ExoPlane: An Operating System for On-Rack Switch Resource Augmentation

Daehyeok Kim, Vyas Sekar, Srinivasan Seshan

Under submission – Please do not distribute

Abstract

Serving concurrent stateful applications on a programmable switch, while an inevitable trend in in-network computing, is challenging today due to limited switch's on-chip resources. In this paper, we argue that an on-rack switch resource augmentation architecture that augments a programmable switch with other programmable network hardware, such as smart NICs, on the same rack can be an affordable and incrementally scalable solution. To realize this vision, we design and implement ExoPlane, an operating system for on-rack switch resource augmentation to support multiple concurrent applications. Our evaluation with various P4 applications shows that ExoPlane can provide applications with low and predictable latency, scalable throughput, and fast failover while achieving these with small resource overheads and no or little modifications on applications.

1 Introduction

Recent advances in programmable switching ASICs [14, 17, 24] have become a key enabler for in-network computing. Programmable switches today are capable of implementing sophisticated network functions, such as NATs, firewalls, and load balancers [5, 43, 51] and accelerating distributed systems [48, 49, 57, 60].

This growing popularity of in-network computing is associated with two trends: (1) increasing number of in-network applications [38, 44] which can co-exist on a switch and (2) increasing demand of these apps to handle heavier workloads in terms of traffic volume and flows [9, 31]. Unfortunately, current switch resources are limited (*e.g.,* 10s MB of SRAM) and cannot keep up with the ever-increasing demands.

Given that cloud and cellular providers are increasingly deploying in-network computing in data centers [4, 51, 52, 54], in this paper, we explore an *on-rack switch resource augmentation* architecture that consists of a programmable switch and other data plane devices (*e.g.,* smart NICs [8, 12, 16, 22] and software switches running on servers [23, 61]) connected to the switch on the same rack. These external devices offer more resources to offload packet processing, albeit with some performance penalty. Perhaps more significantly, they offer a path to affordably and incrementally scale the effective capacity of a programmable network.

To effectively realize this vision of on-rack switch resource augmentation, we need the equivalent of an *operating system* to manage resources spread across multiple on-rack devices. To borrow from Anderson and Dahlin [26], we can draw a first-principles analogy to the three roles that any OS serves: (1) "glue" to provide a set of common services that facilitate the sharing of resources among applications; (2) an "illusionist" to provide an abstraction of physical hardware to simplify application design; and (3) "referee" for managing resources shared between multiple applications. While there is some recent work on mapping a single switch app to heterogeneous devices or to augment memory (*e.g.,* [35, 45, 47, 59]), these fundamentally do not tackle multiple concurrent applications or provide these capabilities.

However, on-rack switch resource augmentation creates new challenges different from those in traditional OSes with respect to these roles. In designing ExoPlane,[1] an OS for switch resource augmentation, we address these as follows:

- Runtime service (the glue): To avoid frequent inter-device communications during packet processing, we propose a *packet-pinning* operating model that guarantees that a packet is processed entirely on a single device.
- State abstraction (the illusionist): To enable correct stateful processing of packets even under dynamically changing workloads, we design a *two-phase state management* that places application states correctly on different devices as workload changes. We also design appropriate levels of consistency for different types of stateful objects that appear in applications.
- Resource allocation (the referee): To achieve performance and policy goals specified by developers and operators, we formulate and solve an optimal resource allocation problem that accommodates heterogeneity across apps and data plane device capabilities.

ExoPlane consists of two key components: the planner and runtime environment. The planner takes multiple P4 [30] apps written for a switch with no or little modifications and optimally allocates resources to each app based on inputs from an network operator and developers. It requires developers to add app-specific logic using our APIs *only if* the app contains a data-plane updatable object. Then, the ExoPlane runtime environment executes workloads across the switch and external devices by correctly managing state, balancing loads across devices, and handling device failures.

We implement the planner in C++, the data plane of the runtime environment in P4, and the control plane of the runtime environment in Python and C++. We evaluate it using

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

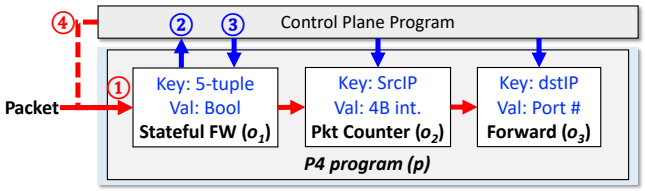


Figure 1: An abstract P4 application and runtime model. An application consists of multiple stateful objects (white boxes) and the control plane logic (blue arrows).

various P4 programs in our testbed consisting of a Tofino-based programmable switch [10] and servers equipped with Netronome Agilio CX smart NICs [8]. Our evaluations show that ExoPlane provides predictable latency (*e.g.,* ≈300 *ns* at the switch and 5.5 μs at an external device in steady-state) and scalable throughput with more external devices (*e.g.,* up to 394 Gbps, the maximum rate in our testbed). In case of an external device failure, ExoPlane can recover an end-to-end TCP throughput within 200 *ms*. ExoPlane achieves these with small control plane (a few tens MB) and switch ASIC resource overheads (less 4.5% of ASIC resources).

