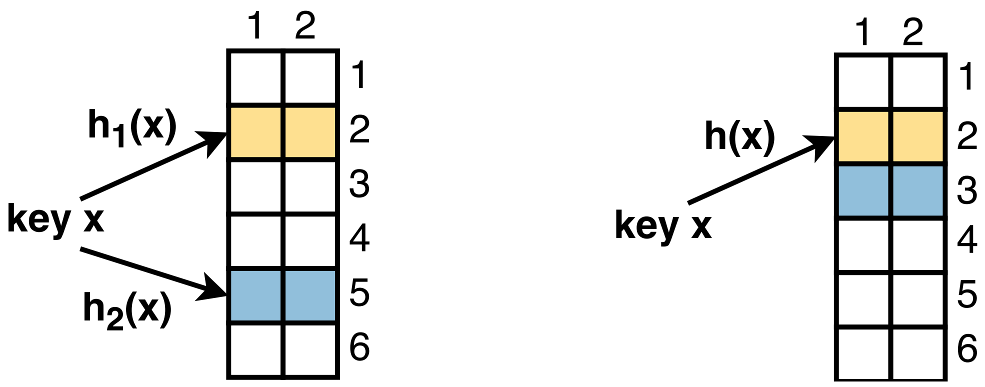
The design of TEA's table data structure, TEA-TABLE, addresses two key issues: (1) how to complete a lookup in a single round-trip to external DRAM and (2) how to defer processing of the current packet until the lookup completes and continue processing other packets without blocking. TEA-TABLE repurposes a data structure that was originally designed for improving cache hit rates in software switches [164] to achieve single RTT lookups and incorporates remote packet buffers within the data structure to accommodate deferred packet processing.

#### 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

[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.



**(a)** Cuckoo hashing. **(b)** Bounded linear probing.

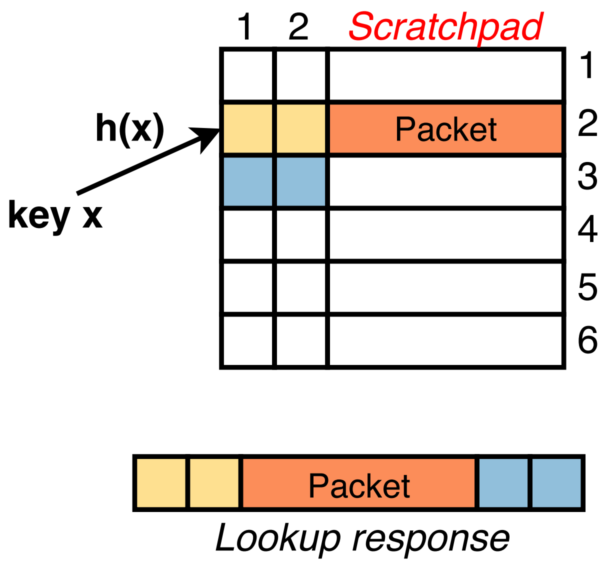
**Figure 3.4:** Cuckoo hashing and bounded linear probing. In this example, there are 6 buckets and 2 cells per bucket. The numbers on the top and right side indicate cell and bucket indices, respectively.

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.

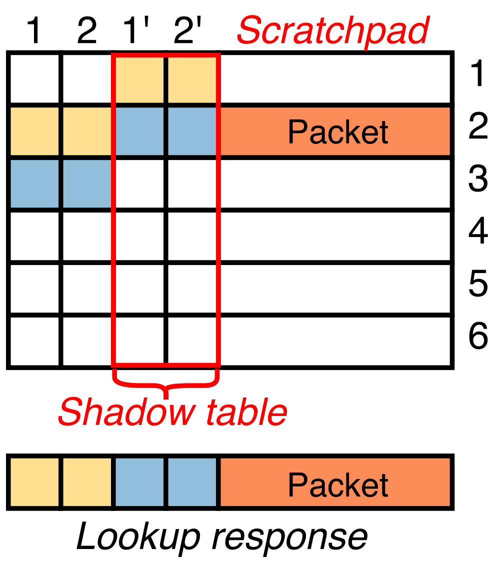
**Strawman solutions.** At first glance, it appears we can use traditional hashing techniques. Indeed, many modern switch ASICs adopt variations of cuckoo hashing [128] for exact-match lookups in SRAM as it guarantees constant-time lookup. A caveat, however, is that each lookup requires multiple memory 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 [132], uses additional Bloom filters [59] 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]

**Our approach.** We build on a recent approach called Bounded Linear Probing (BLP) [164]. 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 3.4 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 3.4a), BLP uses one hash function and lets the second bucket be placed right next to the first bucket (Figure 3.4b).

[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)** Incorrect design: switch cannot parse entries in blue cells.



**(b)** Corrected design with shadow table.

**Figure 3.5:** 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).

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.

**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 3.5, 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 3.5a. 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 3.5b, 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.

## 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 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 §3.3.3), 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

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

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.

### 3.3.3 Multiple DRAM Servers

Recall from §5.2, 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 [94] and rendezvous hashing [151] 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 [73]. 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 [67], for which efficient switch data plane implementations are already available [90, 117].

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 [57, 68, 138]. We show the effectiveness of caching for load balancing and throughput improvement in §3.5.1.

[7]In consistent hashing, multiple virtual nodes are assigned to each physical node for better load balancing [94].

### 3.3.4 High Availability

As mentioned in §5.2, 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 [86], 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 §3.5.1.

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

### 3.3.5 Putting It All Together

Figure 3.6 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 §3.4.3, 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.

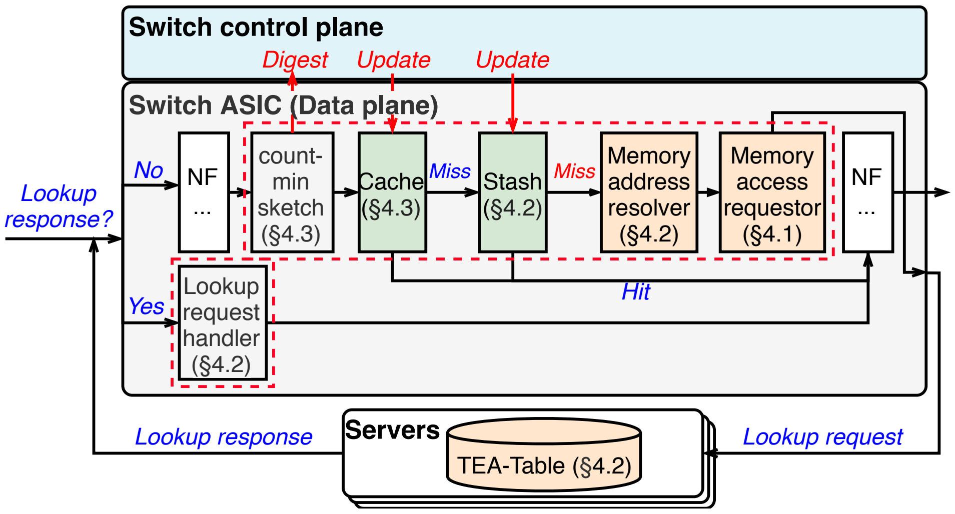


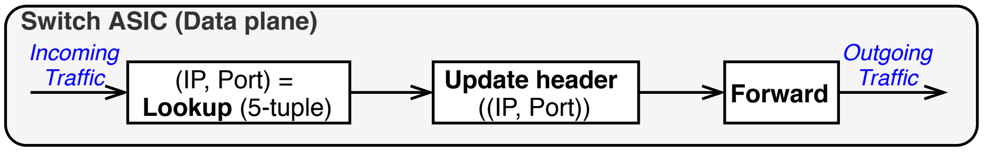
**Figure 3.6:** Summary of key components in TEA. The components form one logical TEA component (dotted-red box) used by an NF pipeline.

**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 §3.5.1, it adds up to around 2 $\mu s$ 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.

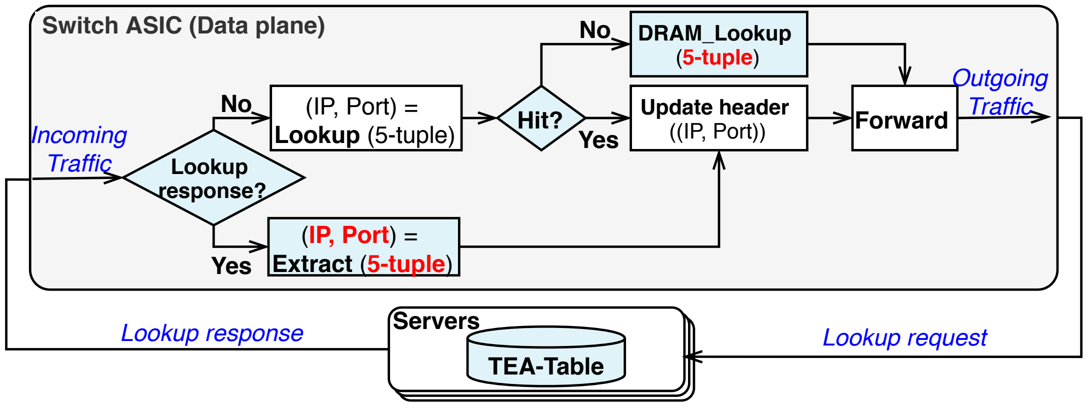
## 3.4 Implementation

### 3.4.1 Data and Control Plane

We implement TEA's data plane in P4 [61] and compile it to Barefoot Tofino ASIC [46] with P4 Studio [42]. 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.



**(a)** Regular NAT data plane.



**(b)** NAT data plane with TEA.

**Figure 3.7:** 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).

We implement the count-min sketch [67] for collecting the statistics and determining popular entries, similar to that of prior work [90, 116]. 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.

### 3.4.2 Programming Network Functions with TEA

Our prototype implements TEA APIs as a library of modularized P4 codes using the concept of control block in P4 [38, §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 3.7 shows how these blocks would be used to implement NAT.

| 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 3.2:** The NFs we developed with TEA. Table sizes are estimated by assuming 10 million entries with IPv6 addresses.

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 3.2 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 **??**.

**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., [2]) based on the details described in prior work [54]. 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.

### 3.4.3 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 (§3.4.2) 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.

## 3.5 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:

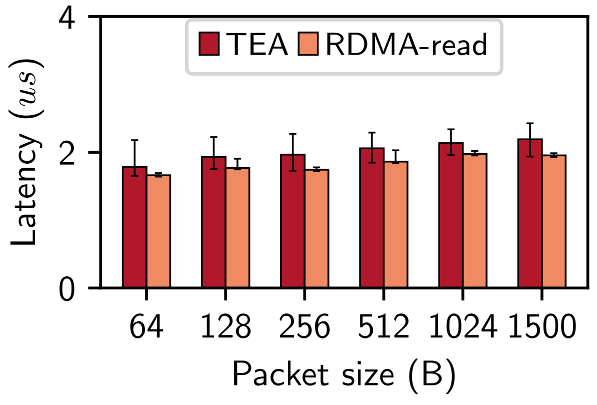
- With a single server, TEA provides a predictable lookup latency (1.8–2.2 $\mu s$) and throughput (7.3–10.9 million lookups per second) for different sizes of packets. With multiple servers, a small cache helps balance loads across servers across different skewness parameters. With the cache, adding servers scales the throughput effectively and 8 servers can perform 138 million lookups per second under a skewed workload. (§3.5.1).
- 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 (§3.5.2).
- 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 (§3.5.2).
- 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% (§3.5.3).

**Experimental setup.** Our testbed consists of a Wedge 100BF-32X 32-ports programmable switch [39] 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.

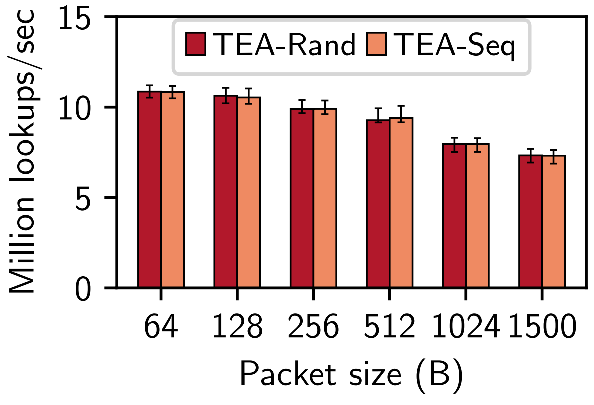
**Traffic workloads.** We use both packet traces collected from a real data center network [20] and synthetically generated ones. The packet sizes vary (64–1500 B) in the real trace. The synthetic traces are based on the observations from several data center measurement studies [57, 68, 138]. We generate packet traces with the flow size distribution in terms of the number of packets per flow that follows Zipf distribution with the skewness parameter ($\alpha$=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 [21] 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.

### 3.5.1 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.



**(a)** Lookup latency



**(b)** Lookup throughput

**Figure 3.8:** Lookup performance of TEA via an RDMA channel with a single server.

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 [31] to measure RDMA 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.

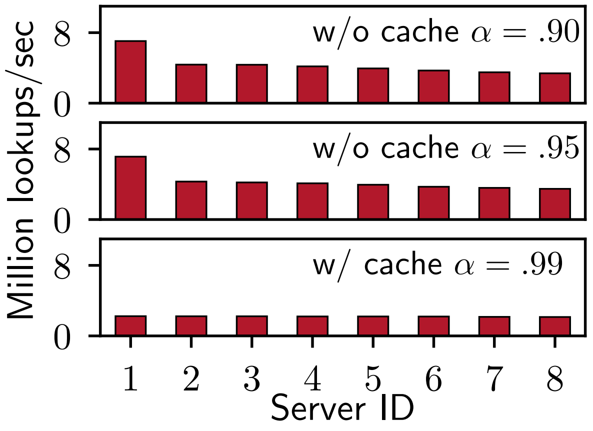
Figure 3.8a shows the median, $10^{th}$ and $90^{th}$ percentile of lookup time. We see that each lookup takes 1.8–2.2 $\mu s$ and the latency grows with the packet size, which is higher than raw RDMA reads (0.1–0.2 $\mu s$). This is mainly because our RDMA read request and response packets are larger 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 3.5b.

Figure 3.8b 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.

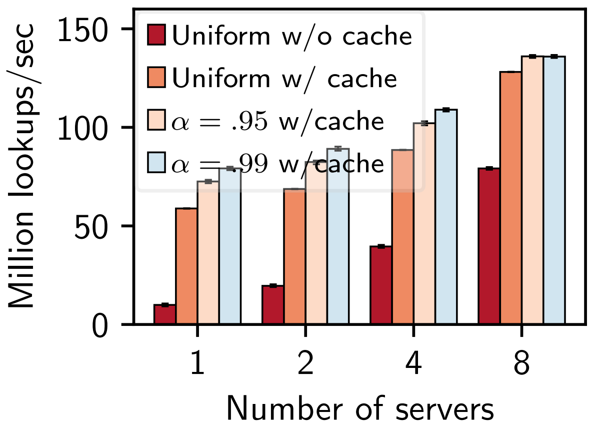
Overall, our evaluation shows TEA's remote DRAM access channel can provide *predictable* performance which is close to the raw RDMA performance.

**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 ($\alpha$) for the flow size distribution and measure the number of lookups served by each server with/without the cache enabled.

Figure 3.9a 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



**(a)** Small cache balances memory access loads.



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

**Figure 3.9:** Scalable lookup throughput of TEA with multiple servers and cache.

with cache, even with the most skewed access pattern ($\alpha$=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 $\alpha$ values. As shown in Figure 3.9b, 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 $\alpha$=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 5.14, 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-

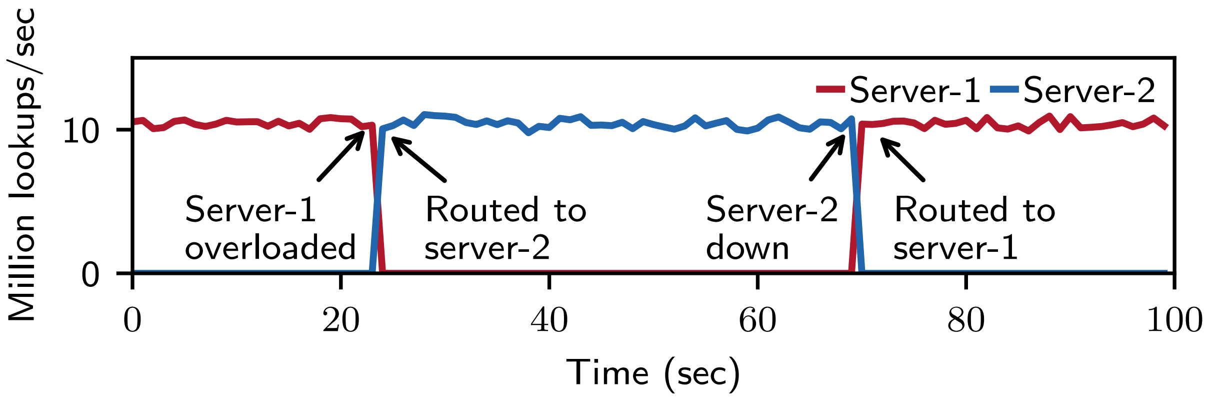


**Figure 3.10:** Lookup throughput changes during failover events.

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 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.

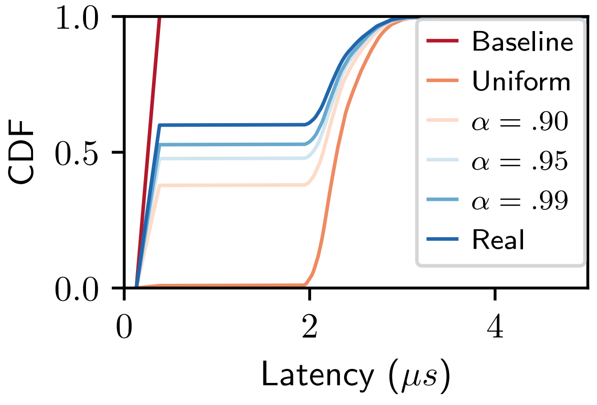
### 3.5.2 NF Performance

**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 3.2 using Click-DPDK [7] which is one of popular ways to implement high-performance NFs. We run them on the server described above.

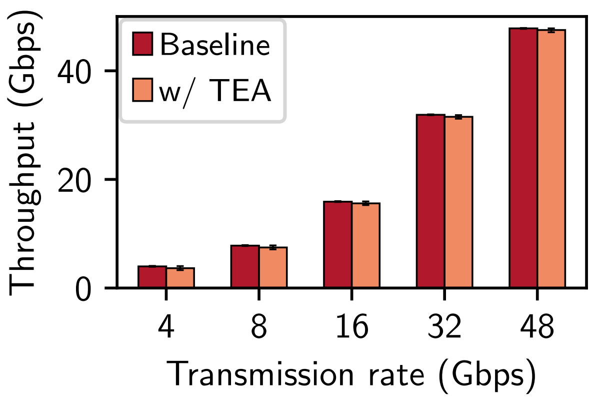
For a fair comparison, we focus on a per-packet processing latency and throughput for 64 B packets with a single server for TEA and server-based NFs. We inject packets using 4 traffic generator nodes (max. traffic rate is ≈138 Mpps). Table **??** summarizes the results with median values for each experiment. 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.

**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 [57] consisting of 64 B packets. Note that since the real traces consist of varying sizes of packets, we make



**(a)** NAT processing latency distribution.



**(b)** NAT throughput for real data center traces.

**Figure 3.11:** Performance of NAT using TEA.

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

**Table 3.3:** Additional switch ASIC resources used by TEA.

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 3.11a 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 $\alpha$=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 3.11b 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.

### 3.5.3 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 §3.3.2, 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 4.3 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.

## 3.6 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 [**?** ], 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 [156] or converts an LPM table into a large exact-match table [**?** ]. 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 [29]. 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.

## 3.7 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 [120] and network monitoring [27, 83, 124] to switches and IPSec gateway, load balancer, and other NFs to FPGA-based smart NICs [74, 106]. 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 [54, 96]. 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 [77, 92, 136, 159]. For example, statelessNF [92] 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 [107], key-value cache [90, 117], and coordination service [91] 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 [70, 93, 121], distributed shared-memory [70], transactional systems [64, 71, 100], and distributed NVM systems [118, 144]. Our work demonstrates a novel use of RDMA, which allows a programmable switch to leverage external DRAM on such servers.

## 3.8 Summary

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. Looking forward, even though our specific focus in this paper was on NFs, we believe that TEA can be a key enabler for many innovative memory-intensive applications running on programmable switches.

# Chapter 4

# Augmented Switch Resource Architecture to Support Concurrent In-Network Applications

Recent advances in programmable switching ASICs [44, 46, 47] have become a key enabler for in-network computing. Programmable switches are now capable of implementing sophisticated network functions, such as NATs, firewalls, and load balancers [27, 97, 120] and accelerating distributed systems [108, 109, 140, 152, 165]. Cloud and cellular service providers have even started deploying them in their data centers [1, 26, 120, 129, 134].

This growing popularity of in-network computing has resulted in two associated trends: the increasing number of in-network applications [85, 99] and the increasing demand on these applications in terms of traffic volume and flows [34, 65]. Since each application requires some amount of switch resources to maintain state (*e.g.,* for per-flow state), the trends imply that the resource requirements will also grow. Unfortunately, current switch resources are extremely limited (*e.g.,* 10s MB of SRAM) and cannot keep up with these ever-increasing demands.

Instead of optimizing applications or adding more resources to switch hardware, we ask a different question in this work: Is there an immediately deployable and future-proof way of enabling multiple applications for demanding workloads? To this end, we explore an alternative *rack-level resource augmentation architecture* that consists of a programmable switch and a few other types of external data plane devices connected to the switch. This leverages recent trends where other programmable data plane devices such as smart NICs [10, 11, 15, 18] (or even software-based data plane running on x86 servers [5, 153]) offer an opportunity to trade-off performance for more resources. Perhaps, more significantly, it also offers a path to more affordably and incrementally scale the effective capacity by adding more external devices.

To effectively realize this vision of rack-level resource augmentation for programmable switches, however, we need a new *operating system* for the architecture. To borrow from Anderson *et al.* [51], we draw a first-principles analogy to three roles that any OS serves: (1) "referee" for managing resources shared between different applications; (2) an "illusionist" to provide an abstraction of physical hardware to simplify application design; and (3) "glue" to provide a set of common services that facilitate the sharing of resources among applications. In our setting, this means that we need new OS abstractions to lower the burden to developers and network operators when multiplexing different in-network applications to manage shared system-wide resources to satisfy per-app or cross-app requirements. While there is some preliminary work that focuses on mapping in-network applications to heterogeneous devices or to augment memory (*e.g.,* [76, 101, 102, 148]), these do not fundamentally provide OS-like abstractions or tackle multiplexed workloads.

To this end, we present ExoPlane,[1] an OS for rack-level switch resource augmentation architectures. Similar to classical OSes for general-purpose computers, ExoPlane provides: (1) abstractions for applications to be written using a "one big switch" model with no or minimal modification to apps (*i.e.,* an illusionist); (2) interfaces to hide the complexity of resources spread on heterogeneous devices (*i.e.,* a glue); and (3) mechanisms to allocate and manage resources across multiple applications to meet per-app and cross-app requirements specified by network operators and developers (*i.e.,* a referee).

While this idea sounds promising, realizing it in practice entails the following high-level challenges:

- Runtime abstraction: Naïvely placing and executing workloads on different devices at runtime can result in high performance overheads due to frequent inter-device communications. It can also overload only a few external devices while the remaining is underutilized.
- State management: Managing application state correctly under dynamic workloads is challenging, especially because each application can involve multiple stateful objects, each of which can co-exist at multiple devices and be updated simulataneously in the data plane at a high rate.
- Resource allocation: There could be multiple per-app and cross-app requirements, which makes it difficult to find an optimal resource allocation that satisfies all the requirements simultaneously.

These challenges are unique to our rack-level resource augmentation architecture-based computing as opposed to server-based computing, especially due to its characteristics including limited compute and memory resources and their capabilities on switching ASICs and the existence of multiple data plane devices where the same application can co-exist and run simulataneously.

We address these challenges with the following ideas:

- To avoid frequent inter-device communications during packet processing, we propose a *run-to-completion operating model* that guarantees that a packet is processed entirely on a single device and implement it resource-efficiently in the data plane.
- To realize our run-to-completion model even under dynamically changing traffic workloads, we design a two-phase state promotion and demotion mechanism that places application states correctly on different devices as workload changes.
- For objects co-existing on multiple devices, we provide two levels of consistency modes for different types of objects. In particular, to synchronize entries in objects that can be updated in the data plane, we propose a periodic combining mechanism that can periodically synchronize the entries correctly by best leveraging capabilities in the device control and data plane.

[1]The name denotes an external (exo-) data plane

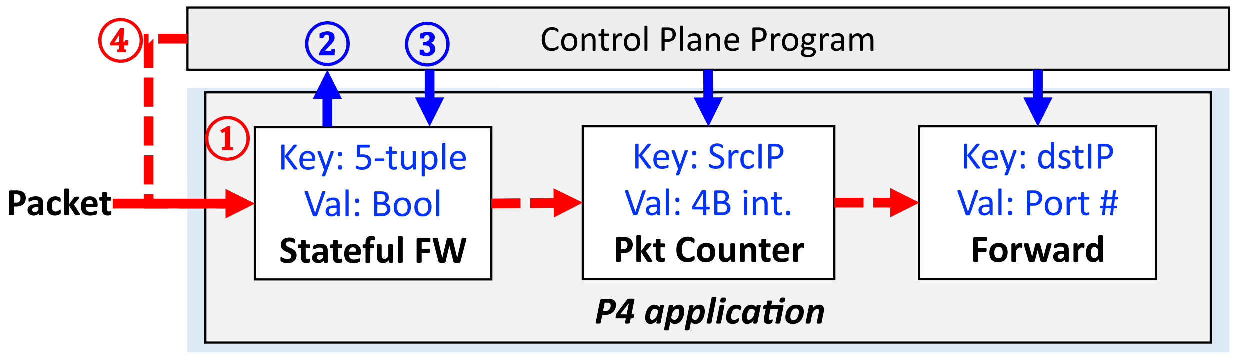


**Figure 4.1:** An abstract P4 application and runtime model. An application consists of multiple stateful objects (white boxes) and the control plane logic (blue arrows).

- To find an optimal resource allocation across multiple apps, we formulate the resource allocation problem as an integer linear program (ILP) that finds an optimal allocation satisfying various constraints provided by developers and operators via our interfaces.

We implement a prototype of the ExoPlane data plane component in P4 and the control plane component in Python and C++. We evaluate it with microbenchmarks as well as application benchmarks using various P4 switch programs in our testbed consisting of an Intel Tofino-based programmable switch and four servers equipped with Netronome Agilio CX smart NICs [11]. Our evaluations show that ...

## 4.1 Background and Motivation

In this section, we provide a primer on in-switch applications and discuss trends in increasing number of applications and per-app requirements that motivate the need for resource augmentation.

### 4.1.1 Primer on Stateful In-Switch Applications

In-network processing has recently flourished, as a natural convergence of network operators' demand for sophisticated network functionality and higher performance and the commercial availability of programmable switches [32, 44, 46]. Programmable switches are used for middlebox functionality [97, 120, 129], monitoring [27, 83], DDoS defense systems [115, 160, 162], and accelerating other networked systems [90, 107, 108, 109, 117, 140, 152, 165].

These applications are *stateful*; *i.e.,* state on the switch determines how to process packets. A typical application program ($p$) contains one or more *stateful objects* ($o_i$), each of which can be represented as a match-action table, register, meter, or counter array in P4 [38].[2] Each object contains not only *state data* in the form of key-value pairs (($K_{o_i}, V_{o_i}$)) but also *actions* for the data. For example, a register object in P4 consists of a data array and actions that access the array. Fig. 4.1 shows an example stateful P4 application ($p$) with three stateful objects ($o_1$–$o_3$)

[2] While our focus of this paper is on P4, other programming languages for programmable switches such as NPL [12] provide similar constructs.

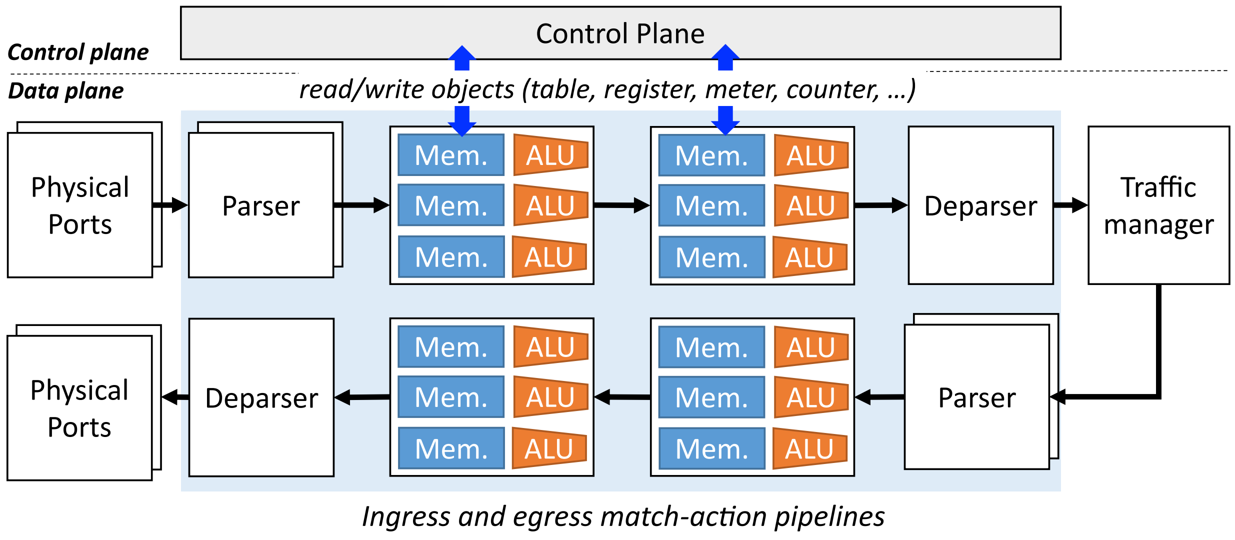


**Figure 4.2:** Conceptual view of programmable switch architecture.

(white boxes). Each stateful object requires some amount of memory (*e.g.,* SRAM and TCAM) for state data and compute resources (*e.g.,* stateful ALUs (SALUs) and hashing units) for actions.

To understand how the application is mapped to hardware, Fig. 4.2 shows a reconfigurable match-action table (RMT) [60]-based programmable switch architecture (*e.g.,* Intel Tofino). In this architecture, packets are processed in a streaming manner in a hardware *pipeline* consisting of multiple match-action stages with memory (denoted as *Mem.*) and compute (denoted as *ALU*) resources. When a target-specific compiler (*e.g.,* Tofino P4 compiler [42]) compiles program codes, it places each object to one or more stages based on the resource requirements and the resource availability, and the placement is fixed at compile-time. Typically, the amount of required memory increases proportional to the size of an object (*e.g.,* the number of table entries or the size of a register array) whereas compute resources are allocated per object basis (*e.g.,* one SALU is allocated to a register array regardless of its size). If the resource demand from the app exceeds the available resources in the switch, the compilation process will fail.

Once the application is successfully loaded to a pipeline, it can process incoming packets using its stateful objects (*e.g.,* dropping packets from an external networks, which do not belong to any established connections). At runtime, the control plane logic can access the objects in the data plane (*e.g.,* inserting a new entry to the stateful firewall (FW) object). Note that in the current switch architecture, inserting and deleting entries from a match-action table can be done only via the control plane while the data plane only can look up an entry from the table. Registers can be read and updated by both the data and control plane.

Fig. 4.1 illustrates the above runtime workflow for our example app. When a packet from an internal network comes in and if a state miss occurs at the stateful FW (①), it reports the packet to the control plane program (②) that inserts new entries for the packet (or flow) (③). Optionally, it sends the packet back to the data plane (④) so that it can be processed with the inserted entries.

**Multiple applications.** In this paper, we focus on serving *multiple* of such stateful applications concurrently. As the number of useful in-network applications increases [85, 99], there is a need

| Applications | States |
|---|---|
| Per-tenant VPN gateway + Packet counter | Ext.-to-int. tunnel mapping and processed packet counter for each tenant. |
| Per-tenant NAT | Per-flow address mapping for each tenant. Per-flow address mapping for each tenant. |
| Per-tenant ACL + Filtered packet counter | Per-flow ACL and dropped packet counter for each tenant. |
| Sketch-based monitor | UnivMon sketch [116] for remaining traffic classes. |

**Table 4.1:** P4 applications deployed in a front-end switch of the data center in our motivating scenario.

for running multiple functionalities simultaneously to maximize the benefit of in-network processing. The typical workflow to serve multiple applications simultaneously on a switch today is to merge them into a single P4 program and compile and load the merged program on the switch. While some multi-pipeline switching chips (*e.g.,* Intel Tofino) can load different P4 programs on different processing pipelines, each pipeline can only load a single (merged) program. That is, if a single application code or data cannot be fit into a single pipeline's resources, compilation will still fail.

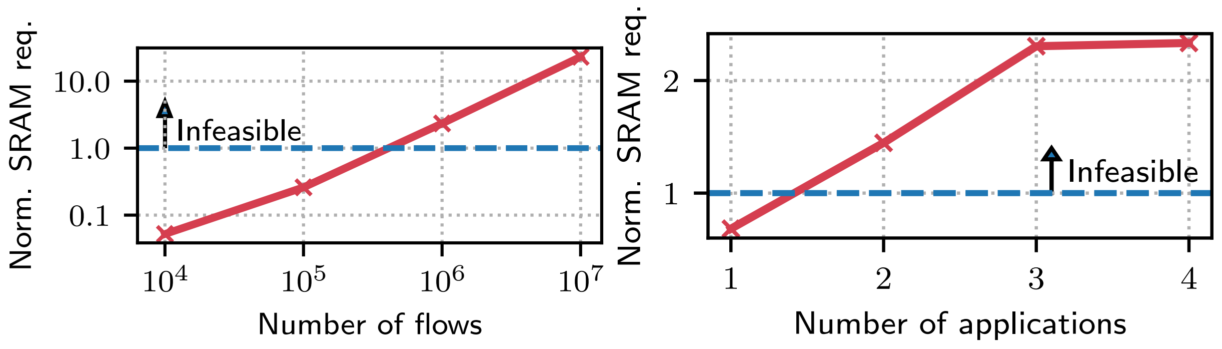
### 4.1.2 Motivation

Unfortunately, however, serving multiple applications on a switch is challenging today, as we describe next.

There are two trends that make it challenging to serve multiple applications concurrently on a switch: (1) The number of applications keeps increasing [85, 99], and (2) per-app workload size in terms of traffic volume and the number of flows keeps growing [34, 65]. Since these trends require more switch resources, the switch cannot keep up with this ever-increasing workloads with its limited on-chip resources (*e.g.,* 10s MB of SRAM).

As a concrete example, suppose the cloud network operator desires to deploy four applications described in Table 4.1 on a switch located at the network frontend where the switch processes traffic arriving at and leaving from the network. Each application maintains per-flow states for each tenant to enable virtual private networks (*VPN gateway*), route traffic from tenants' on-premise networks to VMs running services (*NAT*), or control access to services running on tenants' VMs (*ACL*). Moreover, the *sketch-based monitor* collects statistics for the remaining traffic classes using an UnivMon sketch [116]. We implement these applications in P4 or adopt source codes from the original authors, compose them into a single P4 program using our merger (described in subsection 4.4.4) and compile it using the Tofino P4 compiler.

Unfortunately, however, we find that enabling these applications concurrently in a switch is infeasible with a typical workload in a cloud setting. Fig. 4.3 illustrates the feasibility with a



**(a)** Varying number of flows (both axes are log-scaled).

**(b)** Varying number of applications.

**Figure 4.3:** SRAM requirements (normalized to the total amount of SRAM on a switch) with varying number of applications and workload size. If the requirement exceeds 1, it is an infeasible case.

varying number of applications and workload size. In particular, we use SRAM requirements from each application, normalized to the total amount of SRAM on a switch,[3] which is the bottleneck resource in our scenario. Fig. 4.3a shows how many applications the switch can run with a varying number of flow (*i.e.,* workload size). We see that when each flow need to handle more than a million flows, it becomes infeasible to run all the applications. Also, when we vary the number of applications while fixing the number of flows to 1 million, the switch can support only a single application (Fig. 4.3b).

### 4.1.3 A Case for Rack-Level Resource Augmentation for Programmable Switches

Given these above trends, one can consider several candidate solutions to support multiple applications; *e.g.,* optimizing applications to reduce resource footprints or adding more resources to the switching chip. While these are valid approaches, they have limitations; *e.g.,* applications, even if optimized, may have high resource usage, and extending switching hardware is expensive.

Instead of being constrained in these fundamental ways, we explore a different practical alternative. Specifically, we envision a rack-level architecture consisting of a programmable switch and a few number of other programmable *external* data plane devices connected to the switch could be a potential solution. We call it a *rack-level switch resource augmentation architecture*.

This architecture is well aligned with recent technology trends. First, as the P4 language becomes mature and adopted by switching chip vendors [44, 46], many efforts on enabling P4-based programming for other types of heterogeneous data plane devices, including NPU or FPGA-based smart NICs [11, 17, 88, 157] and software switches on x86 servers [5, 13, 153]. While these devices provide lower packet processing throughput (up to a few 100s of Gbps), compared to hardware switches, they have more resources (*e.g.,* a few GB of DRAM) which are

[3]We use normalized numbers due to NDA.

enough to support demanding workloads.[4] Second, and perhaps more importantly, we can also *extend* the resources needed by simply adding more devices as needed.

Taking these above two factors into account suggests that if we could effectively realize such an architecture, it offers a cost-efficient, incrementally extensible, and potentially "future-proof" way forward to support the growing demands of multiple in-switch applications.

## 4.2 Overview

While the vision of rack-level switch resource augmentation seems promising, to realize it in practice, we need a practical *operating system*. Drawing an analogy from a classical OS for server-based computing that manages resources and provides suitable abstractions for multiple concurrent programs, our OS should ideally provide a *one big switch abstraction* that gives an illusion of large resources to each app. That is, in-switch app developers and network operators can express their programs and requirements at a higher level of abstraction without having to worry about the complexities of managing and multiplexing the resources on heterogeneous devices.

Such an OS must satisfy several natural requirements:

- **Transparency:** It should not require significant modifications on application codes.
- **Flexibility:** Developers and network operators should be able to specify their per-app and cross-app objectives that are considered when allocating resources.
- **Low resource overhead:** It should not consume too much resource, especially scarce switch resources.
- **Performance:** It should not increase per-packet latency and throughput for a common case.
- **Correctness:** Applications should run as if they are running on a single switch with large resources.

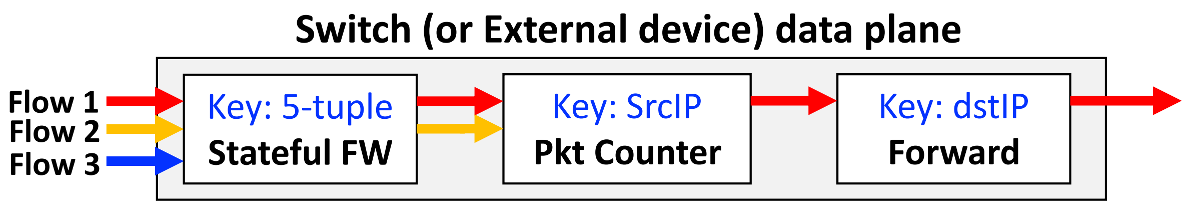
While some preliminary efforts have presented some frameworks for using heterogeneous data plane devices for individual in-switch apps [76, 102, 148], none of them provides an OS with the one big switch abstraction that satisfies the above requirements for multiple applications.

### 4.2.1 Choice of Operating Model

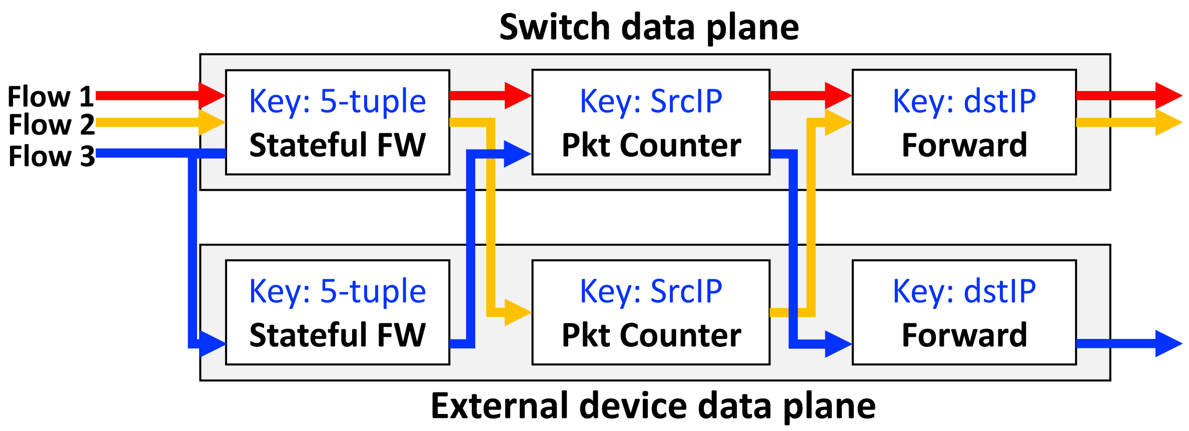
To design an OS, first we need a principled way to choose an appropriate operating model that defines how to effectively multiplex and process workloads. A classical OS multiplexes multiple programs on the limited CPU/memory by choosing when and what processes to swap in/out. Similarly, in our context, the workload is a set of incoming packets mapped to various in-switch applications, and the operating model needs to choose how best to multiplex the processing of these packets across apps and different hardware devices.

**Strawman models.** We consider a design space of options for operating model defined by two key dimensions: (1) Can an application run on multiple devices? and (2) Can an individual packet

[4]For example, Netronome's Agilio CX Smart NICs [11] are equipped with 2 GB of DRAM that is enough for maintaining several million flow states while being able to sustain up to 40 Gbps of packet processing rate.



**(a)** App-pinning operating model: A packet is processed entirely at a single device. However, due to inefficient resource usage, processing capacity is limited.



**(b)** Full-disaggregation operating model: Multiple inter-device communications due to inefficient state placement.

**Figure 4.4:** Strawman runtime operating models. Arrows with different colors indicate trajectories of packets from different flows.

can be processed on multiple devices? Now, given this design space, we pick two representative points to understand key trade-offs and challenges:

- The first option we can consider is an *app-pinning* model where an application is pinned to a single device, and a packet is entirely processed on that device (Fig. 4.4a). In this model, since the packet is processed entirely on a single device without requiring additional logics, there is no processing latency and resource overhead. However, since the app can only run on that particular device, its throughput and available resources are limited.
- A second option is a *full-disaggregation* model where an app can run on multiple devices, and a packet also can be processed on multiple devices (Fig. 4.4b). In this model, since an app can be placed any devices, it could utilize more resources. However, depending on the availability of state, a packet can be routed between the switch and the external device multiple times. Such frequent inter-device routing increases per-packet processing latency and makes it unpredictable. Also, it incurs high resource overhead due to per-object inter-device processing logic to route packets to a particular device and resume processing at that object on the device. It also consumes additional link and device bandwidth.

**Takeaways.** (1) Frequent inter-device routing can increase per-packet latency and make it unpredictable and consume additional bandwidth (2) Maintaining per-object inter-device processing

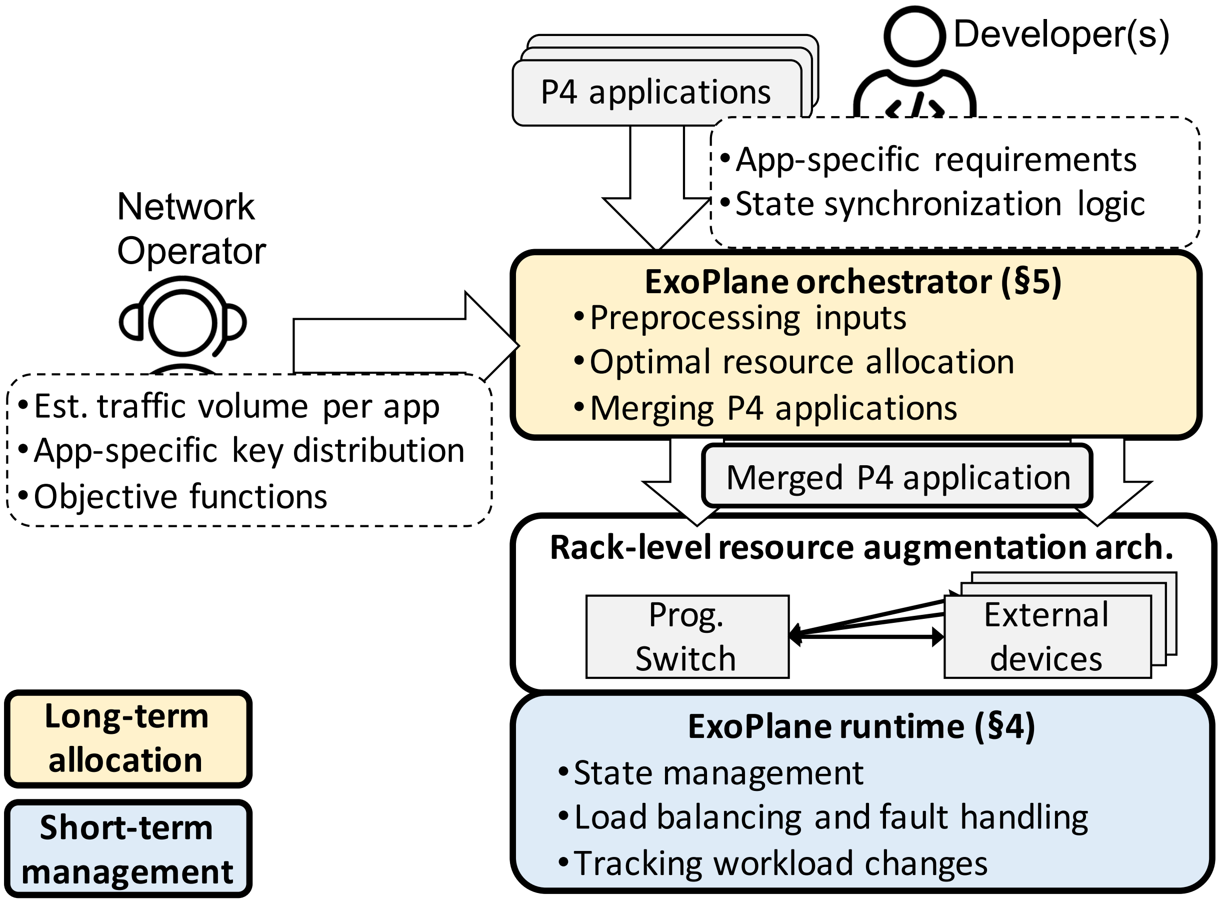

Developer(s)
P4 applications
•App-specific requirements
•State synchronization logic
Network Operator
•Est. traffic volume per app
•App-specific key distribution
•Objective functions
ExoPlane orchestrator (§5)
•Preprocessing inputs
•Optimal resource allocation
•Merging P4 applications
Merged P4 application
Rack-level resource augmentation arch.
Prog. Switch
External devices
ExoPlane runtime (§4)
•State management
•Load balancing and fault handling
•Tracking workload changes
Long-term allocation
Short-term management

**Figure 4.5:** Overview of ExoPlane.

logic incurs high resource overheads, and (3) Running an app on multiple devices allow flexible resource usage.

**Our run-to-completion operating model.** Based on these takeaways, we adopt a *run-to-completion* model that allows a packet to be completely processed at a single device (*i.e.,* no frequent inter-device routing) while an application can run on multiple devices (*i.e.,* flexible resource usage) without per-object inter-device processing logic (*i.e.,* low resource overhead).

This design is based on two key insights. First, we observe from packet traces captured from real networks that flow key distribution is highly skewed, and only a small fraction of popular keys serves the majority of the traffic for an app (*e.g.,* 6% of keys takes more than ≈80% of traffic). Second, by employing a *primary-key based* state management, we do not need to have per-object inter-device processing logic. We define a primary key type of an app as $PK = \cup_i K_{o_i}$ and define a flow as a set of packets with the same primary key. For instance, in the example application, the IP 5-tuple becomes a primary key. Thus, if we could place primary-key based popular flow state entries on the switch, it allows processing the majority of traffic for the app entirely at the switch while the rest of them are processed at the external device. We describe more details in §4.3.

### 4.2.2 Architecture and Challenges

Given the above idea, Fig. 4.5 shows an overview of ExoPlane. It consists of two components: (1) ExoPlane orchestrator that takes inputs from developers and the network operator, allocates resources, on the switch and external devices. (2) ExoPlane runtime that manages states and handles multiple external devices. Realizing this workflow entails four practical challenges:

**C-1. Correctness even under workload changes.** When a new flow arrives or the flow popularity changes, we need to (re)place object entries at the switch. The entry insertion or eviction involves switch control-to-data plane operations (*e.g.,* inserting a table entry) which is slower than the data plane's packet processing speed. We find that this can lead to incorrect packet processing, especially when there are multiple objects in an application. Also, when there is a cross-flow data plane-updatable object (*e.g.,* a per-srcIP packet counter) in the app, multiple copies of an entry can co-exist on multiple devices, each of which can be updated independently and simultaneously. Thus, we need to synchronize them properly. However, we observe that it is infeasible to apply existing shared object management schemes used in server-based network functions [77, 136, 159] due to hardware constraints.

**C-2. Handling multiple devices and their failures.** While one can add more external devices to extend resources or processing capacity, we find that just adding more devices would not be effective due to possible access load imbalance across the external devices. Also, when an external device fails, we need to detect and react to the failure rapidly.

**C-3. Meeting objectives across applications.** Given multiple applications, we have to share resources among them properly while considering per-app and cross-app objective provided by a network operator and developers. However, finding an optimal resource allocation that satisfies all the constraint is not trivial.

**C-4. Tracking workload changes.** To be performant, the ExoPlane runtime should be able to adapt and respond to workload shifts as the popularity changes. In particular, we have to efficiently track the popularity of keys (both "heavy" ones and those cease to become popular). As many prior efforts have shown, implementing such features without consuming switch data plane resources is not trivial [90, 117].

In the following sections, we first describe ExoPlane runtime (§4.3) and then describe ExoPlane orchestrator (§4.4).

## 4.3 ExoPlane Runtime

In this section, we discuss the design of the ExoPlane Runtime that tackles the issues of runtime resource and state management. For clarity of exposition, we start with a few simplifying assumptions—steady state traffic with no workload changes, no cross-flow state, a single external device, no device failure, and a single application. We revisit and relax these assumptions in the following subsections.

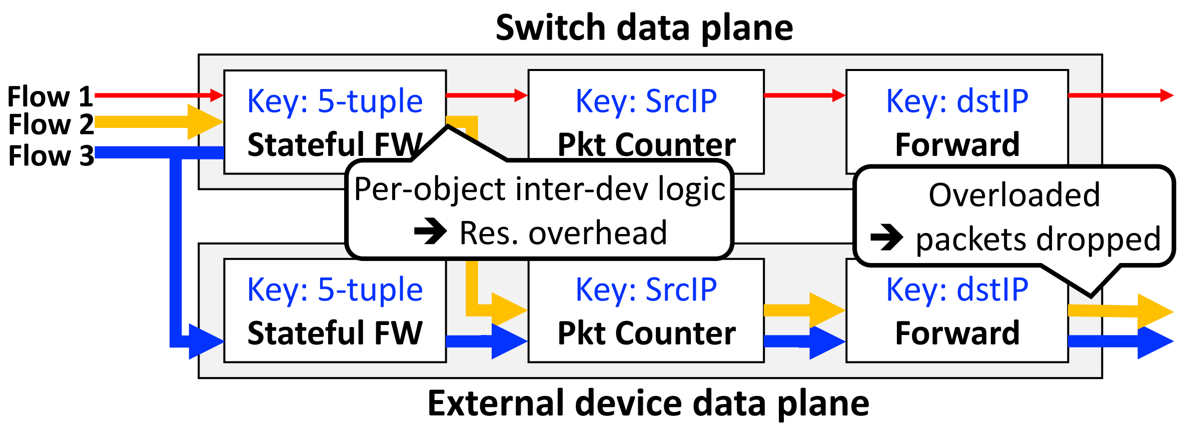


**Figure 4.6:** Inefficient state placement can lead an external device be overloaded, and per-object runtime components incurs high resource overhead. The width of arrows indicates relative traffic volume of each flow.

### 4.3.1 Run-to-Completion Operating Model

Recall from subsection 4.2.1 that we adopt a *run-to-completion* operating model that ensures that each packet is completely processed at a single device (*i.e.,* at most a single round-trip between the switch and an external device). Here, we load an application binary and all state entries on the external device with a subset of entries loaded along with the app on the switch. As mentioned in subsection 4.1.3, each external device has a few GB of DRAM, which is enough to store entire states (requiring up to a few hundred MB for a few million entries). In this model, if there is no entry for an incoming packet at the switch, the packet is routed to the external device where the packet can be processed as all state entries needed to process the packet will be present by design.