Ethics: This work does not raise any ethical issues.

2 Motivation and related work

In this section, we provide a primer on in-switch applications, motivate the need for resource augmentation, and discuss related work.

2.1 Stateful switch applications

Many in-switch apps are *stateful*; *i.e.,* state on the switch determines how to process packets. A typical program (p) contains one or more *stateful objects* (o_i), each of which can be represented as a P4 construct [30] such as a match-action table and a register.[2] Each object contains *state data* in the form of key-value pairs ((K_{o_i}, V_{o_i})) and *actions*. For example, a register in P4 consists of a data array and actions that access the array. Fig. 1 shows an example stateful P4 program (p) with three objects (o_1–o_3). Each object requires some amount of memory (*e.g.,* SRAM) for state data and compute resources (*e.g.,* stateful ALUs (SALUs)) for actions. The vendor-provided compiler (*e.g.,* Tofino compiler) allocates resources to each object using proprietary heuristics; if it cannot find a feasible allocation, the compilation will fail.

Once the program is successfully compiled and loaded to a pipeline, it can process incoming packets using its stateful objects; *e.g.,* the firewall app in Fig. 1 tracks active connections and drops unwanted packets from the Internet that do not belong to active connections. At runtime, the control plane logic can access the objects in the data plane (*e.g.,* inserting a new entry to the stateful 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. From the data plane, a packet only can look up an entry from the table. Registers can be read and updated by both the data and control plane. For example, in Fig. 1, 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.

[2]While our focus of this paper is on P4, other programming languages for programmable switches such as NPL [18] provide similar constructs.

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 [50] for remaining traffic classes.

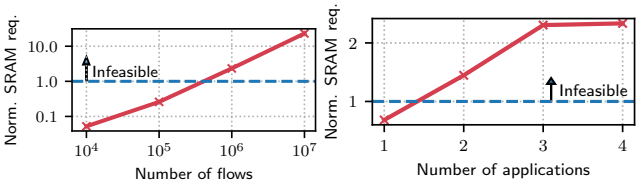
Table 1: P4 applications deployed in a gateway switch of the data center in our motivating scenario.

2.2 Motivation

As in-network computing becomes more popular, we observe two key trends that result in increasing demands on switch resources. First, the number of apps that operators need to run concurrently will likely increase [38, 44]. Second, the per-app workload size in terms of traffic volume and the number of flows also keeps growing [9, 31]. Unfortunately, switches cannot keep up with these ever-increasing workloads with their limited on-chip resources (*e.g.,* 10s MB of SRAM).

As a concrete example, suppose a cloud or cellular operator wants to deploy four applications in Table 1 on a network gateway router (*e.g.,* [52, 54]) processing traffic entering/leaving 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*). The *sketch-based monitor* collects statistics for the remaining traffic classes using an UnivMon sketch [50]. To see if/how these apps can coexist, we implement these applications in P4 or use source code from the original authors, compose them into a single P4 program using our merger (described in §6) and compile the result using the Tofino P4 compiler.

Unfortunately, we find that enabling these applications concurrently in a switch is infeasible for typical workloads, which requires to support at least 1M concurrent flows [33, 51, 52], as shown in Fig. 2. We consider two scenarios: (a) running all 4 apps but varying number of concurrent flows per-app and (b) fixing number of flows to 1M but adding apps incrementally. Here, we use SRAM requirements from each app, normalized (due to vendor NDAs) to the total amount of



(a) 4 apps with varying number of flows (log-scaled). **(b) Varying number of applications with 1 million flows.**

Figure 2: SRAM requirements (normalized to the total amount of SRAM on a switch) with varying workload size and number of apps. If the requirement > 1, it is infeasible.

SRAM on a switch, which is the bottleneck resource in our scenario. In Fig. 2a, we see that as the workload increases, it becomes infeasible to run all the apps. Similarly, in Fig. 2b, we see that the switch can support only a single application.[3]

2.3 Related work

Some recent efforts have proposed mechanisms to augment limited switch resources [35, 45, 59] or compose multiple apps [37, 58, 64, 65]. However, they do not provide capabilities to support multiple concurrent switch apps with demanding workloads.