However, naïvely implementing this basic model has two potential problems (Fig. 4.6). First, if we do not carefully choose which entries to place on the switch, a high volume of traffic will be routed to the external device (flow 2 and 3 in the figure), and it becomes overloaded, limiting the throughput. Second, since a etnry miss can happen at an arbitrary object (*e.g.,* while looking up an entry of the counter object), per-object inter-device processing logic is needed to handle such cases. As discussed subsection 4.2.1, such additional logic incurs switch data plane resource overheads.

To address these problems, we propose a *primary-key based* state management that enables us to process a majority of traffic for the application at the switch and the remaining at the external device (Fig. 4.7). We define a primary key type ($PK$) of an application as the union of key types of its constituent objects (*i.e.,* $PK = \cup_i K_{o_i}$). A flow then is a set of packets with the same primary key value. For example, in the figure, an IP 5-tuple is the primary key type and packets with the same IP 5-tuple forms a flow.

Having defined the primary key, we can next exploit traffic workload characteristics to enable the switch to serve the majority of traffic for the application. Specifically, we build on the observation that the distribution of flow keys (including the primary key) is highly skewed in typical networking workloads. As an example, we measure the distribution of IP 5-tuple which is the

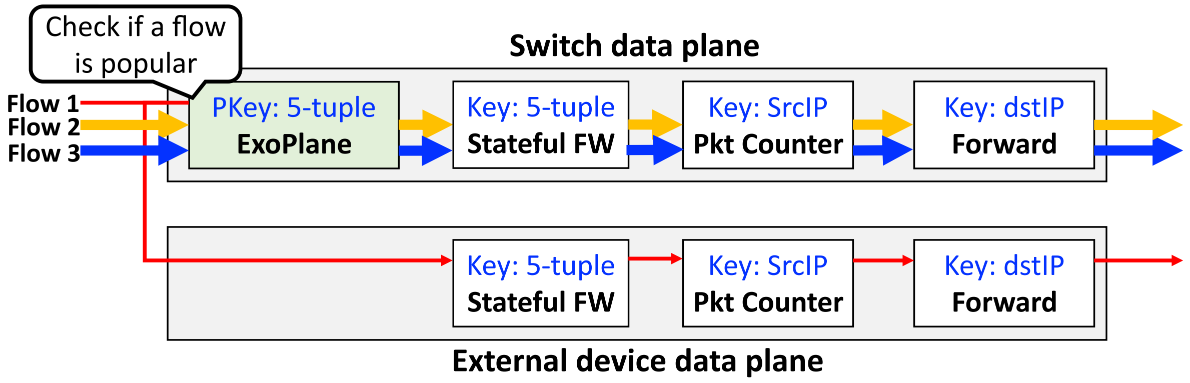
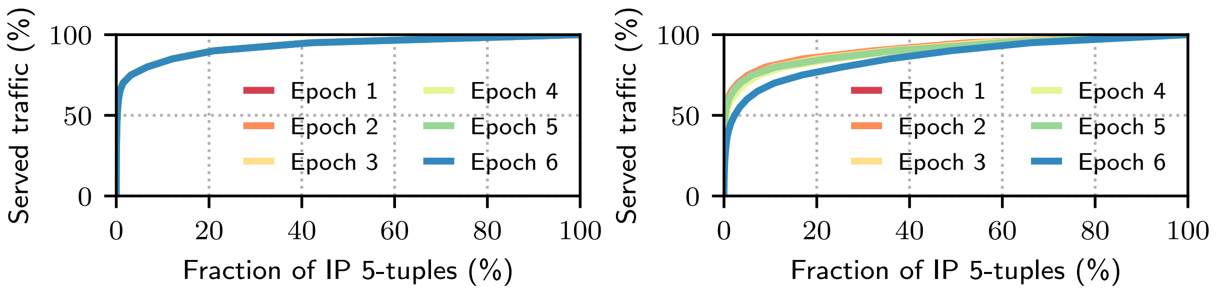


**Figure 4.7:** ExoPlane runtime processes the majority of traffic at the switch and the remaining at the external device in a run-to-completion manner. The green box is a per-app ExoPlane runtime object, and *PKey* indicates a primary key of the application.



**(a)** Internet backbone. **(b)** University data center.

**Figure 4.8:** Skewness in flow key (IP 5-tuple) distribution. For both the I nternet backbone and data center cases, a few popular keys serves the most of the traffic, and this trend holds steadily across measurement epochs.

primary key of our example app, by analyzing packet traces collected from an Internet backbone [4] and a university data center [57]. Fig. 4.8 shows the results. For both cases, we see that a small fraction of the keys contribute to the majority of the traffic; ≈6% of keys in the backbone and ≈10% of keys in the data center takes more than ≈80% of traffic. The skew persists across measurement epochs (5 mins and 1 mins for the backbone and data center, respectively). We also confirm the skew exists for other coarse-grained keys such as the source IP. This suggests that we can serve the majority of the traffic at the switch by placing a few state entries with popular primary keys (*e.g.,* 516 entries for 80% in the data center trace).

Based on this, we employ a per-app *ExoPlane runtime object* (the green box in Fig. 4.7 and denoted as $o_{ExoPlane}$) at the switch, which maintains a list of popular primary keys and checks whether the key of an incoming packet exists in the list when it arrives at the switch. If the key exists (*i.e.,* the packet is from a popular flow), the packet is processed entirely at the switch. Otherwise, it is routed and processed at the external device. This way, we do not need to have per-object inter-device processing logic, which minimizes the resource overhead.

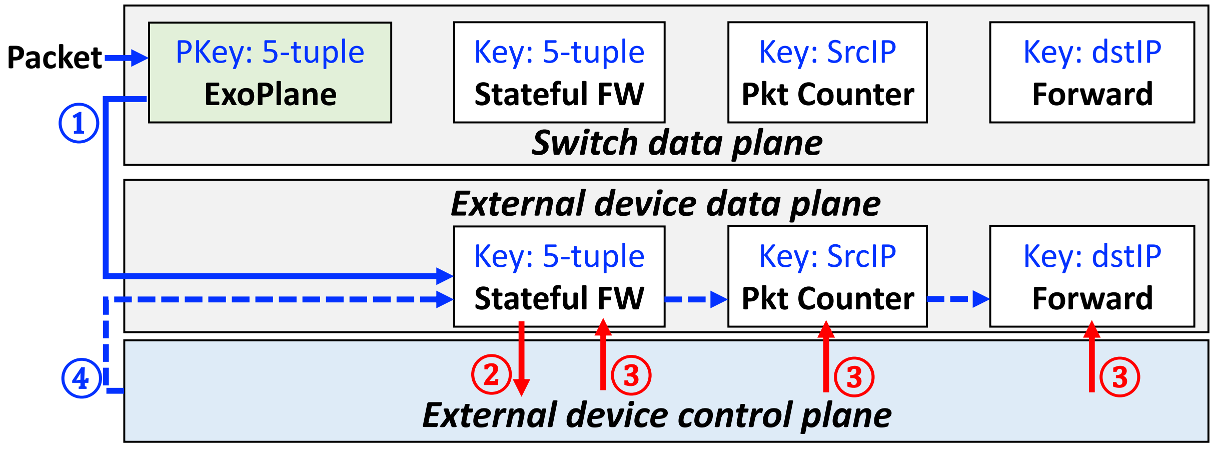


**Figure 4.9:** Steps in the basic workflow for inserting new flow entries in ExoPlane.

In sum, our run-to-completion operating model and its data plane implementation provides the following property:

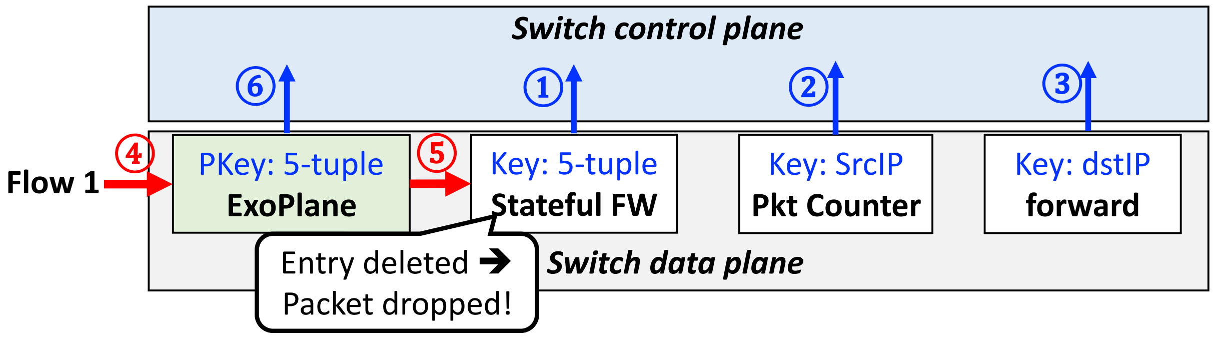
**Invariant 1 (Run-to-completion).** *For each application, if the ExoPlane runtime object ($o_{ExoPlane}$) has the packet's primary key ($PK(pkt)$), the constituent objects ($o_i$) must have entries ($K_{o_i}(pkt)$) for the packet.*

$$\forall pkt : PK(pkt) \in o_{ExoPlane} \implies \forall i : K_{o_i}(pkt) \in o_i.$$

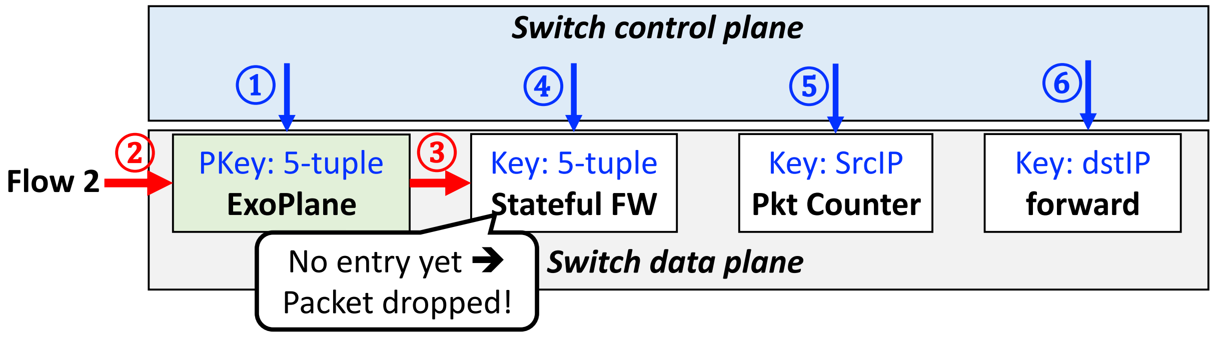
## 4.3.2 Handling Workload Changes

While the above idea works correctly under a simplified assumption of steady state, it does not handle workload changes. Specifically, there are two cases where ExoPlane runtime needs to update objects on devices due to workload changes: (1) new flows arrive and (2) flow popularity changes. In this section, we first describe how ExoPlane runtime handles new flows and then discuss challenges in updating objects to promote popular flow states.

**Handling new flows.** Fig. 4.9 illustrates a new flow arriving in ExoPlane runtime. When a packet belonging to the new flow arrives at the switch, and if a miss occurs in a ExoPlane runtime object (**Step** ①), it routes the packet to the external device. Note that there are two possible cases where the miss happens: (1) the first packet of the new flow arrives or (2) a packet of an old flow arrives, but the flow state does not exist at the switch. Since the ExoPlane runtime object cannot differentiate between the two at the moment, it always routes the packet to the external device for both cases. Once the packet arrives at the external device, since this is a new flow, the packet must first be processed by the application's control logic for handling new flow arrivals. In this example, the stateful FW table reports the packet to the control plane (**Step** ②), and the control plane program inserts entries for the flow to three objects (**Step** ③). Depending on the application logic, the packet can be sent back to the data plane and processed with the new entries (**Step** ④). Subsequent packets in the flow will be processed at the external device.



**(a)** Incorrect state eviction: application's state has been removed while there is a packet being processed.



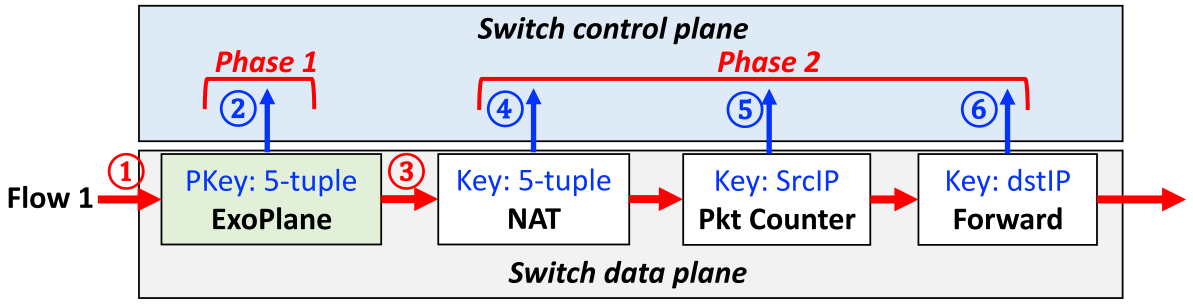
**(b)** Incorrect state insertion: application's state is not available while a packet has started being processed.

**Figure 4.10:** Incorrect state insertion and eviction.

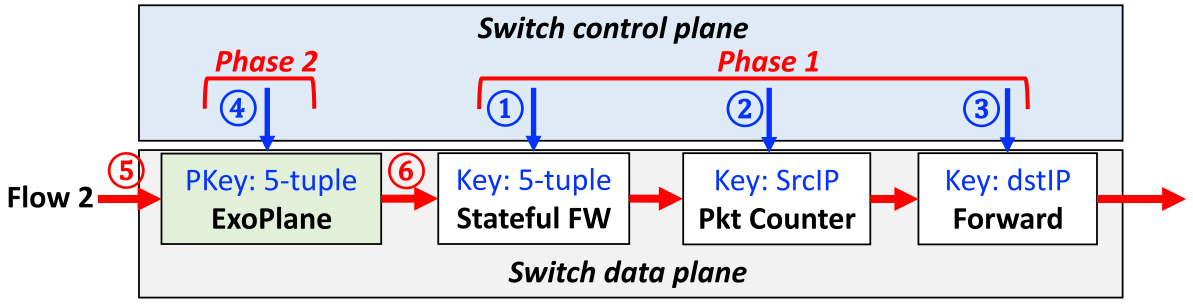
**Promoting popular flows.** So far, we have assumed popular flow states are located at the switch. In practice, however, the popularity of flows can change at runtime. Thus, we need to promote and demote flow states as the popularity changes.

Suppose flow keys that become popular (*i.e.,* their entries are currently not on the switch) and that become unpopular (*i.e.,* their entries are currently on the switch) are given (we describe our implementation of this in subsection 4.5.2). When promoting a new popular flow (*i.e.,* installing state entries for the flow to the switch), there are two possible cases: (1) there is spare space in the ExoPlane runtime object and application's other objects for new entries *vs.* (2) there is no room in the objects. For the first case, we can insert new entries to the objects whereas for the second case, we first need to evict entries for an unpopular flow to make a room, and then insert new entries. At first glance, it seems trivial to insert and/or evict entries for both cases, we find that performing them *correctly* is challenging.

Fig. 4.10 illustrates why a naïve update mechanism can violate this property. Suppose that flow 2 becomes popular while flow 1 becomes unpopular, and there is no room for inserting new entries. Thus, the switch control plane tries to replace the entries for flow 1 with the flow 2's. It first evicts entries for flow 1 from application objects (FW, Counter, and Forward) as well as the ExoPlane runtime object (blue arrows in Fig. 4.10a). However, in the current switch architecture, since a set of eviction operations (blue arrows) cannot be executed atomically, depending on the order of executions, there could be a case where application's state entries have been removed



**(a)** Correct two-phase state eviction.



**(b)** Correct two-phase state insertion.

**Figure 4.11:** Correct two-phase state eviction and insertion.

already while there are packets being processed in the data plane (⑤), violating the correctness property.

Even if the eviction is correct, the insertion process can be incorrect. In Fig. 4.10b, the switch control plane tries to insert entries for flow 2 (blue arrows). In the meantime, a packet in flow 2 arrives and looks up the ExoPlane runtime object to check if it can be processed at the switch (②). Since the entry exists, the packet must be processed completely at the switch. However, since entries in other objects are not available yet, the packet cannot be processed further and gets dropped (③).

**Our approach: Two-phase state update.** To address the issues, we propose a *two-phase state update* mechanism, inspired by classical two-phase update or commit protocols [58, 137]. Fig. 4.11 illustrates how it guarantees the correctness when evicting and inserting state entries for a flow.

As illustrated in Fig. 4.11a, when evicting entries for flow 1, in the first phase, the switch control plane evicts an entry from the ExoPlane runtime object. Since there can be some packets being processed in the switch data plane, it waits for a certain time period ($T_{flush}$) to flush out the packets. And then in the second phase, it evicts entries from the application's objects. This mechanism ensures that all packets that arrive at the switch before the entry of the ExoPlane runtime object has been evicted are correctly processed in the switch. Note that when it evicts entries from the application's objects, it ensures that entries for other non-victim flows will remain.

The insertion works similarly (Fig. 4.11b). To insert entries for flow 2, in the first phase, the switch control plane inserts entries to the application's objects, and then in the second phase, it inserts an entry to the ExoPlane runtime object.

To summarize, our two-phase update mechanism guarantees the invariant for run-to-completion (subsection 4.3.1) by ensuring that during state promotion or demotion, all necessary state entries for a packet exists at the switch when there is an entry for the packet in the ExoPlane runtime object.

### 4.3.3 Synchronizing Shared Stateful Objects

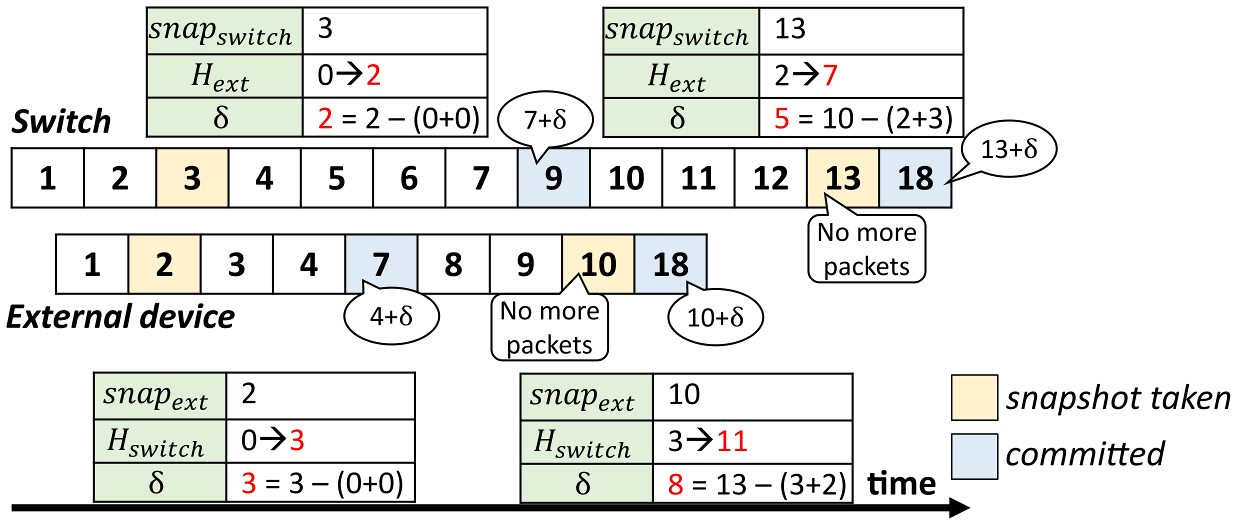
The previous discussion focused on the simple case where entries in each stateful object is local, and there are no cross-flow objects that can be updated at runtime. That is, there was no need for objects on external devices and the switch to be synchronized. However, in practice, some objects may require synchronization, epspecailly when some of their entries co-exist on both devices; *e.g.,* the per-srcIP packet counter in our example application, which can be shared by multiple flows defined by an IP 5-tuple. In this section, we relax that assumption and extend the basic protocol to handle such shared objects.

**Consistency mode.** To provide applications with one big switch abstraction (*i.e.,* an illusion of a single copy of entries in stateful objects), ExoPlane should make the copies on different devices consistent. In P4 programs, there can be two types of stateful objects: (1) *control plane-updatable object* can be updated *only* from the control plane, such as a match-action table and (2) *data plane-updatable object* can be updated from the data plane, such as a register.

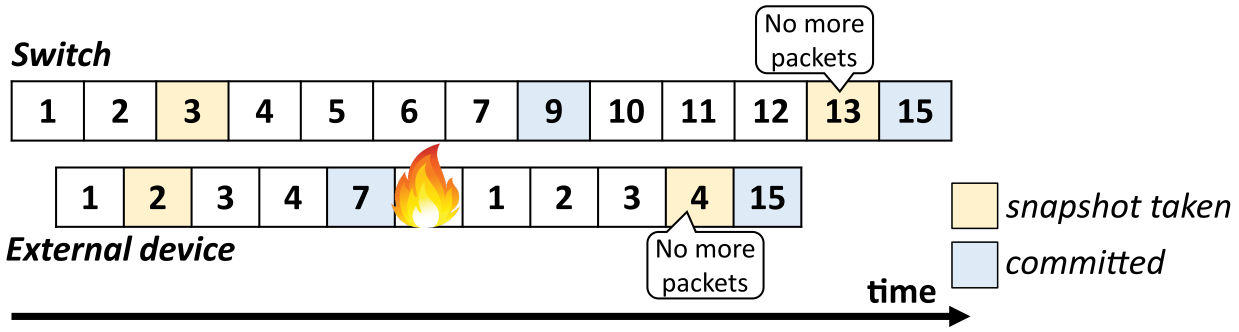
Since each type has a different degree of consistency requirement, ExoPlane provides two levels of consistency for shared objects. Control plane-updatable objects are rarely updated (*e.g.,* a stateful firewall table entry is inserted only for the first packet of each flow generated from an internal network) and an exact value is critical for correct behavior (*e.g.,* allowing packets for an established TCP connection). Thus, for this type, we provide *strong-consistency* mode. In contrast, data plane-updatable objects can be updated more frequently (*e.g.,* per-srcIP packet counter is updated for every packet) and typically do not require strong consistency since they maintain approximate or statistical information (*e.g.,* packet counters and sketches). Thus, for data plane-updatable objects, we provide a *bounded-inconsistency* mode that provides consistency within a configurable time bound $T_\epsilon$ (*e.g.,* 1 second in our prototype).

Supporting strong consistency for control plane-updatable objects is straightforward; when the external device's control plane receives a request for updating (or inserting) an entry to an object (Step ② in Fig. 4.9) with a key (*e.g.,* a srcIP), it updates (or inserts) all entries corresponding to the key co-existing at the external device and the switch. However, we find that realizing the bounded-inconsistency mode is challenging as we describe next.

**Strawman approach for bounded-inconsistency mode.** Consider the per-srcIP packet counter implemented using an array of registers in our example. Suppose that for a given srcIP, there are two copies placed on both the switch and the external device, meaning that the copies can be updated simulataneously. To achieve bounded-inconsistency, the ExoPlane runtime needs to periodically *merge* values of the copies. While one might think of adopting existing mechanisms for managing shared objects in server-based network functions (*e.g.,* [77, 136, 159]), we observe



**(a)** ExoPlane synchronization protocol synchronizes two copies of the state eventually and correctly.



**(b)** Two copies are synchronized correctly even when the external device is recovered from a failure.

**Figure 4.12:** Synchronizing states between the switch and an external device using the ExoPlane state synchronization protocol.

that they are impractical in our context since they assume to have an ability to buffer incoming packets while combining values of the copies, which is expensive and even infeasible in the switch and external device.

**Our approach.** Alternatively, we devise a state synchronization protocol that achieves bounded-inconsistency, which does not require packet buffering by best leveraging capabilities of both the switch and external device's control and data plane. In particular, we utilize the device control plane's DRAM to maintain the history of periodic synchronizations while executing the merge operation in the data plane to avoid an incorrect merging.

Fig. 4.12a illustrates how it works for one entry in our packet counter example. The control plane of each device maintains per-entry metadata including the current snapshot ($Snap$) and a history ($H$) of an entry value on the other side (*i.e.,* the switch tracks the history of the external device and vice versa). For every $T_\epsilon$ second, the switch control plane initiates the synchronization by sending its current snapshot ($Snap$) and the history ($H$), and the external device's control plane replies it with its snapshot and history. In the figure, the switch sends $<Snap$=3, $H$=0$>$

to the external device, and the external device sends $<Snap$=2, $H$=0$>$ back. Then, each side computes the changes that have been made at the other side ($\delta$) after the previous synchronization by subtracting two history values from the received snapshot value. This step prevents a potential under or double-counting issue. Lastly, the control plane of both devices injects a special packet containing $\delta$ to the data plane to combine the changes to the current state value.

Note that our protocol synchronizes the copies of states correctly even when the external device fails and gets recovered. This is because the switch maintains the progress that the external device had made before the failure ($H$) and provides this information to the recovered device so that it can resume the synchronization from the state when it failed (Fig. 4.12b).

While we use the packet counter as an example, our protocol supports other objects that can be expressed in a key-value structure. It provides developers with an interface to specify an object-specific *merge operator* expressed by an addition ($\circ^+$) operator that combines two values and an optional subtraction ($\circ^-$) operator that subtracts one value from the other, which are used by the protocol to compute $\delta$ and commit the update. For example, it can support a Bloom filter [59] which can be expressed as (Key: an integer, Value: $\{0, 1\}$) pairs with the binary OR as $\circ^+$ (no subtraction operator is needed in this case). We describe the pseudo-code of our protocol in **??** and evaluate resource overheads in §5.6.

### 4.3.4 Scaling with Multiple Devices and Handling Failures

So far we have assumed that there is a single external device that does not fail. However, in practice, an architecture with the single device has two potential problems. First, the device may not provide enough processing capacity or resources to serve given application workloads. Second, even if resources are enough, when the device fails, it loses the application state, affecting correctness or performance of the application. To deal with these issues, ExoPlane should support *multiple* external devices. In this section, we describe how ExoPlane effectively utilizes multiple devices for both scenarios.

**Supporting multiple external devices.** When there are multiple devices, ExoPlane shards entries in objects across the devices based on the primary key. So, when an entry miss occurs at the switch's ExoPlane runtime object, it routes a packet based on the primary key to a specific external device that has state for the key. However, the skewness in the primary key space (subsection 4.3.1) could result in load imbalance across the devices (*i.e.,* a subset of devices can be overloaded).

Fortuitously, it turns out that the small fraction of popular entries we already have at the switch is helpful for load balancing. The theoretical result on the effect of a small cache for load balancing in storage systems [73] shows that by caching at least $O(N \log N)$ popular entries where $N$ is the number of backend servers (in our context, external devices), it guarantees uniform load balancing across the servers regardless of the skewness. Thus, by ensuring that at least $O(N \log N)$ popular primary keys are placed at the switch, it provides the cache effect for load balancing.

**Handling external device failures.** Application state loss due to failures can affect the performance or behavior of applications [103]. Specifically, we consider the failure model where an external device (or its hosting machine) fails or a network link between the switch and the

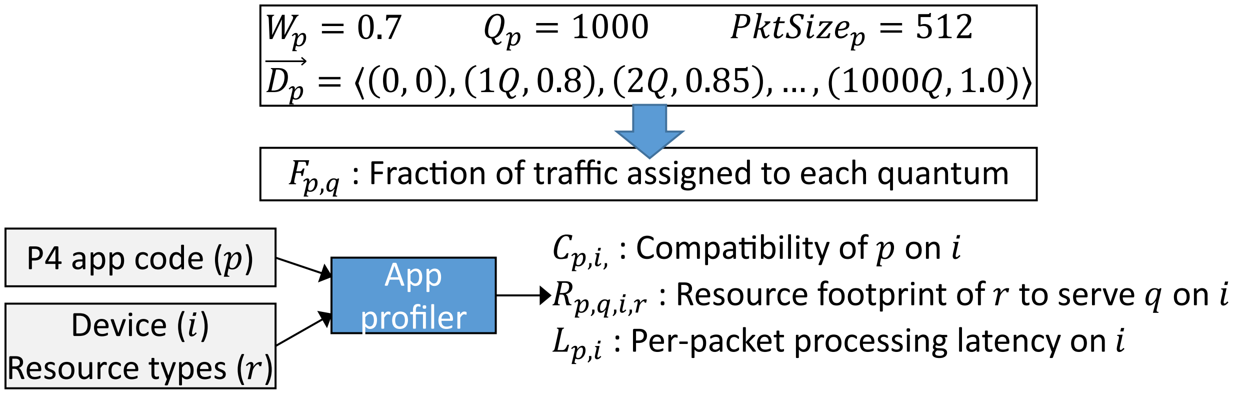

$W_p = 0.7$ $Q_p = 1000$ $PktSize_p = 512$
$\overrightarrow{D_p} = \langle(0,0),(1Q,0.8),(2Q,0.85),\ldots,(1000Q,1.0)\rangle$
$F_{p,q}$ : Fraction of traffic assigned to each quantum
P4 app code ($p$)
Device ($i$)
Resource types ($r$)
App profiler
$C_{p,i,}$ : Compatibility of $p$ on $i$
$R_{p,q,i,r}$ : Resource footprint of $r$ to serve $q$ on $i$
$L_{p,i}$ : Per-packet processing latency on $i$

**Figure 4.13:** Example inputs for an application $p$.

external device fails. To deal with state loss due to such failures, the ExoPlane runtime replicates each flow state to at least one additional external device when initiating entries for the flow, and when the primary device fails, it falls back to a replica. It achieves this by managing the logical to physical external device ID mapping at the switch, where the primary and replica devices share the same logical ID. However, even if there is a replica, if the runtime cannot detect failures and route packets to a replica quickly, the application performance can be degraded (*e.g.,* due to packet drops).

To enable rapid failure detection and reaction, we repurpose the packet generation engine of the switch ASIC. The engine is typically configured to inject packets into a switch pipeline when a certain event occurs mainly for diagnosis purposes. Fortunately, we can configure the engine to generate a packet when ports go down. By processing the generated packet, ExoPlane updates the external device ID table in the data plane. Using this, the ExoPlane runtime decides an egress port for routing the packet to an alive external device.

## 4.4 ExoPlane Orchestrator

Having discussed the operating model for a single app, next we tackle the issue of sharing resources across multiple apps to meet the performance objectives given by developers and the network operator. To this end, we design a ExoPlane orchestrator consisting of the resource allocator and the application merger. The resource allocator takes inputs from the developers and the network operator, finds an optimal resource allocation, and the application merger generates a merged P4 program code based on the optimal allocation decision.

### 4.4.1 Inputs

Before describing how to formulate the problem, we first define inputs from the developers and the network operator. Fig. 4.13 illustrates example inputs for an application.

**Inputs from the developers.** Developers provide a set of application programs ($p$), each of which consists of a set of stateful objects. For each object, developers specify required size (*e.g.,* the number of entries in a table or register object). Optionally, they can also provide a per-app affinity to the switch as one of three values: high, medium, and low. If the affinity of an app is set to high, it means that they want the app to process the entire traffic at the switch. If it is set to low, the entire traffic for the app must be processed at external devices. Otherwise (*i.e.,* medium), our resource allocator finds an optimal affinity for the app.

**Inputs from the network operator.** The network operator provides cross-app and per-app traffic information, which includes a fraction of traffic served by each app out of the entire traffic arriving at the switch ($W_p$) and the cumulative traffic distribution ($D_p$) over the primary key space quantized by $Q_p$. Based on $D_p$, we compute the estimated fraction of traffic served by each quantum $q$ ($F_{p,q}$). The network operator also provides resource type ($r$) information of device ($i$). For example, we consider SRAM, TCAM, hash units, and SALUs for a Tofino-based switch and LUTs, memory blocks, and DSPs for FPGA-based NICs.[5]

### 4.4.2 Application Profiler

Based on the inputs from the developers and the network operator, our application profiler generates per-app profiles consisting of a compatibility matrix and resource footprints for each device type.

**Compatibility with devices.** We observe that P4 compiler backend for external devices may not support certain switch hardware features (*e.g.,* packet recirculation) or P4 objects (*e.g.,* registers) used by applications. Because of this, if an application uses a feature that is not supported by an external device, the device cannot run the application. To consider the compatibility of the app on devices, our profiler generates a compatibility matrix ($C_{p,i}$) that indicates whether $p$ can be run on device $i$ based on a set of features supported by $i$ and a set of features used by $p$. The first set can be typically obtained from device vendor's compiler manual. For the second set, the profiler analyzes the application code to extract features used by the app. In our prototype, we consider a

**Resource footprint.** The profiler estimates resource footprints of $r$ for $p$ serving $q$ on $i$ denoted as $R_{p,q,i,r}$. Since (blackbox) device-specific compilers (*e.g.,* Tofino compiler) determine the resource usage, our preprocessor compiles $p$ to obtain the usage information. For each $q$, it updates the size of each object specified in an application code and compiles it using device-specific compilers. And then it extracts the resource usage from compiler outputs. If the compilation fails due to insufficient resources, it sets the resource usages to infinite. We use constants $Cap_{i,r}$ to represent the total amount of $r$ available on $i$.

**Per-packet processing latency.** The profiler also estimates a per-packet processing latency of $p$ on $i$ denoted as $L_{p,i}$. Specifically, since every packet traverses the switch, it instruments the switch to record two timestamps on a custom packet header field when a packet enters and leaves the rack. And then it injects $PktSize_p$-sized packets to the rack and estimates the processing latency based on the timestamps in returned packets.

[5]The network operator can easily extend this to other resource types.

### 4.4.3 Optimal Resource Allocation

Given these inputs, next we discuss how we formulate the problem of finding an optimal resource allocation satisfying multiple per-app and cross-app requirements as an integer linear program (ILP).

**Assumptions.** In our formulation, we assume that the resource usage of multiple apps can be estimated by accumulating the resource usage of each app.

**Decision variables.** We use binary decision variables $d_{p,q,i}$ to indicate whether $q$ for $p$ is assigned to $i$.

**Constraints.** There are three types of constraints imposed by: (1) the assignment of $q$, (2) the compatibility of $p$ on $i$, (3) the amount of available resources, and (4) the processing latency of $p$ on $i$.

$$\forall p, q : \sum_i d_{p,q,i} = 1 \tag{4.1}$$

$$\forall p, q, i : d_{p,q,i} \leq C_{p,i} \tag{4.2}$$

$$\forall i, r : \sum_p \sum_q d_{p,q,i} \times R_{p,q,i,r} \leq Cap_{i,r} \tag{4.3}$$

$$\forall p : lat_p = \sum_q \sum_i d_{p,q,i} \times F_{p,q} \times L_{p,i} \tag{4.4}$$

First, $q$ must be assigned to a unique $i$ (Equation 4.1). Second, $q$ can be assigned to $i$ if and only if $p$ is compatible with $i$ (Equation 4.2). Third, the amount of $r$ consumed by $q$ on $i$ must be be less than or equal to the total amount of $r$ on $i$ (Equation 4.3). Last, the expected latency of $p$ is the sum of per-packet processing latency of $p$ on $i$ weighted by $F_{p,q}$ (Equation 4.4).

**Objective.** The network operator provides an objective to share resources across multiple application fairly. One possible fairness metric would be minimizing the weighted sum of the expected processing latency of each application:

$$\text{Minimize} \sum_p W_p \times lat_p \tag{4.5}$$

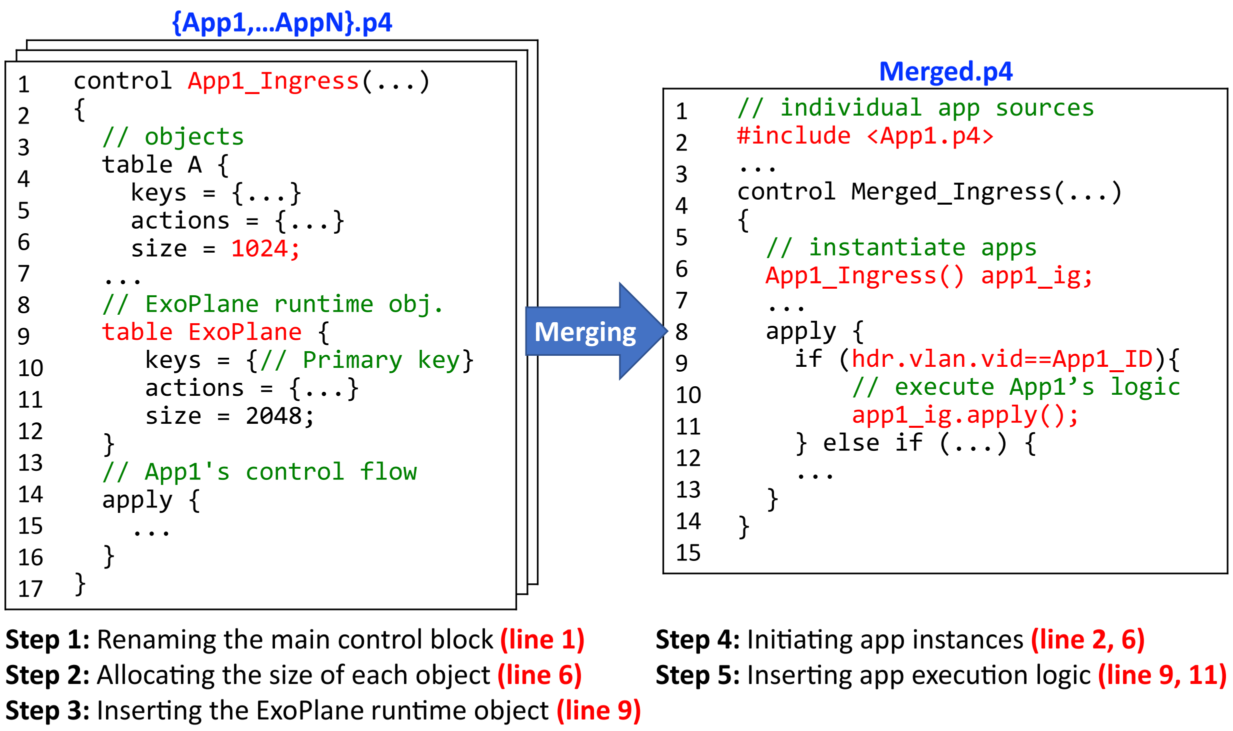
Other commonly used fairness metrics such as maximizing the minimum expected throughput can be used as well.

By solving the ILP, ExoPlane resource allocator finds an optimal assignment of $q$ to $i$ for $p$, and the size of each object and ExoPlane runtime object for $p$ accordingly, which are used as input for the application merger, as we describe next.

Our resource allocation needs to be run when a set of applications or traffic distribution for applications is changed. We assume that such changes rarely happen (*e.g.,* for daily or weekly basis).

### 4.4.4 Application Merger

Given a set of P4 programs and the optimal resource allocation decision, our application merger combines the programs into a single P4 program, following the programming model to support multiple applications, described in subsection 4.1.2.



```
1   control App1_Ingress(...)
2   {
3     // objects
4     table A {
5       keys = {...}
6       actions = {...}
7       size = 1024;
8     ...
9     // ExoPlane runtime obj.
10    table ExoPlane {
11       keys = {// Primary key}
12       actions = {...}
13       size = 2048;
14    }
15    // App1's control flow
16    apply {
17      ...
    }
  }
```

```
1   // individual app sources
2   #include <App1.p4>
3   ...
4   control Merged_Ingress(...)
5   {
6     // instantiate apps
7     App1_Ingress() app1_ig;
8     ...
9     apply {
10      if (hdr.vlan.vid==App1_ID){
11          // execute App1's logic
12          app1_ig.apply();
13      } else if (...) {
14        ...
15      }
    }
```

**Step 1:** Renaming the main control block **(line 1)**
**Step 2:** Allocating the size of each object **(line 6)**
**Step 3:** Inserting the ExoPlane runtime object **(line 9)**
**Step 4:** Initiating app instances **(line 2, 6)**
**Step 5:** Inserting app execution logic **(line 9, 11)**

**Figure 4.14:** Merging multiple P4 programs into a single program.

Merging multiple applications can be complicated, especially when there are dependencies in packet parsers and match-action logics between applications. Currently, there is no P4 program merger that deals with such complexities. Instead of developing a merger that can handle general cases, for simplicity, we consider an application model where each application handles a non-overlapping subset of traffic, which we call a *traffic class*, arriving at the switch so that each packet is processed by only a single application, removing dependencies between applications. To enable this application model, we assume a deployment model where a gateway router (or any other devices) in the network assigns a VLAN tag that encodes an app ID to each packet so that it can be classified at the switch based on the tag.

Our merger supports programs written in P4-16 [38], the latest version of the P4 language. Fig. 4.14 illustrates how the merger works. First, for each application, the merger renames the main control block [38, §13] of each application to avoid naming conflicts between apps. Second, it specifies the size of each object (*e.g.*, the number of entries in a table) based on the decision made by our resource allocator. Third, it inserts an ExoPlane runtime object. Last, in a merged P4 code, it instantiates each application instance and inserts execution logic for each app based on app IDs. The merged P4 code is compiled using the device-specific backend compiler and loaded to the switch and external devices. Somtimes, a backend compiler could fail to compile the merged program due to its proprietary heuristics for resource allocation. In such a case, we try more conservative allocation.

In sum, ExoPlane orchestrator allocates resources across multiple apps based on inputs from developers and network operators and produces a merged P4 program loaded to devices. This process needs to be re-run when a set of applications or workloads changes, which we do not expect to happen very frequently (*e.g.,* for every hour).

## 4.5 Implementation

### 4.5.1 ExoPlane Runtime

We implement the ExoPlane runtime that is compatible with our testbed setup consisting of an Intel Tofino-based Wedge 100BF-32X programmable switch [39] and servers equipped with Netronome Agilio CX Smart NICs [11].

**Data plane.** We implement the data plane components of the ExoPlane runtime in P4-16. It consists of the following modules: (1) per-app ExoPlane runtime object implemented using a match-action table (both on the switch and external devices) and (2) global logical to physical external device ID mapping implemented using a register array (on the switch). To track the popularity of flow keys, on the switch side, we enable the aging feature to the ExoPlane runtime object while on the NIC side, we use counters associated with the object. Note that our application merger inserts these data plane runtime components as the part of a merged P4 program.

**Control plane.** We implement the control plane components of the ExoPlane runtime in Python and C++. The main capability needed here is to initialize new flow entries and promote new popular flows' entries on the switch based on the information reported by the data plane runtime components. To implement this, on the switch side, we use Barefoot Runtime APIs to access the stateful object in the switch data plane and on the smart NIC side, we use Netronome thrift APIs to interact with the NIC data plane. The switch and the external device control planes are communicated via an out-of-band TCP session over the 1 Gbps management network.

### 4.5.2 Tracking State Popularity

Our main assumption in ExoPlane operating model is having an ability to detect primary-key-based popular flows and place their entries to the switch. At first glance, it appears that we can use compact approximate data structures such as the count-min sketch [67] to detect "heavy" primary keys which are frequently looked up at the ExoPlane runtime object. Several studies have shown that it is possible to implement such data structures in the switch data plane [90, 102] at the cost of some switch resources. However, in our context where we do not have enough switch resources, it could be infeasible to use such compact data structures for each app in the switch data plane.

Instead, we design a switch resource-efficient flow key popularity tracking mechanism based on the fact that in ExoPlane, packets corresponding to *potentially* popular flows are routed to an external device which has a larger amount of resources. This allows us to detect new heavy flows at the external device using existing sketch-based heavy-hitter detections.

Fig. 4.15 illustrates how our flow key popularity tracking mechanism works. On the external device, we use the count-min sketch [67] to track the frequently accessed flow keys. When it detects a new popular key, it reports the key to its control plane which maintains a list of reported flow keys and corresponding entries (①), and they are reported to the switch control plane. On

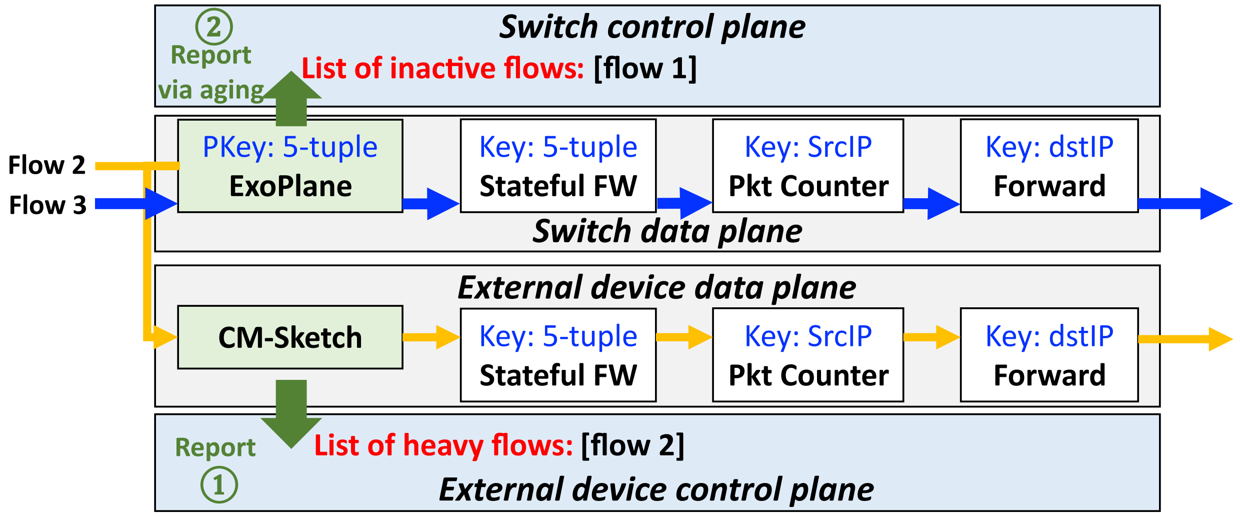


**Figure 4.15:** Resource-efficient flow key popularity tracking.

the switch side, we enable the aging feature for the ExoPlane runtime object. If a certain key of the ExoPlane runtime object has not been accessed for a timeout period ($T_{idle}$), a callback function registered at the switch control plane is triggered along with the information about the idle key (②). In our prototype, we set $T_{idle}$ to 10 seconds.

### 4.5.3 ExoPlane Orchestrator

**Optimal resource allocator.** We implement the resource allocator in C++ based on the Gurobi C++ API [8] to encode and solve our resource allocation ILP.

**Application profiler and merger.** We extend the open-source P4 compiler [14] to parse and analyze input P4 programs. Using its frontend, we extract information from each program including the key and action data sizes of each object. We implement the application merger in C++, which takes an IR generated by the compiler frontend, and produces a merged P4 code based on the logic described in subsection 4.4.4.

## 4.6 Evaluation

We evaluate ExoPlane on a testbed consisting of a programmable switch and servers equipped with a smart NIC using both real packet traces and synthetic packet traces. Our key findings are:

- (Latency and throughput in steady-state)
- (Scalability)
- (Workload changes)
- (Failover)
- (Resource overhead)

| Applications | States |
|---|---|
| Per-VM NAT | Per-flow address mapping for each VM. |
| Per-VM Firewall + Packet counter | Per-VM TCP connection list. |
| Per-VM DDoS mitigation + Packet counter | Per-VM SYN proxy and DNS amplification defense. |
| NetCache [90] | Key-value store cache. |

**Table 4.2:** Switch programs written in P4 used in the evaluation in addition to ones introduced in Table 4.1.

**Testbed setup.** We build a rack-level resource augmentation architecture consisting of Wedge100BF-32X Tofino-based programmable switches [39] and 4 servers equipped with Netronome Agilio CX 40 Gbps Smart NICs [11]. We use 4 additional servers to generate traffic workloads. All servers are equipped with an Intel Xeon Silver 4110 CPU and 128 GB DRAM, running Ubuntu 18.04 (kernel version 4.15.0). We repeat each experiment 100 times unless otherwise noted.

**Traffic workloads.** We use both packet traces collected from a real data center network [20] and synthetically generated ones. The packet sizes vary (64–1500 B) in the real trace. We generate packet traces with the flow key distribution in terms of the number of packets per flow that follows a Zipf distribution with the skewness parameter ($\alpha$=0.99). 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. In experiments, we replay the traces using DPDK-pktgen [21] for UDP workloads or run iperf [28] for TCP workloads.

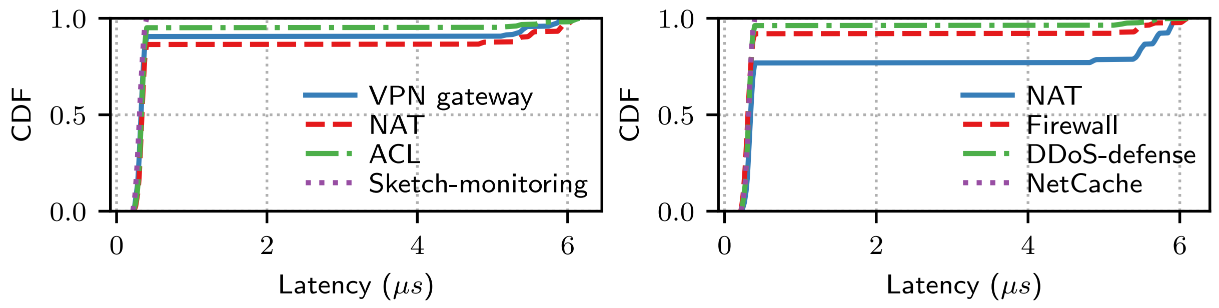
**Applications.** To demonstrate the applicability of ExoPlane, we implement new or adopt existing in-switch P4 applications described in Table 4.1 and Table 4.2.

**Deployment scenarios.** We use two scenarios where multiple P4 applications are running on our rack: (1) at the data center front-end, 4 applications described in Table 4.1 are running and (2) at the leaf of the network, 4 applications described in Table 4.2 are running. Given packet traces, we synthesize inputs from developers and the network operator for the optimal resource allocator in ExoPlane orchestrator (*e.g.,* per-app affinity, the flow key distribution ($\overrightarrow{D_p}$)). For example, for the per-app affinity, we set the level for the sketch-based monitoring and NetCache to high so that workloads for these apps are always processed at the switch. Then our resource allocator and application merger generates a merged P4 program, and we compile and load it to the switch and external devices (*i.e.,* Netronome NICs).

## 4.6.1 Performance in Steady-State

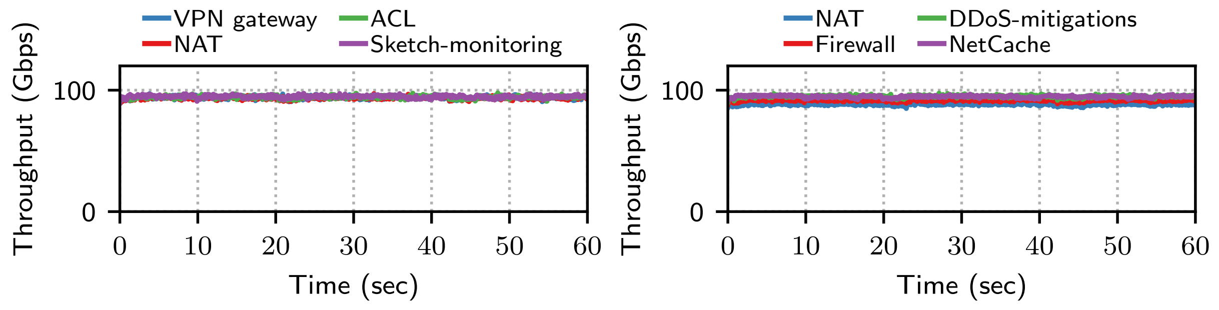
First, we evaluate the per-packet processing latency and throughput of applications running on ExoPlane in steady-state (*i.e.,* no workload changes or device failures).

**Per-packet processing latency.** We define the processing latency as the time difference between when a packet first arrives at the switch from a sender node and when it is sent back to the sender node after processing. To measure it, we instrument the P4 program running on the switch to



**(a)** Data center front-end. **(b)** Data center leaf.

**Figure 4.16:** Per-packet processing latency distribution of applications running on ExoPlane in steady-state.



**(a)** Data center front-end. **(b)** Data center leaf.

**Figure 4.17:** Throughput of applications running on ExoPlane in steady-state.

record two timestamps (48-bits each) to our custom packet header fields of each packet so that the sender node can compute the processing latency a packet. From sender nodes, we replay the data center packet traces, each of which contains more than 600K packets, and let each application sends packets back to the sender nodes after processing them. In this experiment, we use one external device.

Fig. 4.16 shows the CDF of the per-packet latency distribution for each application. For the apps that are assigned to the *high* affinity (Sketch-monitoring and NetCache), every packet is processed at the switch, where each packet is processed in 215–398 $ns$ depending on the packet size. For other applications, the latency distributions vary depending on how much traffic is processed at the switch and the external device. The higher affinity level assigned to an app, the more traffic is processed at the switch. For example, in the data center front-end scenario (Fig. 4.16a), at the switch, the ACL processes ≈90% of its traffic whereas the NAT processes ≈80% of its traffic. At the external device, packets are processed in 5.9–6.1 $\mu s$. The key takeaway from this experiment is while there is latency gap between the switch and the external device, on each device, per-packet processing latency is predictable.