Switch resource augmentation. TEA [45] provides a virtual table abstraction for a single switch app to access remote DRAM for a large lookup table. While TEA can be extended for an app with multiple tables, it requires multiple remote memory accesses (*i.e.,* multiple RTTs), affecting app's performance. Flightplan [59] takes a single app written with custom annotations and disaggregates it to multiple devices. Developers need to manually partition the app so that each device runs only a particular portion of the app. Lyra [35] proposes a custom language for writing a single app split across multiple heterogeneous switches. None of these considers multiple applications.

Language and framework for app composition. Some prior works attempt to support multiple data plane programs or modules in a single device [37, 58, 64, 65]. For example, virtualization approaches such as Hyper4 [37] and HyperV [64] allow composing multiple P4 programs with a constrained programming model. P4Visor [65] 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 program exceeds the available resources in the switch.

Server-based network functions. In the context of server-based NFs, previous work augments servers' resources by leveraging remote compute and storage resources, especially to manage NF state [36, 41, 42, 55, 63]. While they work well for server-based NFs, they do not directly applicable in our setting due to workload characteristics of switch apps and hardware constraints.

[3]In Fig. 2b, adding the 4^{th} app does not increase the SRAM usage much because the sketch's SRAM footprint is independent of the number of flows.

3 Overview

In this section, we make a case for on-rack switch resource augmentation and discuss challenges in realizing it.

3.1 Case for on-rack augmentation

Given the above trends and limitations of prior work, one can consider several candidate approaches; *e.g.,* optimizing apps to reduce resource footprint or adding more resources to the switch ASIC. While these are valid, they have limitations; *e.g.,* apps, even if optimized, may have high resource usage especially with changing workloads, and extending switch hardware is expensive.

We explore a different practical alternative, and envision an *on-rack switch resource augmentation* architecture consisting of a programmable switch connected to a few other programmable *external* data plane devices on the same rack. For example, we can allocate 2U of rack space, where a programmable switch is located, to install a server equipped with four 100 Gbps smart NICs connected to the switch. Since this architecture consumes a small amount of rack space and does not require any changes in other parts of the network, it provides a practical deployment model.

Note that this architecture aligns well with technology trends. First, there are many efforts to enable P4 frontends for many data plane devices, including NPU or FPGA-based smart NICs [7, 8, 39, 62] and software switches on x86 servers [21, 23, 61]. While these devices provide lower packet processing rate (up to a few 100s Gbps), compared to hardware switches (a few tens Tbps), they have more resources (*e.g.,* a few GB of DRAM *vs.* a few 10s MB of SRAM) to support demanding workloads. Second, and perhaps more importantly, we can *augment* the resources as needed by simply adding more devices. Taking these two factors into account suggests that if we could effectively realize such an architecture, it would offer a cost-efficient, incrementally extensible way to support the growing demands of multiple in-switch applications by adapting more and newer generation of external devices.

While the vision of on-rack switch resource augmentation is promising, to realize it in practice, we will need to effectively share compute and memory resources available on multiple devices across multiple applications. Drawing an analogy from traditional computing [26], ideally, we need an OS to provide an *infinite switch resource abstraction.* That is, 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. While some early efforts have leveraged resources on heterogeneous data plane devices for individual in-switch

apps [35, 45, 59], they do not provide the OS-like capabilities and abstractions we need for multiple concurrent apps.

3.2 Choice of operating model

A classical OS multiplexes multiple programs on the limited CPU/memory by choosing when, and what processes, to swap in/out. Our workload is a set of incoming packets mapped to various in-switch applications, and we need to choose a runtime model to multiplex the processing of these packets across apps and different devices.

Strawman models. The design space for the operating model can be defined by two dimensions: (1) Can an application run on multiple devices? and (2) Can an individual packet be processed on multiple devices?
To understand the trade-offs involved, let us consider two candidate options. In the an *app-pinning* model, an app is pinned to a single device, and a packet is entirely processed on that device. Since the packet is processed entirely on a single device without requiring additional logics, there is no additional processing latency due to inter-device rerouting and resource overhead. However, since the app can only run on that particular device, its throughput and available resources are limited. Alternatively, we can consider a *full-disaggregation* model where an app can run on multiple devices, and a packet also can be processed on multiple devices. Since an app can be placed any device, it has more available resources. However, depending on the availability of state, a packet needs to be routed between the switch and the external device multiple times. Such frequent inter-device routing increases packet processing latency and makes it unpredictable. This approach 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. Furthermore, this approach also consumes additional link and device bandwidth.

A case for packet-pinning. We adopt a *packet-pinning* model that pins a given packet to one device (*i.e.,* the switch or an external device) where it is completely processed while still providing flexibility of placing an app and its flows on any devices. First, it can avoid frequent per-packet inter-device routing with much lower complexity. Second, 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 an Internet backbone traffic [25]). Thus, if we could place popular flow state entries on the switch, it allows processing the majority of traffic (*i.e.,* packets) for the app entirely at the switch while the rest are processed at the external device.