**Application throughput.** To measure the application throughput, we replay at synthetic packet traces that consist of 1500 B packets at line-rate (98.6 Gbps in our testbed) from each workload generator node. We use four workload generator nodes, each of which generates traffic for each of four apps (*e.g.,* node 1 generates traffic for app 1). We assume that the traffic is equally dis-

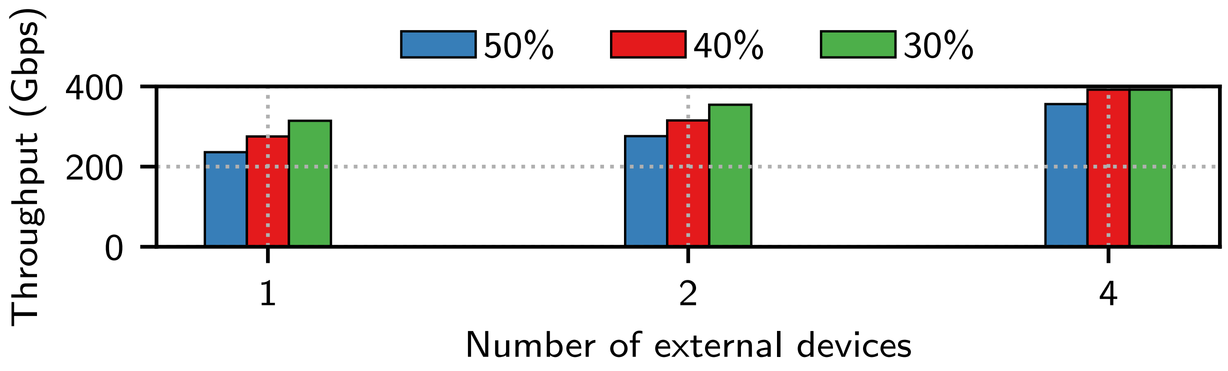


**Figure 4.18:** Scalable application throughput with multiple devices. Different colors represent the different fraction of traffic routed to external devices.

tributed across the apps (*i.e.,* $W_p = 0.25$ for all apps). Each application sends processed packets back to the sender node to let it measure the throughput. While the ExoPlane orchestrator allocates resources based on the assumption of four external devices, here, we start with running the apps with a single external device to demonstrate the impact of the number of external devices to the throughput.

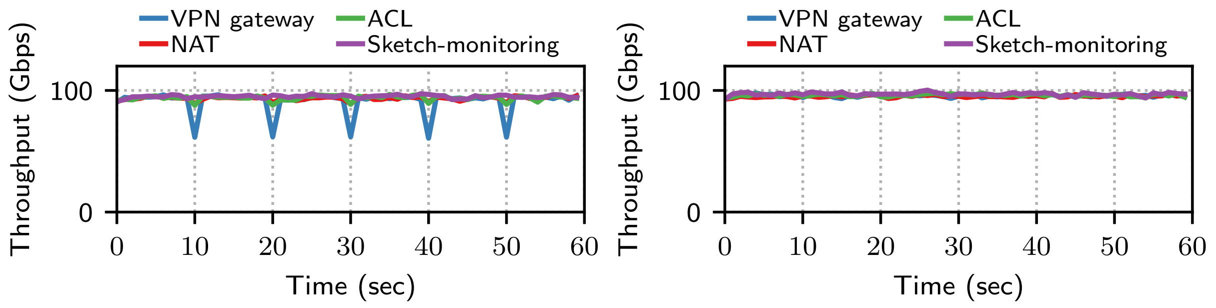
Fig. 4.17 shows the throughput of each application. The apps that run entirely on the switch (Sketch-monitoring and NetCache) process their traffic at line-rate without dropping any packets. However, we observe that other apps cannot process their traffic at line-rate. This is mainly because the aggregate amount of traffic across the apps, which needs to be processed at the external device (≈45 Gbps in the front-end case) exceeds the processing capacity of the single device (40 Gbps).

**Scaling throughput with multiple devices.** Fortunately, by adding more devices, ExoPlane can support higher application throughput. To demonstrate this, we measure the aggregate throughput of the four apps of the front-end scenario (*i.e.,* maximum traffic rate is 400 Gbps) while varying the fraction of traffic routed to external device(s)[6] and the number of external devices. Fig. 4.18 show the results. We can see the throughput effectively increases as we add more devices. This is because as described in subsection 4.3.4, serving popular flows at the switch helps balance accessing loads across multiple devices. In an extreme case where 50% of the traffic are routed to external devices, the applications cannot process the entire traffic even with four devices because the amount of traffic routed to the external devices exceed the capacity of the external devices. However, in more common cases, ExoPlane can support the entire traffic with four devices.

## 4.6.2 Performance with Workload Changes

**Per-packet processing latency.** As mentioned in subsection 4.3.2, workload changes happen when new flows arrive or some flows become more popular. Handling new flows in ExoPlane

[6] In this experiment, we control the fraction of traffic routed to external devices by manually choosing the affinity level of each app.



**(a)** With a single external device. **(b)** With 4 external devices.

**Figure 4.19:** Throughput changes of applications running on ExoPlane due to workload changes with a single and multiple external devices.

can increase per-packet processing latency because the first packet of the new flow can be processed only after installing necessary state. In contrast, packets in the flow that becomes popular can be processed either at the switch or an external device with the same latency shown in subsection 4.6.1. Thus, for each application, we measure the processing latency of the first packet of each flow. We find that the median processing latency for the first packet of a new flow is 32 $ms$, which is an order of magnitude higher than the steady-state processing latency at external devices. We find there are two factors that contribute to this latency. First, the thrift-based Netronome control plane APIs take a few tens of $ms$ to insert new entries to the objects in the data plane, which is not a ExoPlane-specific overhead. Second, since ExoPlane replicates entries for new flows to one another external device, it incurs additional latency for handling new flows.

**Application throughput.** While the changes in flow popularity do not affect the latency, it can impact the application throughput. To measure the throughput changes, we use the same experimental setup as the previous throughput measurement in steady-state, but for every 10 second, we alter the most popular top 10 flows for the VPN gateway app of the data center front-end scenario. Fig. 4.19 shows the application throughput changes over time. Again, we first use a single external device. As shown in Fig. 4.19a, when the popularity changes, there is a sharp drop in the throughput of the VPN gateway app. Also, the throughput of other apps slightly decreases as well. This is because until the state entries for the new set of popular flows are installed at the switch (*i.e.,* a transient period), a high volume of traffic for the flows are routed to the external device, exceeding its processing capacity. On the other hand, with 4 external devices, there is no such performance drop because the processing capacity of the external devices is large enough to handle the traffic during the transient period.

## 4.6.3 Failover

Next, we measure how fast the end-to-end performance can be recovered by ExoPlane in the presence of external device failure. We run iperf [28] to measure TCP throughput changes between two servers. We assume that all traffic passes through a NAT application running on an external device to make all traffic be affected by the device failure. There are two external devices enabled, and we compare changes in TCP throughput when (1) there is no failure (Baseline) and

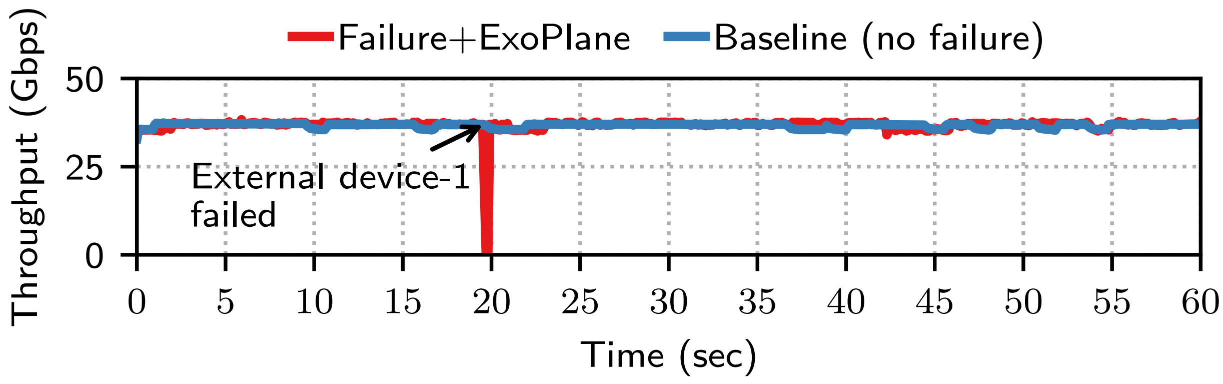


**Figure 4.20:** End-to-end TCP throughput changes during failover and recovery.

| Resource | Additional usage |
|---|---|
| Match Crossbar | % |
| Meter ALU | % |
| Gateway | % |
| SRAM | % |
| Hash Bits | % |

**Table 4.3:** Switch ASIC resources used by the ExoPlane runtime.

(2) one of the external device fails. We emulate the failure by disable a physical port connected to the external device.

Fig. 5.14 shows the results. At around 20 second, when the external device-1 goes down, the packet generation engine is triggered to generate our special packet that modifies the logical to physical external device ID mapping in the data plane. After this, subsequent packets are routed to a replica device. We see that the TCP throughput is recovered to its original rate within a 200 $ms$ second.[7]

## 4.6.4 Resource Overheads

**Control plane resource overhead for state synchronization.** Our state synchronization protocol (subsection 4.3.3) consumes some amount of control plane memory resources to maintain the history of other device(s) for each state entry. In terms of network bandwidth overhead, the control plane uses the control network bandwidth to periodically exchange the information between devices over an out-of-band TCP channel. Each packet contains a snapshot and a history which has the same size as the state entry.

**Switch ASIC resource usage.** Table 4.3 shows the additional switch ASIC resource consumption of ExoPlane for XXX popular flows (using the P4 compiler's output), expressed relative to each application's baseline usage. Overall, there are ample resources remaining to implement other functions along with ExoPlane. SRAM is the most used resource (XXX%); usage of ALUs

[7]The finest sampling granularity supported by iperf is 100 $ms$.

and other computational resources are all less than XX%. In terms of scale vs. number of concurrent flows, only the SRAM usage would increase proportional to the number of flows as it stores per-flow information.

## 4.7 Alternative Architectures

When is the ExoPlane approach likely to beat alternative architectures (*e.g.,* a regular switch with a bunch of programmable NICs)?

## 4.8 Related Work

**Language support for P4 program composition.** There have been works that propose new programming languages or abstractions to compose multiple programs or support heterogeneous hardware platforms. MicroP4 [147] proposes a variant of P4 that makes it easy to compose different pieces of programs into a single program that runs on a single device. Lyra [76] is a high-level language that is designed to express a switch program that can be partitioned and loaded onto multiple heterogeneous switch devices. SNAP [53] proposes a programming abstraction that allows network operators to express network-wide packet processing tasks.

**Data plane virtualization.** While some prior works attempt to support multiple P4 applications or modules in a switch [84, 147, 161, 163], none of them considers cases failed by limited switch resources. Virtualization approaches such as Hyper4 [84] and HyperV [161] allow composing multiple P4 applications with a constrained programming model. P4Visor [163] provides a resource-efficient merger that merges different versions of a program. However, they fail to work when the amount of resource required by the composed application exceeds the available resources in the switch. Moreover, adding more resources to the switch or using multiple switches would not be a viable solution; given the ever-increasing workloads [34, 65] and the number of in-switch applications [85, 99], adding expensive resources does not fundamentally solve the problem and is not cost-efficient.

**Switch resource augmentation.** TEA [102] provides a virtual memory abstraction that allows a single switch app to access remote DRAM and store lookup tables, it requires to use its APIs and is optimized for a single app with a single table. Flightplan [148] takes a single app written with custom annotations and disaggregates it to multiple devices. Developers need to be aware of external devices and manually partition the app so that each device runs only a particular portion of the app. Lyra [76] proposes a custom language for writing a single app that is disaggregated across multiple heterogeneous switches.

**NF state management.** Fault-tolerance for NFs or middleboxes has been addressed by prior systems like Pico [135] and FTMB [145]. When an NF instance fails, the state of the failed NF is recovered through checkpoint or rollback recovery on a new NF instance. These approaches cannot be applied directly to a heterogeneous data plane. Previous work on state management for server-based stateful NFs uses local or remote storage to manage NF state [77, 136, 159]. However, these APIs target planned state migration rather than unplanned failures. Similar work (again, targeting planned migration) has also been proposed for router migration [98]. Our recent work shares our approach of using servers' memory as external storage for switch state [102], but towards a different goal: allow switches to handle state larger than their on-device memory. It does not address fault tolerance or multi-writer consistency. Other recent work runs coordination protocols between switches to build reliable storage [91]. Our goal is conceptually different – to replicate state for in-switch applications rather than provide a networked storage service – but uses some similar mechanisms, like network sequencing [107]. Some recent work focuses on compile-time data plane multiplexing specifically for the P4 behavioral model [84, 163]. Our focus is on runtime resource management atop an ensemble target.

## 4.9 Summary

# Chapter 5

# Supporting Fault-Tolerance for Stateful In-Network Applications

Today's data center switches are no longer simple stateless packet forwarders. They implement sophisticated network functions, such as NATs, firewalls, and load balancers [27, 97, 120] and accelerate distributed applications [108, 109, 141, 152, 165]. Cloud service providers have even started deploying them in production networks [26].

Such stateful processing in switches leads to a new challenge: fault tolerance. Classic network designs followed the end-to-end principle [139], keeping critical state only on the end hosts. This enabled a fate-sharing approach to reliability [66]; when switches are stateless, recovering from their failure simply entails finding a new communication path. Stateful in-switch applications [26] challenge this paradigm; *e.g.,* the failure of a switch running a load balancer may cause the loss of its forwarding state, breaking thousands of active connections. While data center networks are engineered with redundant network paths [81, 138, 146] to provide fault tolerance at the routing layer, there are no capabilities for recovering in-switch state after failure.

Thus, we need to reconsider fault tolerance for in-switch processing – something previously done in ad hoc, application-specific ways. Our goal in this paper is to ensure that, after a failure and reroute, the same application state becomes available at the replacement switch, without degrading performance and while remaining transparent to end hosts.

Making switch state fault tolerant is uniquely challenging because of the scale and resource constraints involved. Techniques like checkpointing and active replication, which have been applied to software middleboxes [135, 145], are designed for server-based systems. These techniques rely on obtaining a consistent snapshot of state and buffering output until state updates are durably recorded to other servers. However, a switch's high packet processing speed (a few *billion* packets/second [44, 46, 47]) and its limited compute and storage capabilities make it infeasible to translate these techniques to the switch context.

In this paper, we introduce *RedPlane*,[1] a fault-tolerant state store for in-switch applications. RedPlane provides APIs for developers to (re)write their stateful P4 programs and make them fault-tolerant. This allows an application to retain consistent access to its state, even if the switch it runs on fails or traffic is rerouted to an alternative switch. RedPlane achieves this through a

[1]The name denotes a *replicated data plane*.

data plane centric replication mechanism that continuously replicates state updates to an external state store implemented using DRAM on commodity servers. Note that running entirely in the data plane channel is key to keeping up with the switch's full processing speed.

Realizing this high-level idea in practice entails several challenges. First, traditional notions of strict correctness with linearizability and exactly-once semantics for operations require reliable communication and output buffering. However, this is infeasible on the switch data plane due to its limited capabilities. Second, at the traffic volumes the switch data plane needs to process, naïvely requiring per-packet coordination with the server-based state store imposes severe performance overheads. Last, routing decisions when a switch fails could be unpredictable. Thus, we must be able to transparently migrate the relevant state between two switches regardless of the routing decisions.

We address these challenges with the following key ideas:

- Based on the requirements of in-switch applications, we define two practical correctness models. First, based on our observation that network applications are already resilient to packet loss, we define a strict consistency mode by explicitly adopting the standard definition of linearizability [87], which permits operations that do not complete while still providing strong consistency. Second, for write-centric applications (*e.g.,* monitoring using sketches [67]) that can tolerate approximate results, we propose a relaxed consistency mode that allows some state to be lost after a failure, but bounds the inconsistency with lower overheads.
- Instead of buffering packets using limited switch resources, we use the network itself and state store's memory as temporary storage by piggybacking packet contents on coordination messages.
- To enable reliable state replication, we build a lightweight sequencing and retransmission protocol that ensures state updates are processed in the correct order, without requiring complex protocols (*e.g.,* TCP) in the switch data plane.
- To avoid overheads due to frequent coordination with the state store, we propose a lease-based state ownership protocol [80, 111, 125] to provide correctness without coordinating on every state access and migrate ownership between different switches as needed.

We design the RedPlane protocol that realizes our consistency modes, prove its correctness, and confirm this using a TLA+ model checker [45]. We implement a prototype of RedPlane in P4 [38] and C++ and Python, and show that different types of applications can be fault tolerant using it. We evaluate it with various applications in our testbed consisting of two Tofino-based programmable switches, four regular switches, and 10 servers. Our evaluation results show that under failure-free operation, RedPlane has negligible per-packet latency overhead for read-centric applications like NAT, and less than 8 $\mu$s overhead even for the worst case. When a switch fails, RedPlane can recover end-to-end TCP throughput within a second by accessing the correct state.

## 5.1 Background and Motivation

In-network processing has flourished in recent years, as a natural convergence of the demand for sophisticated network functionality from data center operators and the commercial availability of programmable switch platforms [32, 44, 46]. Programmable switches are used for classic middlebox functionality [97, 120], monitoring [27, 83], DDoS defense systems [160, 162] and accelerating other networked systems [90, 107, 108, 109, 117, 141, 152, 165].

These applications are *stateful*; *i.e.,* state on the switch determines how to process packets. In this paper, we focus primarily on *hard state* applications, where a loss of state disrupts network or application functionality.[2] An example is an in-switch NAT, where the key state is an address translation table. Losing this state would make it impossible to forward packets for existing connections.

**Network model.** We consider a deployment model where programmable switches are installed into the network fabric such that all traffic to be processed by an in-switch application traverses one of the programmable switches. This could be achieved in several different ways, depending on the network architecture. In a typical data center architecture (Fig. 5.1), this could be achieved by using the switches on all core or all aggregation-layer switches.[3] All traffic entering or leaving a cluster, for example, would traverse one of these switches. Alternatively, an operator might deploy a cluster of programmable switches as dedicated "NF accelerators", explicitly routing traffic through them; this approach is similar to how software load balancers [72, 130] are deployed today.

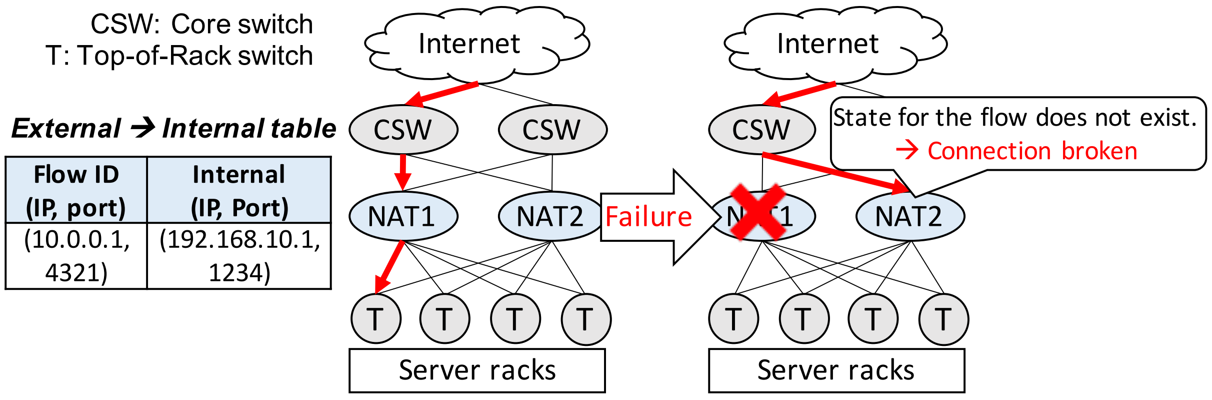
**State partitioning.** We assume that application state is partitionable using some key derived from the packet header, and that each packet's processing uses only state from the associated partition. In many cases, such as for the NAT example, the key will be the IP 5-tuple, and, hence, we use "partition" and "flow" interchangeably. However, other applications may use different partitioning, *e.g.,* partitioning on VLAN ID to detect heavy-hitter flows for a particular tenant.

We also assume that the network is configured to provide best-effort affinity such that packets from the same partition usually arrive at the same switch. Standard layer-3 routing protocols such as Equal-Cost Multi-Path routing (ECMP) provide this property when they are configured to use the partition key as their hash key.

**Primer on programmable switches.** Programmable switch architectures used today, *e.g.,* Intel Tofino [46], use a limited amount of on-chip memory (*e.g.,* SRAM and TCAM) to provide a variety of stateful object abstractions, including tables, registers, meters, and counters. Applications can use these to keep state across multiple packets, such as the address translation table in the NAT example above. In the ingress and egress match-action pipeline, objects are allocated in each stage and accessed by packets via ALUs. These objects are also accessible by the switch control plane through the ASIC-to-CPU PCIe channel which has a limited bandwidth ($O(10$ Gbps$)$) compared to the ASIC's per-port bandwidth ($O(100$ Gbps$)$). In addition, the ASIC

[2]Other applications maintain only soft state in the switch and provide their own failure recovery mechanisms. These are not the focus of our work, though RedPlane could perhaps help simplify their design or improve recovery performance.

[3]In principle, RedPlane could be deployed on top-of-rack (ToR) switches, but it is potentially less useful. If each rack has one ToR switch, and it fails, connectivity to the servers in that rack is lost. RedPlane can restore the switch state onto a different rack, but depending on the application that may not be useful. However, if there are two ToR switches per rack, RedPlane would be useful.



| Flow ID (IP, port) | Internal (IP, Port) |
|---|---|
| (10.0.0.1, 4321) | (192.168.10.1, 1234) |

**Figure 5.1:** Impact of switch failures on in-switch NATs.

| State access | Applications | Impact of switch failure |
|---|---|---|
| Read-centric | NAT | Connection broken |
| | Stateful firewall | Connection broken |
| | Load balancer [120] | Connection broken |
| | SYN flood defense [162] | Dropping valid packets |
| Write-centric | Super-spreader detection [149] | Inaccurate detection |
| | Heavy-flow detection [116] | Inaccurate detection |
| Mixed-read/write | SGW in EPC [143] | Active session broken |
| | In-network sequencer [107] | Incorrect sequencing |
| | Per-object routing [109, 165] | Choosing wrong servers |
| | In-network key-value store | Losing key-value pairs |

**Table 5.1:** Examples of stateful in-switch applications and impact of switch failures.

provides other built-in functionality such as packet replication, recirculation, and mirroring for more advanced packet processing.[4]

## 5.1.1 Impact of Switch Failures

Switches can fail, either by a switch failing entirely (a fail-stop model), or by individual links losing their connectivity. Measurement studies in production data centers have shown that such switch failures are prevalent. For example, in Microsoft's data center, 29% of customer-impacting incidents are related to hardware failures including ASIC failure, fiber cuts, or power failures [112], and in Facebook's data center, 26% of incidents are related to switch failures [119].

Switch failures can impact stateful applications in two ways. If a switch fails entirely, all application state it held is lost. Beyond that, a link failure or the failure of a *different* switch

[4]While we use Tofino-based programmable switches for our work, we believe our design can be implemented on other programmable switch ASICs since hardware capabilities leveraged in RedPlane's switch data plane (*e.g.*, packet mirroring) are general features supported by most switch ASICs.

can impact many paths in the network [114], causing traffic to be rerouted [52, 81, 104]. Traffic that previously traversed one switch might be routed to a different one, where the appropriate state is unavailable. In the absence of this state, application processing can fail. For example, as illustrated in Fig. 5.1, lacking the proper translation table entries, the NAT cannot forward packets for existing connections, breaking open connections *en masse*. Indeed, this is a serious problem – software-based stateful load balancers at cloud providers implement complex failover mechanisms [72, 130].

Beyond the conventional NFs (*e.g.,* NATs, load balancers, firewalls), there are several in-switch applications (shown in Table 5.1) that exhibit complex state access patterns. For example, many applications that are designed to enforce QoS policies (*e.g.,* rate limits) employ streaming algorithms (*e.g.,* sketching) to capture characteristics of traffic such as heavy-hitters [116, 149]. Switch failures lead them to make inaccurate decisions as the statistical data is lost. Such applications update state (*e.g.,* sketches) on every packet, so we call them *write-centric*. In contrast, many conventional NFs and DDoS defense systems (*e.g.,* SYN proxy) [160, 162] are *read-centric*.

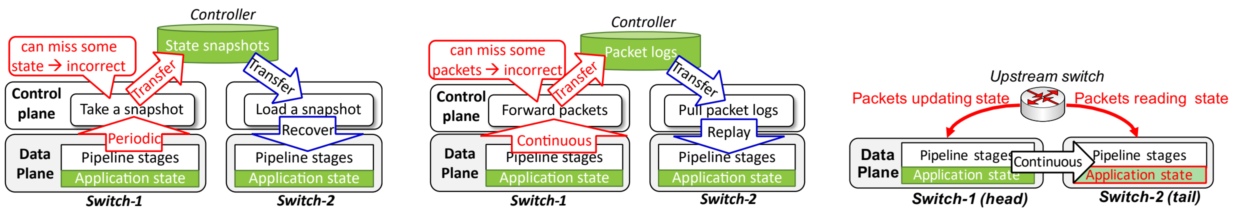
Another group of applications have *mixed-read/write* state access patterns, typically with much less frequent updates than write-centric applications. One example in this category is NFs in the packet core for cellular networks (*e.g.,* Evolved Packet Core (EPC) for LTE) [40]. Packet core NFs such as a serving gateway (SGW) route users' data traffic from user devices to the Internet and vice versa based on per-user states (*e.g.,* forwarding state), which are updated when the control plane receives signaling messages (*e.g.,* device attached). To cope with the increasing volume of signaling traffic [24, 34],[5] there have been recent efforts to accelerate the control plane functions by offloading them to the programmable data plane [25, 30, 131, 143]. For example, a SGW running on a switch maintains per-user tunnel endpoint IDs (TEIDs) to route packets, and this state is updated by signaling messages and read by data packets that are encapsulated with TEIDs. Thus, when a switch fails, since the SGW loses the state, it cannot forward packets for users, disrupting active connections. Affected users need to re-establish connections after the failure [49], increasing the service latency. Other applications that route requests in application-specific ways (*e.g.,* for databases [165] or key-value stores [109]) also fall into this category since they require state updates on every write (but not read) request.

### 5.1.2 Existing Approaches and Limitations

We now examine classical fault tolerance mechanisms [78, 126, 155] and mechanisms tailored for network middleboxes [135, 145]. At a high level, these approaches can be categorized into three classes: (1) checkpoint-recovery, (2) rollback-recovery, and (3) state replication. All prior work targets server-based implementations. In what follows, we discuss why natural adaptations of these approaches to the switch environment fail to ensure correct behavior during failures.

**Checkpoint-recovery.** Checkpointing approaches periodically snapshot application state (*e.g.,* an address translation table in NAT) and commit it to stable storage (*e.g.,* [135]). When a failure occurs, the latest snapshot is populated on a backup node (*i.e.,* an alternative switch in our con-

[5]Despite the growth, it is expected that signaling traffic rate is still much lower than that of data traffic (*e.g.,* 5% of data traffic [123]).



**(a)** Checkpoint-recovery : Switch control plane periodically snapshots and commits state to the controller. During this time, all packets must be buffered.

**(b)** Rollback-recovery : Each packet is forwarded to the control plane and logged by the controller.

**(c)** State replication : Switches replicate state to data plane memory using chain replication. Packets must be routed to the correct chain node.

**Figure 5.2:** Highlighting why adapting existing approaches for fault tolerance fails for hardware switches.

text). Fig. 5.2a illustrates a candidate implementation on switches using an external controller to store snapshots via the switch control plane. To achieve a consistent snapshot, data plane execution must be paused and packets buffered during the snapshot period. Limited data-to-control plane bandwidth in modern switch architectures makes this impractical.

**Rollback-recovery.** This approach, previously used for software middleboxes [145], logs every packet to stable storage and replays the traffic logs on a new device after failure to reconstruct application state. A natural implementation is sending every packet to the switch control plane, which logs it to the controller (Fig. 5.2b). In principle, this approach can guarantee correctness if every packet is synchronously logged and replayed after a failure. However, the mismatch between the data traffic rate (*Tbps*) and the data-to-control plane bandwidth (*Gbps*) will result in many packets being dropped and will, thus, be incorrect.

**State replication among switch data planes.** Consider a state machine replication approach using chain replication [155], but applied to switch data planes (Fig. 5.2c). Packets are forwarded through a sequence of switches, each of which updates its state and forwards the packet to the next switch in a chain. Only once the packet has reached the tail of the chain is it forwarded on its way to its destination. This is done entirely on the data plane, so it can function at high speed. This approach achieves correctness *only if* state updates are not lost. However, the state updates are delivered over an unreliable channel, and since the switch data plane cannot effectively support reliable transport protocols (*e.g.,* TCP) updates could be lost or reordered, violating correctness. Also, using one switch to replicate another switch's state makes poor use of data plane-accessible switch memory – the most costly and limited resource. It also requires changes to the routing policy of the network since a packet needs to be explicitly routed to a specific switch in the chain depending on whether the packet updates state or not.

**Takeaways.** From the above discussion, we see two key takeaways. First, approaches that rely on the switch control plane must consider the mismatch between control and data plane speeds. Second, while switch data-plane-only approaches can provide good performance, they suffer three shortcomings: (a) incurring significant switch resource overhead; (b) making it difficult to reason about correctness due to unreliable communication channels between switch data planes; and (c) they may additionally constrain routing policies.

## 5.2 RedPlane Overview

Our goal is to design a fault tolerance solution that provides the following four properties:

- **Correctness**: Switch failure should be transparent to applications: clients should not see state that would not be possible in the absence of a failure.
- **Performance**: Under failure-free operation, overhead for per-packet latency should be low (say, a few tens of $\mu s$).
- **Low resource overhead**: It should not consume switches' limited compute and storage resources excessively.
- **Transparency to routing policies**: That is, we must allow a packet to update and/or read state regardless of the location of a switch where the packet is routed.

To this end, we present RedPlane, which provides an abstraction of fault-tolerant state storage for stateful in-switch applications. RedPlane provides an illusion of "one big fault-tolerant switch" – the behavior is indistinguishable from the same application running on a single switch that never fails. To achieve this, RedPlane continuously replicates state updates which can be restored without loss after a failure.

RedPlane takes a state replication approach with two defining characteristics: (1) the switch's state replication mechanism is implemented entirely in the data plane, and (2) state storage is done through an *external state store*, a reliable replicated service made up of traditional servers. Property (1) means that the switch's control plane is not required for state replication, avoiding the issues with the checkpointing and rollback-recovery approaches of subsection 5.1.2. Property (2) means that the replicated state is stored in commodity server DRAM, a relatively low cost storage medium compared to switch data plane memory. This avoids the high resource overhead of the state replication approach discussed in subsection 5.1.2.

While the idea of using servers' memory as an external store is similar to recent work on TEA [102], it is important to note that TEA does not tackle fault tolerance. It focuses on the problem of resource augmentation to enable a switch to retrieve state stored in memory of servers. Furthermore, that design can only utilize servers *directly attached* to a top-of-rack switch. As such, their design does not tackle fault tolerance or provide provable correctness in the scenarios when multiple switches can access the store.

RedPlane provides a set of APIs (Fig. 5.3) implemented in P4 [38], a language to specify data plane programs on programmable switches, to allow developers to easily integrate RedPlane with their stateful P4 applications. Once developers (re)write their applications using RedPlane APIs, the P4 compiler generates a binary of RedPlane-enabled applications loaded to the switch, which continuously replicates updates to the state store through the data plane.

**Scope and limitations:** In this work, our focus is on enabling fault tolerance for stateful applications with partitionable hard state, where a loss of state disrupts network or application functionality, shown in Table 5.1. Applications only with non-partitionable state (*e.g.,* global counter) are beyond the scope of this work. Also, we assume that global state in an application (*e.g.,* a port pool in NAT) is sharded across and managed by state store servers. Other applications that

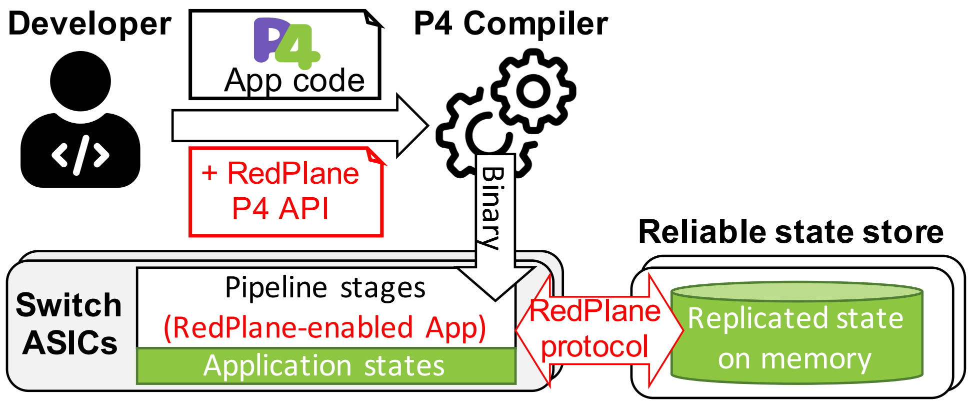


**Figure 5.3:** RedPlane overview highlighting extensions to traditional workflows for in-switch applications

need soft-state (*e.g.,* in-network caches or ML accelerators) do not require fault tolerance, but may benefit from RedPlane.

### 5.2.1 Challenges

While replicating state updates through the data plane to an external state store seems appealing, realizing this idea in practice presents some challenges:

**C-1. Providing correct replication in the data plane while tolerating unreliable communication.** Traditional server-based replicated systems aim to provide strict correctness by ensuring not just linearizability but also that each operation is executed exactly once even in the presence of dropped or retransmitted messages [105, 111, 125]. To do so, they build on reliable communication channels like TCP. However, the switch data plane cannot support reliable communication, nor can it buffer significant amounts of traffic.

**C-2. Handling high traffic volume.** Switch data plane operates at immense traffic volumes (up to a few billion packets per second [44, 47, 55]), in contrast to server-based systems handling a few million. If each packet that reads or updates state requires interacting with a server-based state store, the servers' capacity will rapidly be exceeded. It will also incur significant performance overhead.

**C-3. Being transparent to routing policies.** A switch failure, recovery, or network routing change could cause traffic flows originally processed at a switch S1 to be routed to a different switch S2. However, since the routing decisions may be unpredictable, we cannot make assumptions on S2 or presuppose what backup routes will be taken. That is, we must be able to transparently migrate the relevant state from S1 to S2 irrespective of the location of S2. For instance, we need to make the NAT table entry available when packets for a particular connection are processed by a different instance.

### 5.2.2 Key Ideas

To tackle these challenges, we build on four key ideas:

**I-1. Practical correctness for switch state (§5.3).** We define two correctness models based on the requirements of in-switch applications. The first, a strict consistency mode, is based on linearizability [87]. Because we observe that network applications are already designed to tolerate packet loss, we explicitly adopt the standard definition of linearizability, which permits operations that do not complete while still providing strong consistency for those that do. Second, since many write-centric applications (*e.g.,* monitoring using sketches [67]) accept approximate results, we propose a relaxed consistency mode that allows some state to be lost after a failure, but bounds the inconsistency.

**I-2. Piggybacking output packets (subsection 5.4.1).** Instead of buffering output packets using limited switch resources, we use the network itself as temporary storage by piggybacking packet contents on coordination messages.

**I-3. Lightweight sequencing and retransmission (subsection 5.4.2).** To cope with the unreliable communication channel between the switch data plane and the state store with low resource overhead, we employ a sequencing mechanism for protocol messages and devise a lightweight switch-side retransmission mechanism by repurposing the switch ASIC's packet mirroring feature.

**I-4. Lease-based state ownership (subsection 5.4.3).** To reduce the frequency with which the switch must coordinate with the state store, especially for applications with read-centric and mixed-read/write workloads, we adopt a lease-based mechanism inspired by prior work [80, 111, 125]. This allows us to avoid coordination with the state store for packets which need to read but do not modify state. At the same time, we ensure that all state updates are durably recorded before any of their effects are externalized, guaranteeing linearizability. This mechanism also serves as the means by which state is migrated between switches to support the transparency.

Taken together, these high-level ideas address the aforementioned challenges. First, the linearizability-based consistency model coupled with the piggybacking and lightweight sequencing and retransmission mechanism allows to replicate state reliably and correctly (C-1). Second, the relaxed consistency and lease-based state ownership help cope with high traffic volume (C-2). Lastly, the lease-based state ownership makes RedPlane transparent to routing policies (C-3).

## 5.3 Correctness Model

RedPlane provides two levels of consistency, which applications can choose between based on their requirements. A *linearizable* mode provides strict guarantees, making the system indistinguishable from a single fault-tolerant switch. Because this has a high overhead for write-centric applications due to frequent coordination with the state store, RedPlane also offers a *bounded-inconsistency* mode that permits some state updates to be lost on switch failure, but guarantees a consistent view of switch state.

### 5.3.1 Preliminaries

By default, RedPlane provides *linearizability* [87], a correctness condition for concurrent systems. We model a stateful in-switch program as a state machine, where the output and next state are determined entirely by the input and current state:

**Definition 1 (Stateful in-switch program).** A stateful program $P$ is defined by a transition function $(I, S) \rightarrow (O^*, S')$ that takes an input packet and the current state, and produces zero, one, or multiple output packets, along with a new state.

To simplify the definitions below, we will assume that each input packet $p$ produces exactly one output packet $P(p)$; it is straightforward to extend them to the zero- or many-output case. This implies that the program's behavior is determined entirely by the sequence of input packets, and in particular that it is deterministic and that packets are processed atomically. Although switch architectures are pipelined designs that process multiple packets concurrently [60], their compilers assign state to pipeline stages in a way that makes packet processing appear atomic [56].

The gold standard for replicated state machine semantics is single-system linearizability [87]. That is, that the observed execution matches a sequential execution of the program that respects the order of non-overlapping operations. To adapt linearizability for in-switch programs, we first redefine a history in terms of packet processing:

**Definition 2 (History).** A history is an ordered sequence of events. These can be either input events $I_p$, in which a packet $p$ is received at a RedPlane switch, or output events $O_p$ in which the corresponding output packet is output by a RedPlane switch.

Note that it is possible for there to be input events $I_p$ without the corresponding output $O_p$, if the processing of $p$ is still in process or due to a failure. We discuss this in depth next.

### 5.3.2 Linearizable mode

**Definition 3 (Linearizability for a stateful in-switch program).** A history $H$ is a linearizable execution of a program $P$ if there is a reordering $S$ of the input events in $H$ such that (1) the value for each output event $O_p$ that exists in $H$ is given by running $P$ on the input events in $S$ in sequence, and (2) if $O_x$ precedes $I_y$ in $H$ then $I_x$ precedes $I_y$ in $S$.

Here, $S$ is the apparent sequential order of execution.

Linearizability is fundamentally a safety property, not a liveness one: it specifies what output values are acceptable, but does not guarantee that all operations complete. It is possible for a packet to be received and (1) update switch state, but produce no output, or (2) neither update switch state nor produce any output. Definition 3 reflects this: a packet with an input event but no output event can still appear in the sequential order $S$. If it precedes the processing of other packets, then they see the effects of its state update. If it appears at the end of $S$, it has no visible effect on system state.

While these anomalies comport with the definition of linearizability, most replicated systems aim to provide a stronger property: that every operation is executed exactly once and returns its result to the client. Ensuring this requires several protocol-level mechanisms: typically, clients retry requests that do not receive a response, and replicas keep state to detect duplicate requests and resend the responses without executing them twice [105, 111, 125]. As we see (subsection 5.4.2), these techniques are not feasible in our environment.

Accordingly, RedPlane takes a different approach: it *explicitly permits* these two types of anomalies. While this may seem surprising, it matches the semantics of modern networks. The two cases correspond to a packet being lost (1) between the RedPlane switch and its destination or (2) between the source and the RedPlane switch, respectively. Network applications must already tolerate lossy networks, so they are resilient to such losses.

Relaxing the definition of correctness enables a tractable implementation. By not requiring the system to achieve complete reliability, our protocol may drop packets during failover, or if messages between a switch and the state store are lost. In these scenarios, an input packet or its output may be lost. Of course, dropping too many packets is undesirable for performance reasons; such loss events are rare.

### 5.3.3 Per-flow Linearizability

In most in-switch programs, some or all state is associated with a particular flow – a subset of traffic identified by a unique key, *e.g.,* an IP 5-tuple, VLAN ID, or an application-specific object ID. For example, each translation table entry in a NAT is tied to a specific flow based on an IP 5-tuple. For many applications, per-flow state is the only state that needs to be consistent or fault tolerant – either because there is no global state, or because global state can tolerate weaker consistency, *e.g.,* traffic statistic counters that need not be precise. RedPlane generally provides consistency for per-flow state (consistency for global state is optional):

**Definition 4** (**Per-flow linearizability**)**.** A history $H$ is per-flow linearizable if, for each flow $f$, the subhistory $H_f$ for the packets in flow $f$ is linearizable.

As long as programs use only per-flow state, per-flow linearizability is *the same* as global linearizability, because linearizability is a *local* (*i.e.,* composable) property [87]. The benefit of operating on a per-flow level is that it means synchronization between states associated with different flows are not required. As we show in §5.4, this allows RedPlane to distribute execution of a program across multiple switches: each has the state associated with certain flows, and can process packets for those flows. This matches the way many applications are deployed in practice, *e.g.,* a NAT will be deployed to a cluster of switches, using ECMP for load balancing. Because this load balancing is done on a per-flow granularity, each switch is responsible for performing translation for a subset of flows, and does not need access to the translation table for the other flows.

### 5.3.4 Bounded-inconsistency mode

RedPlane's linearizable mode uses a synchronous replication protocol (subsection 5.4.1), which can induce high overhead for write-centric applications. However, we observe that many write-centric applications in programmable switches operate in contexts where approximate results are acceptable, *e.g.,* monitoring using sketches [67] or Bloom filters [59]. For these applications, RedPlane offers a *bounded-inconsistency* mode that has lower overhead.

In this mode, RedPlane takes periodic snapshots of data plane state and replicates them asynchronously. This means that upon switch failure, the most recent state updates can be lost. However, RedPlane ensures that the system recovers to a consistent state from within a time interval $\epsilon$. RedPlane's consistent snapshot mechanism ensures that the state after recovery reflects an actual state of the system, which simplifies reasoning about the correctness of complex data

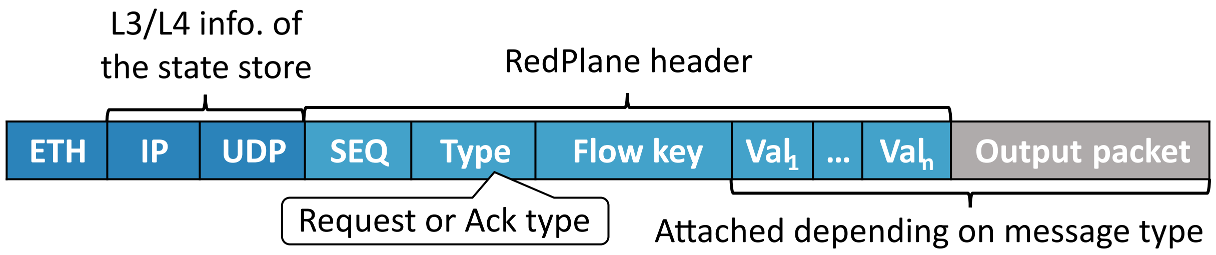


**Figure 5.4:** RedPlane state replication protocol packet format.

structures.[6] In subsection 5.4.4, we describe how we address key challenges in realizing this mode in RedPlane.

## 5.4 RedPlane Design

Now, we describe the RedPlane protocol that realizes our linearizable and bounded-inconsistency modes. We begin with an overview of the protocol and explain how we address practical challenges.

### 5.4.1 Basic Design

As shown in Fig. 5.3, RedPlane consists of (1) an external state store built on commodity servers and (2) a RedPlane-enabled application running on the switch data plane. In this section, we describe how the components work together via the state replication protocol.

For clarity of exposition, we start with simplifying assumptions: there is no packet loss or reordering between switches and the state store, switches do not fail while messages are in transit, and packets for a flow are routed to only one switch at a time. We revisit these assumptions in subsection 5.4.2 and subsection 5.4.3.

#### External state store:

The external state store is an in-memory key-value storage system. We partition it across multiple shards by flow – identified by an IP 5-tuple or other key. Each state store shard can be replicated using conventional mechanisms and we do not seek to innovate here as many existing key-value stores meet our needs (*e.g.*, [43, 110, 127]). Specifically, our prototype is a simple in-memory storage server implemented in C++ that uses chain replication [155] with a group size of 3.

[6]Although the bounded-inconsistency mode may affect properties of some approximate data structures (*e.g.*, no false negatives in Bloom filters), since it bounds the inconsistency within $\epsilon$, developers or network operators can easily reason about the potential inconsistency.

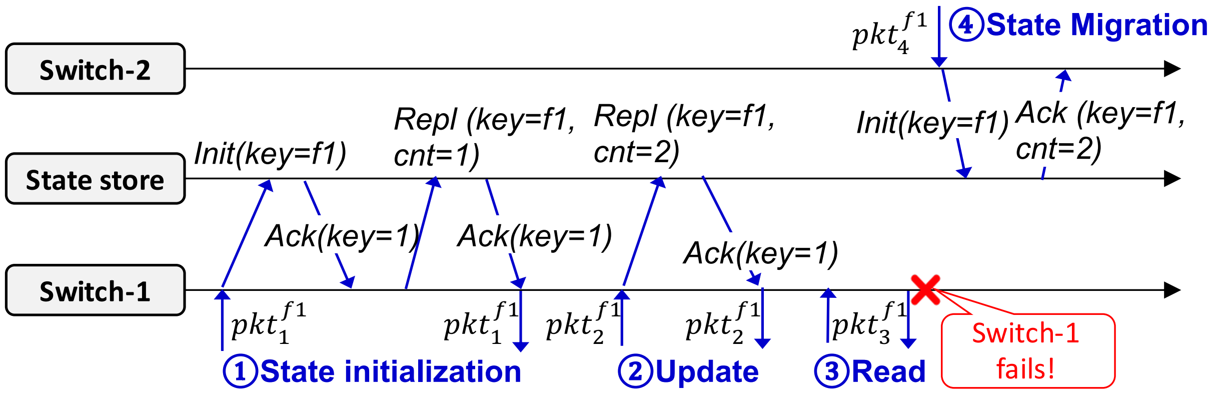


**Figure 5.5:** Basic workflow of RedPlane state replication protocol. "Repl" indicates a state replication request. $pkt_n^f$ indicates $n^{th}$ packet of a flow $f$.

**Basic replication protocol:**

A RedPlane-enabled application replicates state updates to the state store by exchanging protocol messages formatted as shown in Fig. 5.4. It uses standard UDP and IP headers to address messages to the state store or the switch using their respective IP addresses. The RedPlane header consists of a sequence number, a message type, and a flow key. Depending on the message type, it can also include flow state and an output packet. We will discuss these fields shortly. Note that we assign an IP address to each RedPlane switch and use it for routing requests and response packets between state store servers and RedPlane switches. This works with general L3 routing protocols including ECMP and BGP.

As an illustrative example to help understand the protocol, we consider a per-flow counter application shown in Fig. 5.5, This application updates or reads the state for each packet. In the example, there are two switches and a state store. We have multiple packets in each flow $f$, with the $n^{th}$ packet denoted as $pkt_n^f$. This example illustrates a case where the Switch-1 initially handles $f1$, but after its failure, the flow is rerouted to the Switch-2.

**State initialization or migration (Step ① or ④ in Fig. 5.5).** When the application receives a packet that belongs to a flow it has never seen before (*e.g.,* $pkt_1^{f1}$), it needs to send a *state initialization request*. It identifies the corresponding state store server by hashing the flow key (*e.g.,* IP 5-tuple), and looking up the corresponding server IP and UDP port from a preconfigured table.

There are two possible cases: (1) the flow is new and so has no state, or (2) the flow state previously existed on a failed switch, and a packets for that flow are now being routed to a switch on an alternative path (*i.e.,* failover). In case (1), upon receiving the request, the state store initializes its storage for the state and sends a response back to the switch (**Step ①**). In case (2), since the state store already has the flow state, it sends a response containing the latest state (**Step ④**).

Upon receiving the response, the application installs the returned state into the corresponding switch memory. For stateful memory registers, this can be done entirely in the data plane. On the Tofino architecture, updates to match tables or certain other resources need to be done through the switch control plane. In this case, RedPlane routes the processing through the control plane. This can introduce additional latency (we measure this in subsection 5.6.1). However, many in-switch applications already require a control plane operation on a new flow (*e.g.,* to install a new translation mapping in a NAT), in which case the added overhead is minimal.

**Reading or updating state (Step ② or ③ in Fig. 5.5).** Once the state has been initialized, the application can read the state value (*i.e.,* the counter in our example) directly (**Step ③**). When it updates the state (*i.e.,* the counter value), RedPlane sends a *replication request* with the new value to the state store. This message is generated entirely through the data plane. The state store applies the update, and sends a *replication reply* message (**Step ②**).

**Piggybacking output packets.** When the application updates the state, RedPlane should not allow an output packet to be released until the state has been recorded at the state store – otherwise, the update could be lost during a switch failure, violating correctness. This requires the output packet to be buffered until the replication reply is received.

Unfortunately, the switch data plane does not have sufficient memory to buffer packets in this way (and various other constraints on how memory can be accessed make it unsuitable for storing complete packet contents). RedPlane instead piggybacks the packet onto its replication request message, and the state store returns it in its reply. When the reply is received, RedPlane decapsulates and releases the packet. In effect, this uses the network and the memory on the state store as a form of delay line memory – trading off network bandwidth, which is plentiful on a switch, for data plane memory, which is scarce.

Note that it is possible to receive packets that read state when there are in-flight replication requests for the state. In this case, the packets are buffered in the same way through the network (with a special RedPlane request type) until a switch receives a response for the latest replication request.

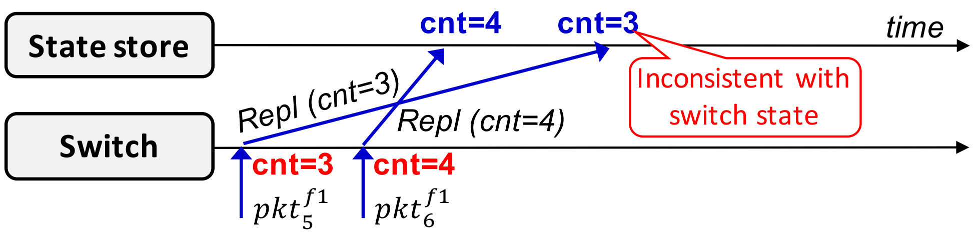
While our basic design provides correctness under the simplified assumptions, we find that in more realistic environments, it may not be able to guarantee correct behavior. In the following sections, we describe potential challenges, and how we extend the basic design to address them.

### 5.4.2 Sequencing and Retransmission

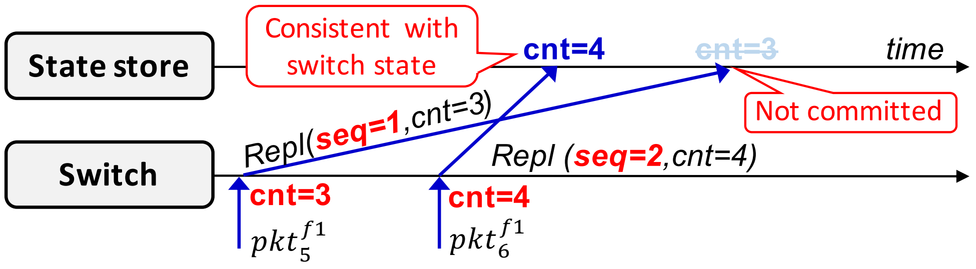
To guarantee correctness, replication requests must be *successfully* delivered and replicated *in order* at the state store. For example, the replication request (**Step ②** in Fig. 5.5) must be delivered in order. However, successful in-order delivery is not guaranteed in a best-effort network between switches and the state store.

Fig. 5.6a illustrates why such unreliability in the network can be problematic. We use the same per-flow counter as an example. Each time the counter is incremented, RedPlane sends the new value to the state store. If the state store just processes updates in the order they are received, a reordering could cause a later counter value to be replaced with an earlier one. Request loss can cause a similar issue.

A traditional replication system, like chain replication, might address this by relying on a reliable transport protocol like TCP. Unfortunately, it is not practical to implement a full TCP stack on the switch data plane – if it is possible at all, it would excessively consume data plane resources.



**(a)** Out-of-order delivery of requests causes inconsistency.



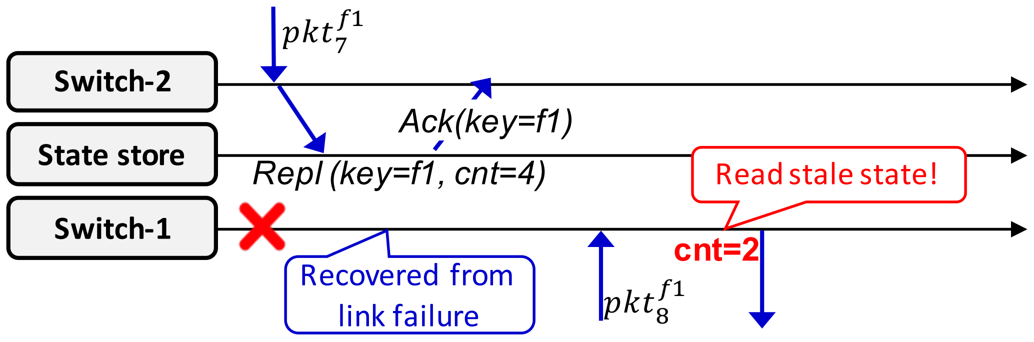
**(b)** Request sequencing serializes replication requests.

**Figure 5.6:** Serializing out-of-order requests with sequencing. Counter values (cnt) in red and blue indicate the state on the switch and the state store, respectively.

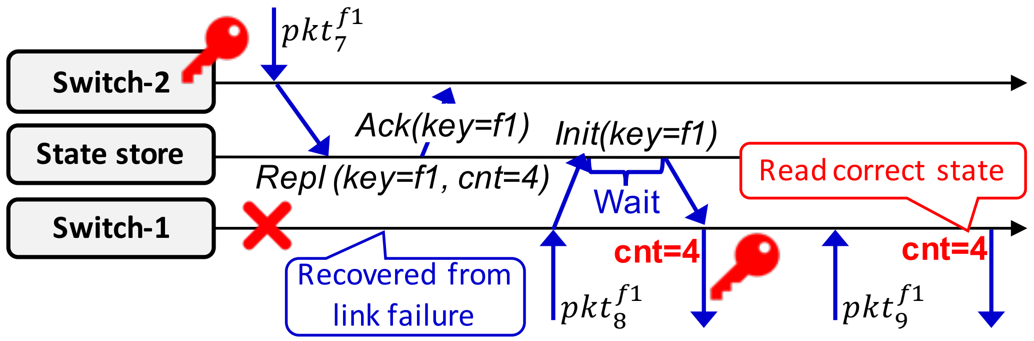
**Our approach.** Instead of implementing a full-fledged reliable transport on a switch data plane, we choose to build a simpler UDP-based transport with mechanisms that deal with possible packet reordering and loss. First, to handle out-of-order state replication request messages, we employ a mechanism called *request sequencing* [107], which assigns a per-flow monotonically increasing sequence number to each request message. The state store uses this sequence number to avoid applying updates out of order (Fig. 5.6b).

Second, to cope with lost replication requests or responses, we develop a mechanism for *request buffering*. RedPlane buffers replication requests and retransmits them if it does not receive a reply before a timeout. We implement this by repurposing the egress-to-egress packet mirroring capability of switch ASICs. When RedPlane sends a replication request, it mirrors a copy with the current timestamp as metadata. When the mirrored request enters the egress pipeline and it has not been acknowledged by a response with the same or a higher sequence number, RedPlane checks whether the request has timed out by comparing the current timestamp to the timestamp in its metadata. If it has timed out, it resends the request to the state store. Otherwise, it mirrors the request again without ending the request to the state store.

As discussed previously, buffering a full packet payload is challenging on a switch due to memory limitations. Instead, RedPlane buffers *only state updates* (*i.e.,* the RedPlane header) – not the piggybacked output packet by truncating the packet. This reduces the amount of data that needs to be mirrored. A consequence of this is that if a replication request or its response is dropped, the output packet will be lost. This is permitted by our linearizability-based correctness model: it is indistinguishable from the output packet being sent and dropped in the network. The state updates must be retransmitted, however, because subsequent packets processed by the switch may see the new version, and thus it must be durably recorded. We measure the overhead of request buffering in subsection 5.6.4.



**(a)** Stale state access after a switch recovers from link failure.



**(b)** Only one switch holds a lease (red key) on state at a time.

**Figure 5.7:** Consistent state access for multiple switches.

### 5.4.3 Lease-based State Ownership

What if multiple switches attempt to process packets for a particular flow at the same time, especially during failover or recovery? The protocol in subsection 5.4.2 will not be correct in this case, when there are concurrent accesses to the same state. Fig. 5.7a illustrates why. After Switch-1 has a link failure (but does not lose its state, which is *cnt*=2), packets are routed to an alternate, Switch-2. If Switch-1 recovers, a packet may read its old state, a violation of linearizability.

**Our approach.** RedPlane ensures that only one switch can process packets for a given flow at a time using *leases*, a classic mechanism for managing cached data in file systems [80] and replicated systems [111, 125]. Fig. 5.7b illustrates this. If a packet wants to access state, but the state is not available at the switch, it first requests a lease for the flow. The state store grants a lease for a specific time period (1 second in our prototype) only if no other switch holds an active lease on the same flow state. The lease time is renewed each time the switch sends a replication request for that flow; switches that frequently read but infrequently update state can send explicit lease renewal requests. Our prototype does so every 0.5 seconds.

### 5.4.4 Periodic Snapshot Replication

As described in §5.3, RedPlane offers bounded-inconsistency mode for write-centric applications that permit approximate results, *e.g.,* monitoring using sketches [149] or Bloom filters [160]. In this section, we describe how we realize it in the switch data plane.

For such applications, RedPlane replicates *snapshots* of state *asynchronously* and *periodically*. Every $T_{snap}$ seconds, a snapshot of the current state is sent to the state store, while output packets are released without waiting for replication to complete.

However, realizing this approach entirely in the data plane is challenging. While data structures often consist of multiple entries (*e.g.,* slots in sketches), the switch is architected, and the P4 language is designed, to allow access to a single entry per register array per packet. Also, building hardware that could atomically copy entire register arrays would be costly.

To address this challenge, we employ a *lazy snapshotting* approach. We maintain two copies of the data structure that are lazily synchronized with each other. These are interleaved in the switch's register arrays so that each array index contains two entries, one from each copy. Two metadata registers are used to indicate which entry at each index is the *active* copy. The first, a 1-bit flag, is toggled when a snapshot is taken. The second, a 1-bit register array, represents whether that index has been updated since the current snapshot started.

To take a snapshot, we flip the flag and read values from the now-inactive copy. Meanwhile, when packets arrive and update the array, one of two operations occur. The first packet to update an index synchronizes the two copies and then updates the active copy. Later packets simply update the active copy. This allows us to take a consistent snapshot of the entire structure while incoming packets continue to update it. Additional snapshots must wait for the current one to complete. We describe the pseudocode of our mechanism in **??**.

Replication is achieved using the switch ASIC's packet generator. We configure it to generate a batch of packets every $T_{snap}$ seconds. To replicate a data structure with $n$ entries, we generate a batch of $n$ packets, each with a unique ID $p_i$. The ID in each packet is used to address the $i$th entry in the data structure and copy its value into a RedPlane replication protocol header. Note that while RedPlane asynchronously replicates snapshots, it still guarantees successful replication with its sequencing and retransmission mechanisms.

### 5.4.5 Protocol Correctness

RedPlane's replication protocol provides per-flow linearizability defined in §5.3. Due to space constraints, we give only a brief sketch of the reasoning here. The lease protocol ensures that at most one switch is executing a program for a particular flow at a time. The sequencing, retransmission, and buffering protocol ensure that an output packet is never sent unless the corresponding state update has been recorded and acknowledged by the state store.

During non-failure periods, RedPlane provides per-flow linearizability because the single switch processing packets for a flow operates linearizably, but some output packets may be lost (due to dropped replication traffic with piggybacked messages). After a failover, the new switch receives a state version at least as new as the most recent output packet from the old switch. This satisfies the linearizability requirement that any packet sent after these output packets were observed follow it in the apparent serial order of execution. We also wrote a TLA+ specification of the linearizable mode to model-check the above property (**??**).

Our periodic snapshot replication guarantees that the system recovers to a consistent state from within a time bound $\epsilon$ (*i.e.,* bounded inconsistency) by tracking the time since the last successful replication; if the time bound is exceeded, an application-specific action may be taken (*e.g.,* dropping further packets or treating the switch as failed).

## 5.5 Implementation

Our prototype implementation is available in our repository [48].

**Data plane.** We implement RedPlane's data plane components in P4-16 [38] (≈1192 lines of code) and expose them as a library of P4 control blocks [38, §13], which form the RedPlane APIs that developers can use to make application state fault tolerant. We compile RedPlane-enabled applications to the Intel Tofino ASIC [46] with P4 Studio 9.1.1 [42]. We implement key functions such as lease request generation, lease management, sequence number generation, and request timeout management, using a series of match-action tables and register arrays. We evaluate the additional resource usage in subsection 5.6.4. As mentioned in subsection 5.4.2, we implement request buffering via the mirroring and truncation capabilities of the switch ASIC, which allows us to buffer only the replication protocol data and discard the original payload. We implement a basic sketch that supports lazy snapshotting; developers can modify it to implement similar data structures such as Bloom filters.

**Control plane.** We implement the switch control plane in Python and C++. Its main function is to initialize and migrate (if available) state for the data plane by processing corresponding responses forwarded by the data plane component.

**State Store.** Our contribution is in the fault tolerance protocol design and switch components. As such, our state store prototype is built based on readily available libraries and simple implementations. We implement RedPlane's state store in C++ for Linux servers. It uses Mellanox's kernel-bypass raw packet interface [22] for optimized I/O performance. To ensure reliability in the presence of server failures, we implement chain replication [155] using a group of 3 servers located in different racks.

**Applications.** To demonstrate the applicability of RedPlane, we implement various applications in P4 described below. The simplified P4 code for NAT is available in **??**.
*(1) NAT:* The NAT implementation uses RedPlane to implement a fault-tolerant per-5-tuple address translation table and available port pool. Since the port pool is a shared by different flows, it is sharded across state store servers and managed by them. The state is updated when a TCP connection is established from an internal network.
*(2) Firewall:* The stateful firewall adds fault-tolerance to a per-5-tuple TCP connection state table using RedPlane. Its state is updated when a TCP connection is established from an internal network.
*(3) Load balancer:* The load balancer maintains a per-5-tuple server mapping table; we make it fault-tolerant using RedPlane. It also uses a server IP pool, which is shared state. When a new TCP connection is established from an external network, the state is updated.

*(4) EPC-SGW:* We also implement a simplified serving gateway (SGW) used in cellular networks, a mixed-read/write application. It maintains per-user tunnel endpoint ID state. The state is updated by signaling messages and read by data packets.

*(5) Heavy-hitter (HH) detection:* We implement a heavy-hitter detector using count-min sketches [67] as an example of write-centric applications; there are 3 sketches, each consisting of $64\times32$-bit slots indexed by a hash of the IP 5-tuple. We implement separate sketches per VLAN ID, assuming that the network operator wants to enforce different policies for each cloud tenant. Since sketches are an approximate data structure which can be replicated asynchronously, we use periodic snapshot replication.

*(6) Per-flow counter:* To demonstrate RedPlane's worst-case performance, this application counts packets forwarded for each IP 5-tuple. State is updated for every packet and synchronous replication must be used.

## 5.6 Evaluation

We evaluate RedPlane on a testbed consisting of six commodity switches (including two programmable ones) and servers (see **??**) using both real data center network packet traces and synthetic packet traces. Our key findings are:

- In failure-free operation, RedPlane adds no per-packet latency overhead for applications that are read-centric or replicate state asynchronously. For write-centric applications in linearizable mode, RedPlane incurs 8 $\mu s$ per-packet overhead (subsection 5.6.1).
- In failure-free operation, the throughput of read-centric applications is not degraded. For write-centric applications, the throughput is bottlenecked by state store performance in linearizable mode, but periodic snapshot replication reduces the overhead. Similarly, RedPlane incurs almost no bandwidth overhead for read-centric applications and small overhead for write-centric in bounded-inconsistency mode even at scale (subsection 5.6.2).
- After a switch failure, RedPlane-enabled applications access their correct state and recover end-to-end TCP throughput within a second (subsection 5.6.3).
- RedPlane provides these benefits with little resource overhead as it consumes $<$14% of ASIC resources (subsection 5.6.4).

**Testbed setup.** We build a three-layer network testbed (shown in **??**). The aggregation layer has two 64-port Arista 7170 Tofino-based programmable switches [55] running stateful applications written in P4. The core and ToR switches run 5-tuple-based ECMP routing to route packets to end hosts even when one aggregation switch fails. Each ToR switch has two servers connected, and four additional servers attached to the core switch emulate hosts outside the datacenter. The state store runs on one server in each rack. All servers are equipped with an Intel Xeon Silver 4114 CPU (40 logical cores), 48 GB DRAM, and a 100 Gbps Mellanox ConnectX-5 NIC, running Ubuntu 18.04 (kernel version 4.15.0). We repeat each experiment 100 times unless otherwise noted.

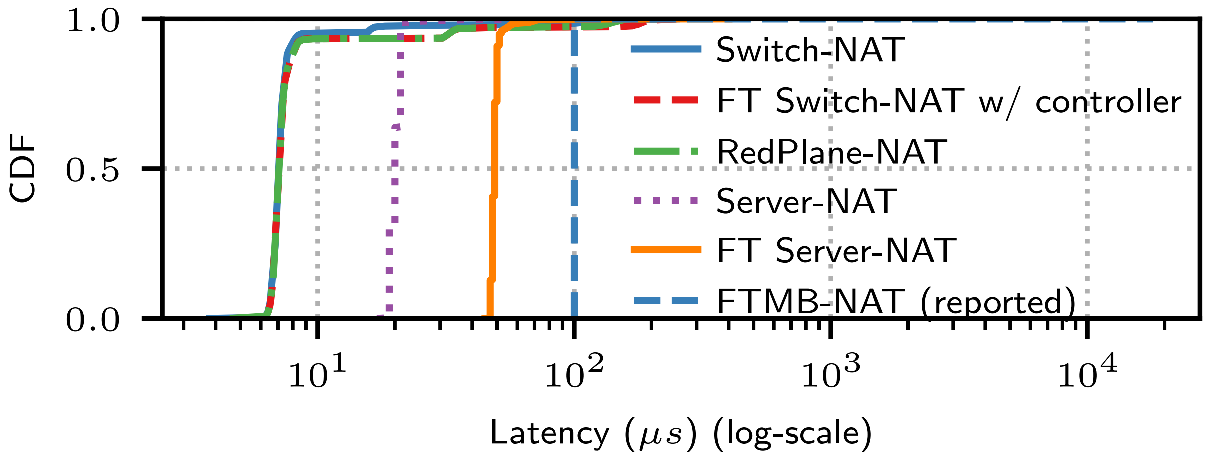


**Figure 5.8:** End-to-end RTT when RedPlane-NAT processes packets *vs.* other approaches.

### 5.6.1 Latency in Normal Operation

First, we evaluate the per-packet latency overhead introduced by RedPlane under failure-free operation for the 5 applications in §5.5. To measure the processing latency, we have each application send packets back to a sender node and track the RTT of each packet. We replay publicly available packet traces from a real data center and enterprise network [19, 20] to generate 100,000 packets and measure the processing latency of each packet. The packet sizes vary (64–1500 bytes) in the real traces. To evaluate EPC-SGW, we inject a signaling packet for every 17 data packets, following statistics used in previous studies [123, 133].

**Overhead of RedPlane.** As an exemplar application, we evaluate the per-packet latency for a NAT in RedPlane[7] and compare it with baseline implementations: (1) NAT written in P4 without fault-tolerance (Switch-NAT), (2) NAT written in P4 with controller based fault-tolerance (Switch-NAT w/ an external controller)[8] (3) NAT implemented on a CPU server without fault-tolerance (Server-NAT), (4) NAT implemented on a server with fault-tolerance (FT Server-NAT), and (5) FTMB-NAT which uses rollback-recovery for server-based middleboxes [145].[9] For switch-NAT w/ controller, RedPlane-NAT, and server-NAT, we enable chain replication for the controller, state store, and NAT instances, respectively.

Fig. 5.8 shows the CDF of the per-packet latency distribution. Compared to Switch-NAT, which is expected to have the lowest latency, RedPlane-NAT shows the same 50th and 90th percentile latency (7 $\mu s$ and 8 $\mu s$, respectively), meaning that there is no overhead. This is because for NATs, packets except for the first packet of each flow only require state (*i.e.,* address translation table) to be read. Both Switch-NAT and RedPlane-NAT show a high 99th percentile latency (110 $\mu s$ and 142 $\mu s$, respectively), mainly due to the overhead introduced by our control plane implementation; in Switch-NAT, the first packet of every flow is forwarded to the switch

[7]We choose NAT to compare results with those reported in prior work [145].

[8]We implement a simple external controller to emulate SDN controller-based approaches (*e.g.,* Morpheus [142] and Ravana [95]), which communicates with the switch control plane via a 1 Gbps management channel.

[9]We use the latency reported in the original FTMB paper [145] since we were not able to get its full implementation.

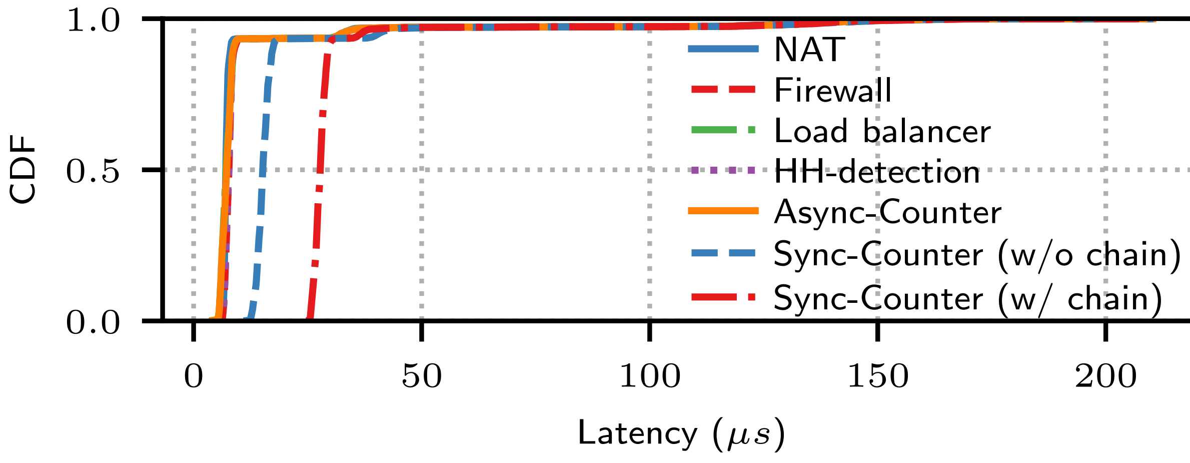


**Figure 5.9:** End-to-end RTT for RedPlane-enabled applications. All applications have chain replication enabled for the state store. For Sync-Counter, we also show its overhead without chain replication.

control plane to create and insert a new entry to the translation table. RedPlane-NAT has additional overhead since it needs to request a lease from the state store before updating state. Switch-NAT with the external controller incurs higher 99th percentile latency (185 $\mu s$) due to the communication overhead between the switch control plane and the controller and between controller instances (for chain replication) over the slower management network. Server-based versions (FT Server-NAT and FTMB-NAT) have 7–14$\times$ higher median latency compared to the switch-based approaches, as packets need to traverse additional hops in the network and they have inherent performance limitations.

**Impact on different applications.** Next, we evaluate the per-packet processing latency overhead of different RedPlane-enabled applications. As shown in Fig. 5.9, RedPlane-enabled NAT, firewall, load balancer, EPC-SGW, and heavy-hitter (HH) detection, all have the same 8 $\mu s$ median latency, identical to that without fault-tolerance. The NAT, firewall, and load balancer are read-centric and update state only when a new flow is created; EPC-SGW is mixed-read/write, and updates state on signaling packets whose frequency is 5% of data packets. HH detection, although it is write-centric, performs periodic state replication asynchronously, so it does not affect the latency. On the other hand, since Sync-Counter updates state and replicates updates *synchronously* for every packet, it adds an additional latency of 20 $\mu s$ to every packet. 12 $\mu s$ of this overhead is due to the 3-way chain replication used to tolerate state store server failures.

### 5.6.2 Bandwidth Overheads

To evaluate network bandwidth overheads, we inject 64-byte packets from three traffic generation servers at≈207.6 Mpps[10] which is the maximum rate that our traffic generator can achieve.

[10]Each server generates packets at ≈69.2 Mpps.

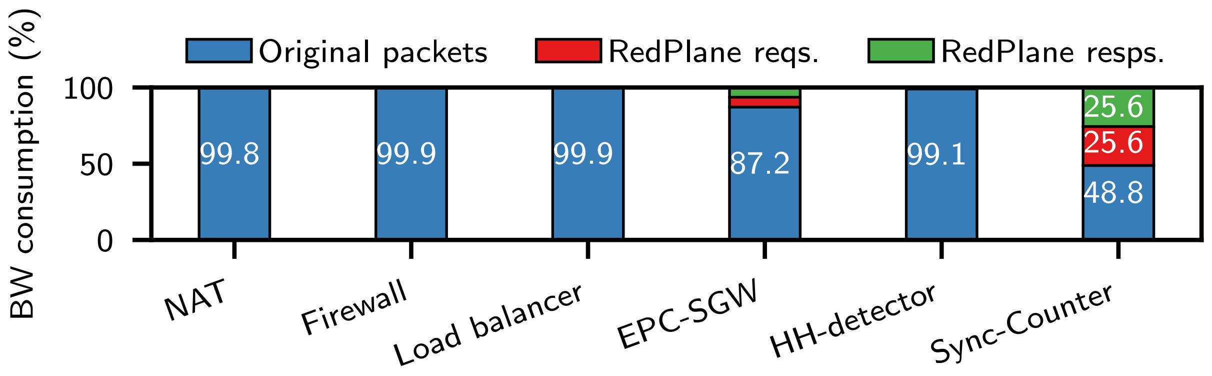


**Figure 5.10:** RedPlane replication bandwidth overhead.

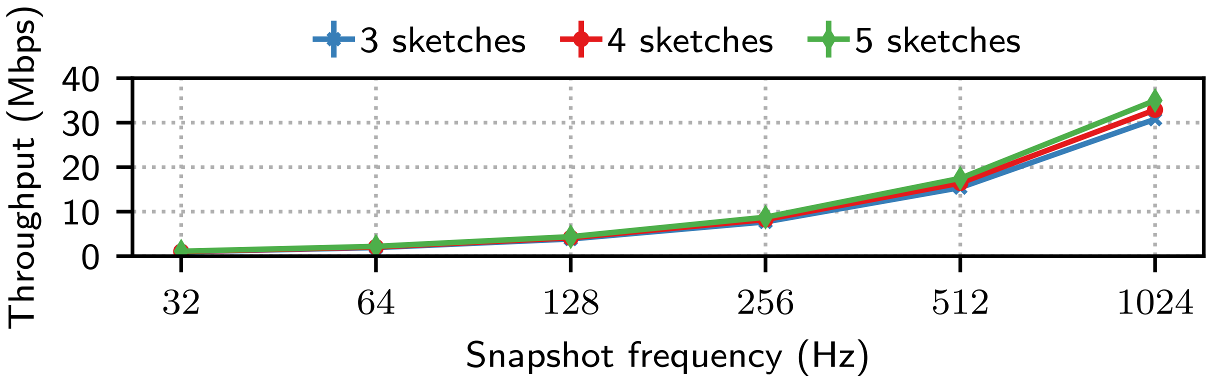


**Figure 5.11:** Impact of the frequency of snapshotting on RedPlane-enabled HH-detector.

**Additional bandwidth consumed.** In this experiment, we instrument each application to count the number of bytes it sends and receives, including both original packets and protocol message packets. Fig. 5.10 shows the ratio of bandwidth used for RedPlane messages to the total traffic. For read-centric applications including NAT, firewall, load balancer, we see that there is almost no bandwidth overhead since RedPlane generates protocol messages only for the first packet of each flow. For EPC-SGW, RedPlane incurs 12.8% overhead since it generates protocol messages for signaling packets, and some of data packets are buffered through the network as described in subsection 5.4.1. For HH-detector, which asynchronously replicates a snapshot of state for every 1 ms, RedPlane incurs negligible overhead. We also measure the absolute bandwidth overhead for different snapshot frequencies and number of sketches as shown in Fig. 5.11. For a 1 ms period, it consumes 34.16 Mbps (13.8%). Even with 5 sketches, this is lower than the bandwidth overhead for Sync-Counter (51.2%) because in the latter case RedPlane requests and responses contain both headers and original payload. This result implies that in an extreme case where an application replicates state updates synchronously for every packet, achieving fault-tolerance is expensive. We also analyze the bandwidth overhead at scale (*i.e.,* a topology with more RedPlane switches) for all 6 applications using our analytical model-based simulation, and the result is consistent with Fig. 5.10 in terms of the percentage overhead.

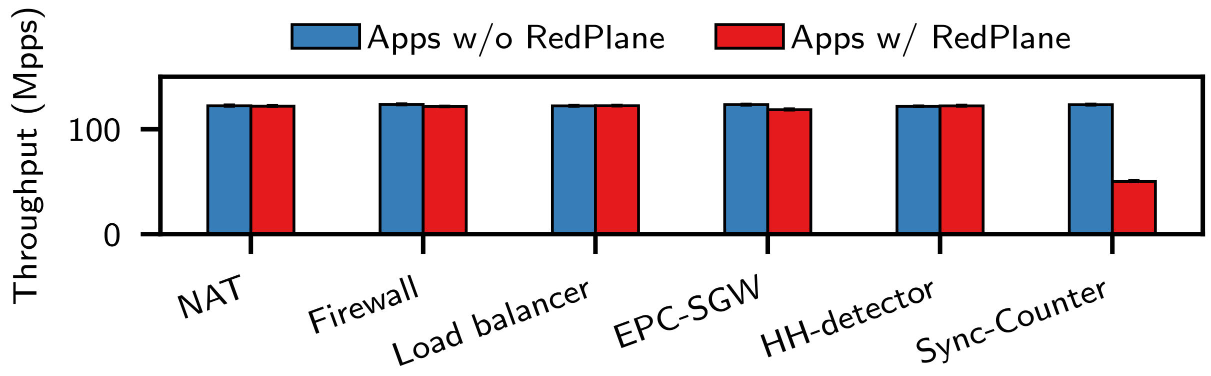


**Figure 5.12:** Impact of RedPlane on data plane throughput of RedPlane-enabled applications.

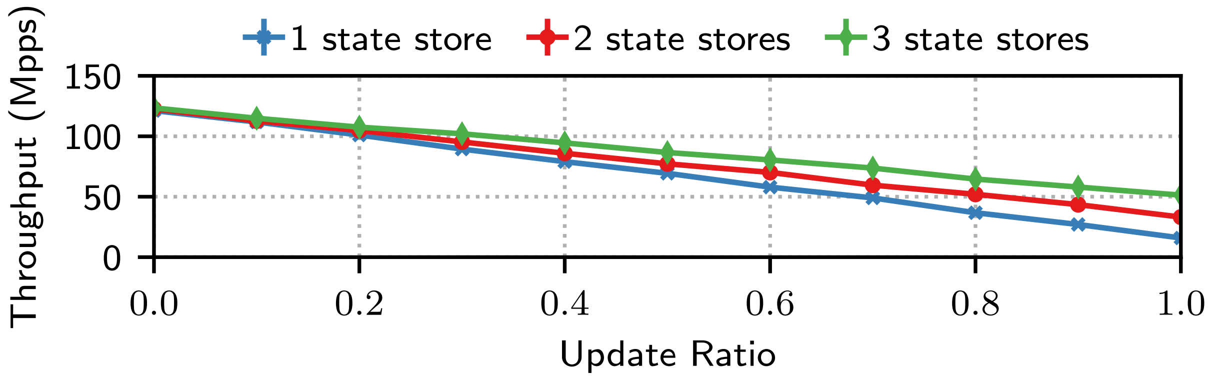


**Figure 5.13:** Impact of update ratio on data plane throughput of the RedPlane-enabled key-value store.

**Throughput impact on applications.** In this experiment, we measure the throughput of RedPlane-enabled applications and compare it with the same applications without fault tolerance. We send 64-byte packets from three servers, one from each rack, to one of servers attached to the core switch at ≈207.6 Mpps. In our testbed, the link between an aggregation and a core switch becomes the bottleneck, and we observe that the maximum forwarding rate the aggregation switch can achieve is around 122.5 Mpps. Fig. 5.12 shows the median throughput of each application with and without RedPlane. Obviously, applications achieve the maximum throughput without RedPlane. With RedPlane, read-centric (NAT, firewall, and load balancer) applications and applications that replicate state updates asynchronously (HH-detector) can achieve the same throughput as their non fault-tolerant counterparts. The RedPlane-enabled EPC-SGW achieves a slightly lower throughput than that of its counterpart, mainly due to some data packets buffered through the network during the replication. The throughput of Sync-Counter becomes nearly half that of its counterpart: we find that it is bottlenecked by the performance of the state store. This suggests that applying a strict consistency mode degrades the throughput of write-centric applications as they are also affected by the performance of the state store.

**Varying update ratios.** While most of existing in-switch applications are read-centric or perform asynchronous replication, incurring little overhead, it is important to understand the max-

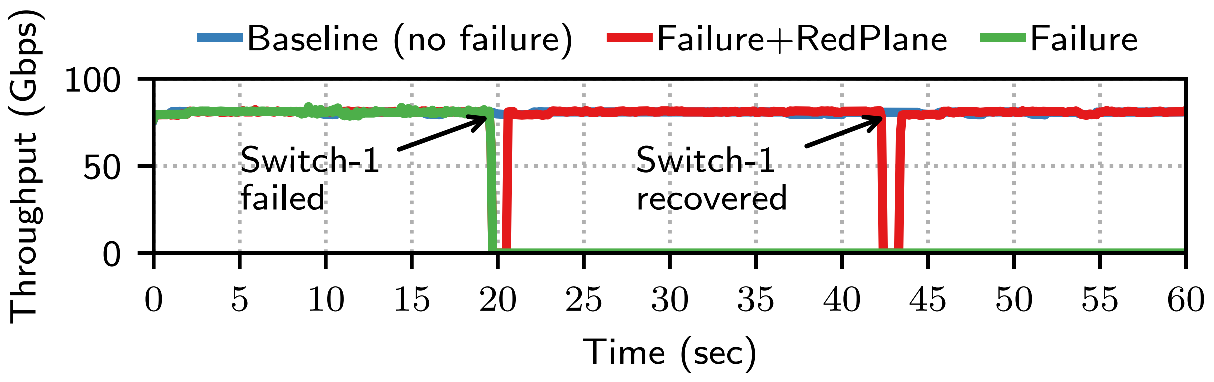


**Figure 5.14:** End-to-end throughput changes during failover and recovery with and without RedPlane.

imum throughput of applications characterized by different read/write (*i.e.,* update) ratios. For this experiment, we write a simple in-switch key-value store in P4 with RedPlane and generate packets consisting of custom header fields that indicate an operation (read or update), a key, and a value (for updates). We use the same setup as the previous experiment and let each server generate packets based on a predefined update ratio with uniformly distributed random keys. Fig. 5.13 shows that as the update ratio increases, the throughput degradation depends on the number of state store servers; by adding more servers, we can achieve higher throughput.

### 5.6.3 Failover and Recovery

Next, we measure how fast the end-to-end performance can be recovered by RedPlane in the presence of switch failure and recovery. We run `iperf` [41] to measure between two servers, attached to a core switch and a ToR switch respectively. All traffic passes through a NAT running on the programmable switches. We compare changes in TCP throughput when (1) there is no failure (Baseline), (2) one switch fails without RedPlane (Failure), (3) one switch fails while using RedPlane (Failure+RedPlane).

Fig. 5.14 shows the results. In a network without RedPlane, when Switch-1 fails, packets are rerouted to another switch and dropped, breaking the TCP connections. In contrast, RedPlane-enabled NAT successfully maintains high throughput when the switch fails and recovers after short disruptions (0.9 and 1.0 seconds). This recovery time is affected both by the core switch's failure detection/rerouting time and RedPlane's lease period (set to 1 second here). Control plane and state store optimizations could further reduce this.

### 5.6.4 RedPlane Switch ASIC Resource Usage

**Packet buffer usage.** In this experiment, we evaluate the overhead of our request buffering mechanism (subsection 5.4.2). Since RedPlane buffers a replication request until it receives a reply corresponding to the request from the state store, it consumes some amount of the switch packet buffer. Since there is no precise way of measuring the buffer usage in real-time, we instead

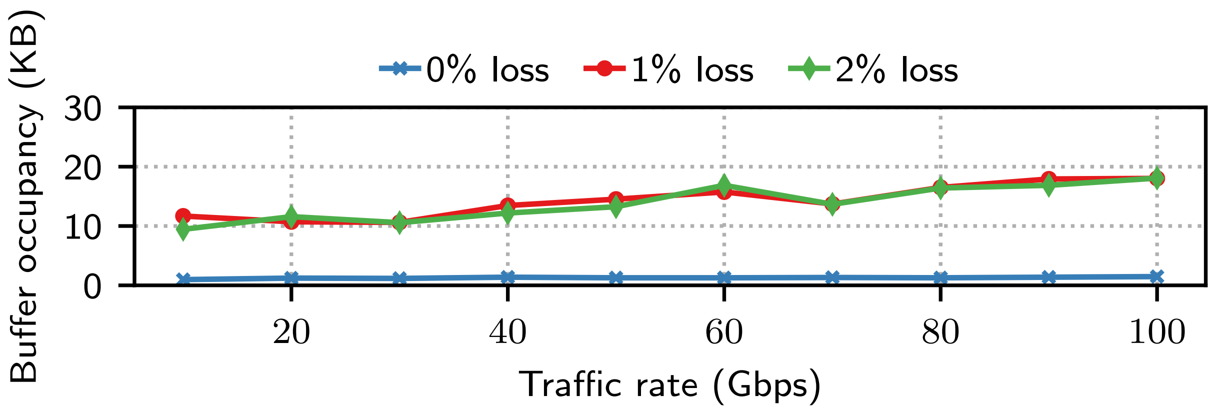


**Figure 5.15:** Switch packet buffer occupancy due to request buffering.

use the queue depth information provided by the switch ASIC to estimate the upper bound of the buffer occupancy.[11] Specifically, we assume a write-centric application where every incoming packet issues a request (*i.e.,* the most demanding scenario). And we let each request packet record its queue depth information to a P4 register in the data plane and read it from the control plane for every second and take the maximum value. We generate packets from a traffic generation server while varying the traffic rate and the request loss rate.[12] Fig. 5.15 shows the result. When there is no request loss, the buffer occupancy is less than 1.5 KB even at 100 Gbps traffic rate. As we increase the request loss rate, the buffer usage also grows; when the traffic rate is 100 Gbps and ≈2% of requests are lost, our buffering mechanism consumes at most 18 KB, which is acceptable for a given a few tens of MB of the packet buffer in the switch ASIC.

**Other ASIC resource usage.** We also measure the usage of other ASIC resources consumed by RedPlane data plane for 100K concurrent flows (using the Tofino-P4 compiler's output), expressed relative to each application's baseline usage. Ample resources remain: SRAM is the most used (13.2%), and all others are less than 10% (details in **??**). Scaling up concurrent flows would increase only SRAM usage, as it stores per-flow information (lease expiration time, current sequence number, and last acknowledged sequence number).

## 5.7 Related Work

**In-switch applications.** Recent efforts have shown that offloading to programmable switches enhances performance. For example, offloading the sequencer [107], key-value cache [90, 117], and coordination service [91] improves the performance of distributed systems. However, these applications can lose their state due to switch failures. RedPlane can help make them fault-tolerant or simplify their designs.

[11]It is a per-packet queue depth measured when a packet is dequeued from the buffer, and the Tofino ASIC provides this information as an intrinsic metadata that can be accessed at the egress pipeline.
[12]We emulate the request loss by dropping requests at a certain probability at the switch.

**Fault-tolerance and state management for NFs.** Fault-tolerance for NFs or middleboxes has been addressed by prior systems like Pico [135] and FTMB [145]. When an NF instance fails, the state of the failed NF is recovered through checkpoint or rollback recovery on a new NF instance. These approaches cannot be applied directly to the switch data plane (subsection 5.1.2). Previous work on state management for stateful NFs uses local or remote storage to manage NF state [77, 136, 159]. However, these APIs target planned state migration rather than unplanned failures. Similar work (again, targeting planned migration) has also been proposed for router migration [98].

**External memory for switches.** Recent work shares our approach of using servers' memory as external storage for switch state [102], but towards a different goal: allow switches to handle state larger than their on-device memory. It does not address fault tolerance or multi-writer consistency.

**Switch-based reliability protocols.** Other recent work runs coordination protocols between switches to build reliable storage [69, 91]. Our goal is conceptually different – to replicate state for in-switch applications rather than provide a networked storage service – but uses some similar mechanisms, like network sequencing [107].

## 5.8 Summary

While many recent efforts have demonstrated the potential benefits of running datacenter functions on programmable switches, we argue that there is one critical missing piece in current designs, which is fault tolerance. To address this issue, in this paper, we present RedPlane, which provides a fault tolerant state store abstraction for in-switch applications. We formally define a linearizability-based correctness model for a replicated switch data plane state and build a practical replication protocol based on it. Our evaluation with various stateful applications on a real testbed shows that RedPlane can support fault-tolerance with minimal performance and resource overheads and enable end-to-end performance to quickly recover from switch failures.

# Chapter 6

# Conclusions

## 6.1 Lessons Learned

**Control plane APIs (Northbound APIs).**

**Lack of feedback control loop.**

**Runtime reconfigurability.**

**Data plane-only applications.**

## 6.2 Future Directions

**Language and compiler supports.**

**Capability.**

**Large-scale analysis.**

## 6.3 Closing Thoughts

# Bibliography

[1] AT&T Picks Barefoot Networks for Programmable Switches. https://www.sdxcentral.com/articles/news/att-picks-barefoot-networks-programmable-switches/2017/04/, .

[2] Azure VPN Gateway. https://docs.microsoft.com/en-us/azure/vpn-gateway/vpn-gateway-about-vpngateways, .

[3] In-network DDoS Detection. https://www.barefootnetworks.com/use-cases/in-nw-DDoS-detection/, .

[4] The CAIDA UCSD Anonymized Internet Traces. https://www.caida.org/data/passive/passive_dataset.xml, .

[5] The Software Switch Pipeline. https://doc.dpdk.org/guides/prog_guide/packet_framework.html#the-software-switch-swx-pipeline, .

[6] 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, .

[7] Fastclick. https://github.com/tbarbette/fastclick, .

[8] Gurobi - C++ API Overview. https://www.gurobi.com/documentation/9.1/refman/cpp_api_overview.html, .

[9] Compare Kemp LoadMaster, F5 Big-IP & Citrix Netscaler. https://kemptechnologies.com/compare-kemp-f5-big-ip-citrix-netscaler-hardware-load-balancers/, .

[10] Intel FPGA Programmable Acceleration Card N3000. https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/intel-fpga-pac-n3000/overview.html, .

[11] Agilio CX SmartNICs - Netronome. https://www.netronome.com/products/agilio-cx/, .

[12] NPL Specifications. https://nplang.org/npl/specifications/, .

[13] P4 DPDK backend. https://github.com/p4lang/p4c/tree/master/backends/dpdk, .

[14] p4c: a reference P4 compiler. https://github.com/p4lang/p4c, .

[15] Pensando DSC-25 Distributed Services Card. https://pensando.io/wp-content/uploads/2020/03/Pensando-DSC-25-Product-Brief.pdf, .

[16] Recommended network configuration examples for roce deployment. https://community.mellanox.com/s/article/recommended-network-configuration-examples-for-roce-deployment, .

[17] P4-SDNet User Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1252-p4-sdnet.pdf, .

[18] Alveo U25 SmartNIC Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u25.html, .

[19] 2009-M57-Patents packet trace. http://downloads.digitalcorpora.org/corpora/scenarios/2009-m57-patents/net/, 2009.

[20] Data Set for IMC 2010 Data Center Measurement. http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html, 2010.

[21] pktgen-dpdk: Traffic generator powered by DPDK. https://git.dpdk.org/apps/pktgen-dpdk/, 2011.

[22] RDMA Aware Networks Programming User Manual. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf, 2011.

[23] 802.1qbb – priority-based flow control. https://1.ieee802.org/dcb/802-1qbb/, 2011.

[24] A signaling storm is gathering - is your packet core ready? https://www.nokia.com/blog/a-signaling-storm-is-gathering-is-your-packet-core-ready/, 2012.

[25] vEPC Acceleration Using Agilio SmartNICs. https://www.netronome.com/media/documents/SB_vEPC.pdf, 2017.

[26] Barefoot Networks Unveils Tofino 2, the Next Generation of the World's First Fully P4-Programmable Network Switch ASICs. https://www.barefootnetworks.com/press-releases/barefoot-networks-unveils-tofino-2-the-next-generation-of-the-worlds-f
., 2018.

[27] Advanced network telemetry. https://www.barefootnetworks.com/use-cases/ad-telemetry/, 2018.

[28] iperf3. http://software.es.net/iperf/, 2018.

[29] BCM88690–10 Tb/s StrataDNX Jericho2 Ethernet Switch Series. https://www.broadcom.com/products/ethernet-connectivity/switching/stratadnx/bcm88690, 2018.

[30] Offloading vnfs to programmable switches using p4. https://wiki.onosproject.org/download/attachments/12420314/p4-vnf-offloading-ons2018.pdf, 2018.

[31] Perftest package. https://github.com/linux-rdma/perftest, 2018.

[32] Cavium xpliant ethernet switches. https://www.cavium.com/xpliant-ethernet-switch-product-family.html, 2018.

[33] Amazon ec2 f1 instances. https://aws.amazon.com/ec2/instance-types/f1/, 2019.

[34] Cisco Visual Networking Index. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-738429.html, 2019.

[35] Mellanox innova-2 flex open programmable smartnic. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf, 2019.

[36] Liquidio ii 10/25gbe adapter family. https://www.marvell.com/ethernet-adapters-and-controllers/liquidio-smart-nics/liquidio-ii-smart-nics/index.jsp, 2019.

[37] Netronome agilio smartnics. https://www.netronome.com/products/smartnic/overview/, 2019.

[38] $P4_{16}$ Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.2.0.html, 2019.

[39] EdgeCore Wedge 100BF-32X. https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=181&id=335, 2019.

[40] The Evolved Packet Core. https://www.3gpp.org/technologies/keywords-acronyms/100-the-evolved-packet-core, 2020.

[41] iperf(1) - linux man page. https://linux.die.net/man/1/iperf, 2020.

[42] Barefoot P4 Studio. https://www.barefootnetworks.com/products/brief-p4-studio/, 2020.

[43] Redis. https://redis.io/, 2020.

[44] Cisco Silicon One Q200 and Q200L Processors Data Sheet. https://www.cisco.com/c/en/us/solutions/collateral/silicon-one/datasheet-c78-744312.html, 2020.

[45] The TLA+ Home Page. https://lamport.azurewebsites.net/tla/tla.html, 2020.

[46] Intel Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html, 2020.

[47] Trident4 / BCM56880 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series, 2020.

[48] Redplane public repository. https://github.com/daehyeok-kim/redplane-public, 2021.

[49] 3GPP. 3GPP TS 23.007: Restoration procedures, 2012.

[50] D. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The switchware active network architecture. *Netwrk. Mag. of Global Internetwkg.*, 12(3):29–36, May 1998. ISSN 0890-8044.

[51] Thomas Anderson and Michael Dahlin. *Operating Systems: Principles and Practice*, volume 1. Recursive books.

[52] Alexey Andreyev. Introducing data center fabric, the next-generation facebook data center network. https://engineering.fb.com/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-cente 2014.

[53] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. Snap: Stateful network-wide abstractions for packet processing. In *ACM SIGCOMM*, 2016.

[54] Mina Tahmasbi Arashloo, Pavel Shirshov, Rohan Gandhi, Guohan Lu, Lihua Yuan, and Jennifer Rexford. A scalable vpn gateway for multi-tenant cloud services. *SIGCOMM Comput. Commun. Rev.*, 48(1):49–55, April 2018. ISSN 0146-4833.

[55] Arista Networks. Arista 7170 Series. https://www.arista.com/en/products/7170-series, 2020.

[56] Barefoot Networks. Tofino Switch Architecture Specification (accessible under NDA), 2017.

[57] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *ACM IMC*, 2010.

[58] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

[59] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[60] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *ACM SIGCOMM*, 2013.

[61] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4:

Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[62] K. L. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz. Directions in active networks. *IEEE Communications Magazine*, 36(10):72–78, Oct 1998.

[63] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *ACM SIGCOMM*.

[64] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using rdma and htm. In *EuroSys*, 2016.

[65] Cisco. Cisco Global Cloud Index: Forecast and Methodology, 2016–2021 White Paper.

[66] David Clark. The design philosophy of the darpa internet protocols. In *Symposium proceedings on Communications architectures and protocols*, pages 106–114, 1988.

[67] Graham Cormode and Marios Hadjieleftheriou. Finding frequent items in data streams. *Proceedings of the VLDB Endowment*, 1(2), 2008.

[68] 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. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *USENIX NSDI*, 2018.

[69] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé. P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking*, 2020.

[70] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *USENIX NSDI*, 2014.

[71] 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 *ACM SOSP*, 2015.

[72] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *USENIX NSDI*, 2015.

[73] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *ACM SOCC*, 2011.

[74] 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 *USENIX NSDI*, 2018.

[75] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. In *ACM SIGCOMM*, 2014.

[76] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *ACM SIGCOMM*, 2020.

[77] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. In *ACM SIGCOMM*, 2014.

[78] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SOSP*, 2003.

[79] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *ACM SIGCOMM*, 2011.

[80] Cary Gray and David Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review*, 23(5), 1989.

[81] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM*, 2009.

[82] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *ACM SIGCOMM*, 2016.

[83] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *ACM SIGCOMM*, 2018.

[84] David Hancock and Jacobus Van der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. In *ACM CoNEXT*.

[85] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. A survey on data plane programming with p4: Fundamentals, advances, and applied research.

[86] J. Heinanen and R. Guerin. A two rate three color marker. RFC 2698, RFC Editor, September 1999.

[87] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), 1990.

[88] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. The p4-¿ netfpga workflow for line-rate packet processing. In *ACM/SIGDA FPGA*.

[89] Infiniband Trace Association. Supplement to InfiniBand architecture specification volume 1 release 1.2.1 annex A16: RDMA over converged ethernet (RoCE), 2010.

[90] 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 *ACM SOSP*, 2017.

[91] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *USENIX NSDI*, 2018.

[92] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless network functions: Breaking the tight coupling of state and processing. In *USENIX NSDI*, 2017.

[93] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In *ACM SIGCOMM*, 2014.

[94] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, 1997.

[95] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. Ravana: Controller fault-tolerance in software-defined networking. In *ACM SOSR*, 2015.

[96] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *ACM SOSR*, 2016.

[97] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *ACM SOSR*, 2016.

[98] Eric Keller, Jennifer Rexford, and Jacobus E van der Merwe. Seamless bgp migration with router grafting. In *USENIX NSDI*, 2010.

[99] Elie F Kfoury, Jorge Crichigno, and Elias Bou-Harb. An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends.

[100] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: Group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *ACM SIGCOMM*, 2018.

[101] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan. Generic external memory for switch data planes. In *ACM HotNets*, 2018.

[102] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *ACM SIGCOMM*, 2020.

[103] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. Redplane: Enabling fault-tolerant stateful in-switch applications. In *ACM SIGCOMM*, 2021.

[104] Petr Lapukhov, Ariff Premji, and Jon Mitchell. Use of bgp for routing in large-scale data centers. *Internet Requests for Comments RFC Editor RFC*, 7938, 2016.

[105] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John K. Ousterhout. Implementing linearizability at large scale and low latency. In *ACM SOSP*, 2015.

[106] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *ACM SIGCOMM*, 2016.

[107] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say no to paxos overhead: Replacing consensus with network ordering. In *USENIX OSDI*, 2016.

[108] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *ACM SOSP*, 2017.

[109] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan RK Ports. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In *USENIX OSDI*, 2020.

[110] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *USENIX NSDI*, 2014.

[111] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA, July 2012.

[112] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *ACM SOSP*, 2017.

[113] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. In *ACM ASPLOS*, 2017.

[114] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *USENIX NSDI*, 2013.

[115] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric ddos attacks with programmable switches. In *USENIX Security*.

[116] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *ACM SIGCOMM*, 2016.

[117] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *USENIX FAST*, 2019.

[118] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an rdma-enabled distributed persistent memory file system. In *USENIX ATC*, 2017.

[119] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. A large scale study of data center network reliability. In *ACM IMC*, 2018.

[120] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *ACM SIGCOMM*, 2017.

[121] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *USENIX ATC*, 2013.

[122] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *ACM SIGCOMM*, 2015.

[123] Ali Mohammadkhan, KK Ramakrishnan, Ashok Sunder Rajan, and Christian Maciocco. Considerations for re-designing the cellular infrastructure exploiting software-based networks. In *IEEE ICNP*, 2016.

[124] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *ACM SIGCOMM*, 2017.

[125] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, 2014.

[126] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In *ACM SOSP*, 2011.

[127] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[128] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *European Symposium on Algorithms*, 2001.

[129] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *ACM SIGCOMM*, 2021.

[130] 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. Ananta: Cloud scale load balancing. In *ACM SIGCOMM*, 2013.

[131] Larry Peterson, Tom Anderson, Sachin Katti, Nick McKeown, Guru Parulkar, Jennifer Rexford, Mahadev Satyanarayanan, Oguz Sunay, and Amin Vahdat. Democratizing the network edge. *ACM SIGCOMM Computer Communication Review*, 49(2), 2019.

[132] S. Pontarelli, P. Reviriego, and M. Mitzenmacher. Emoma: Exact match in one memory access. *IEEE Transactions on Knowledge and Data Engineering*, 30(11):2120–2133, Nov 2018.

[133] Zafar Ayyub Qazi, Melvin Walls, Aurojit Panda, Vyas Sekar, Sylvia Ratnasamy, and Scott Shenker. A high performance packet core for next generation cellular networks. In *ACM SIGCOMM*, 2017.

[134] Kun Qian, Sai Ma, Mao Miao, Jianyuan Lu, Tong Zhang, Peilong Wang, Chenghao Sun, and Fengyuan Ren. Flexgate: High-performance heterogeneous gateway in data centers. In *APNET*.

[135] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. Pico replication: A high availability framework for middleboxes. In *ACM SOCC*, 2013.

[136] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *USENIX NSDI*, 2013.

[137] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. 42(4):323–334.

[138] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM*, 2015.

[139] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4), 1984.

[140] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with in-network aggregation. In *USENIX NSDI*.

[141] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with in-network aggregation. 2021.

[142] Karla Saur, Joseph Collard, Nate Foster, Arjun Guha, Laurent Vanbever, and Michael Hicks. Safe and flexible controller upgrades for sdns. In *ACM SOSR*, 2016.

[143] Rinku Shah, Vikas Kumar, Mythili Vutukuru, and Purushottam Kulkarni. Turboepc: Leveraging dataplane programmability to accelerate the mobile packet core. In *ACM SOSR*, 2020.

[144] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *ACM SOCC*, 2017.

[145] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, et al. Rollback-recovery for middleboxes. In *ACM SIGCOMM*, 2015.

[146] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *ACM SIGCOMM*, 2015.

[147] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. Composing dataplane programs with $\mu$p4. In *ACM SIGCOMM*.

[148] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. Flightplan: Dataplane disaggregation and placement for p4 programs. In *USENIX NSDI*.

[149] L. Tang, Q. Huang, and P. P. C. Lee. Spreadsketch: Toward invertible and network-wide detection of superspreaders. In *IEEE INFOCOM 2020*, 2020.

[150] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *SIGCOMM Comput. Commun. Rev.*, 26(2):5–17, April 1996. ISSN 0146-4833.

[151] David G. Thaler and Chinya V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Trans. Netw.*, 6(1):1–14, February 1998. ISSN 1063-6692.

[152] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. Cheetah: Accelerating database queries with switch pruning. In *ACM SIGMOD*, 2020.

[153] William Tu, Fabian Ruffy, and Mihai Budiu. Linux network programming with p4. In *Linux Plumb. Conf.*

[154] J. E. van der Merwe, S. Rooney, L. Leslie, and S. Crosby. The tempest-a practical framework for network programmability. *IEEE Network*, 12(3):20–28, May 1998.

[155] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *USENIX OSDI*, 2004.

[156] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable high speed ip routing lookups. In *ACM SIGCOMM*, 1997.

[157] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4fpga: A rapid prototyping framework for p4. In *ACM SOSR*.

[158] T. Wolf and J. S. Turner. Design issues for high-performance active routers. *IEEE Journal on Selected Areas in Communications*, 19(3):404–409, March 2001.

[159] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *USENIX NSDI*, 2018.

[160] Jiarong Xing, Wenqing Wu, and Ang Chen. Ripple: A programmable, decentralized link-flooding defense against adaptive adversaries. In *USENIX Security*, 2021.

[161] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. Hyperv: A high performance hypervisor for virtualization of the programmable data plane. In *IEEE ICCCN*.

[162] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *ISOC NDSS*, 2020.

[163] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *ACM CoNEXT*.

[164] Dong Zhou. *Data Structure Engineering for High Performance Software Packet Processing*. PhD thesis, Carnegie Mellon University, 2019.

[165] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan RK Ports, Ion Stoica, and Xin Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *Proceedings of the VLDB Endowment*, 13(3), 2019.