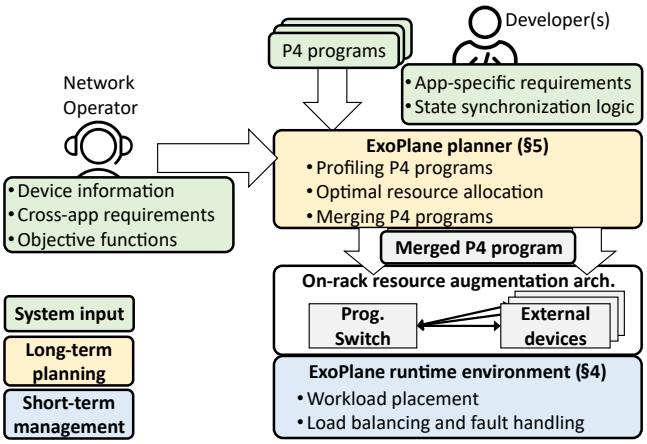


Figure 3: ExoPlane Overview: Green boxes represent inputs, yellow and blue box indicate key modules.

3.3 ExoPlane architecture

Our ExoPlane OS implements the packet-pinning operating model via two key components (Fig. 3):

- **The ExoPlane planner** takes inputs from developers and the network operator and allocates resources on the switch and external devices to each app.
- **The ExoPlane runtime environment** places workloads on devices, manages app states, and handles external device failures. In particular, at runtime, it tracks workload changes (*i.e.,* new flows arrive or flow popularity changes) and updates app's objects at the switch and external devices according to the changes.

ExoPlane assumes the following deployment capabilities: (1) A switch and external devices located on the same rack are programmable with the same set of P4-16 [11] constructs (*e.g.,* tables and registers), and we have blackbox access to vendor P4 compilers; (2) External devices are equipped with enough memory (*e.g.,* a few GB) to store all state for multiple applications;[4] (3) Each app handles a non-overlapping subset of traffic, which we call a traffic class with no dependency between different apps (*i.e.,* a given packet is processed by only a single app);[5] and (4) Data-plane updatable stateful objects maintain mergeable statistical data (*e.g.,* packet counter) that do not impact the control flow.

As illustrated in Fig. 3, to run apps on ExoPlane, developers provides P4 program codes and app-specific requirements (*e.g.,* affinity to the switch). Note that ExoPlane requires app modifications only if it contains data-plane updatable object

[4]We acknowledge that not every P4-programmable device supports all the features used by an switch app. According to our conversations with vendors, they plan to add such missing features, so this is not a fundamental limitation. Nonetheless, we design ExoPlane to adapt such devices as well by considering the app to device compatibility.
[5]If needed, we can apply prior offline preprocessing steps to convert overlapping subsets into an equivalent non-overlapping set [53].

whose copies can exist on multiple devices. The operator provides information on devices (*e.g.,* resources types), cross-app workload (*e.g.,* traffic distribution), and an objective function. The ExoPlane planner profiles the apps to get a compatibility to each device type and estimated resource footprint and performance, computes an optimal resource allocation, and generates a merged P4 program. It compiles the merged program using vendor-provided P4 compilers and loads the binaries to the switch and external devices. At runtime, the ExoPlane runtime environment executes the workload (*i.e.,* packets) across the switch and external devices.

3.4 Design challenges

C-1. Correctness under new flow arrivals and popularity changes. When the traffic workload changes, we need to update app's objects at the switch. We find that this can lead to incorrect packet processing due to the slow control plane operations. Also, when there are multiple copies of a data-plane updatable object across devices, each copy can be updated simultaneously. Unfortunately, is infeasible to adopt shared object synchronization schemes used in server-based systems [36, 55, 63] due to hardware constraints.

C-2. Handling multiple devices and device 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 apps. Given multiple apps, we have to share resources among them properly while considering per-app and cross-app objectives provided by an operator and developers.

4 ExoPlane runtime environment

In this section, we discuss the design of the ExoPlane runtime environment. For clarity, we start with a few simplifying assumptions— a single instance of external device, steady state traffic with no workload changes, no data plane-updatable state, no device failure, and a single application. We relax these assumptions subsequently.

4.1 Packet-pinning operating model

Recall from §3.2 that the packet-pinning model ensures that each packet is completely processed at a single device (*i.e.,* requires at most a single round-trip between the switch and an external device). Here, we load an app binary and all state entries on an external device with a subset of entries loaded along with the app on the switch. As mentioned in §3.1, an external device has a few GB of DRAM, which is enough to store all state entries (requiring up to a few hundred MB for a few million entries). If there is no entry for an incoming packet at the switch, the packet is routed to an external

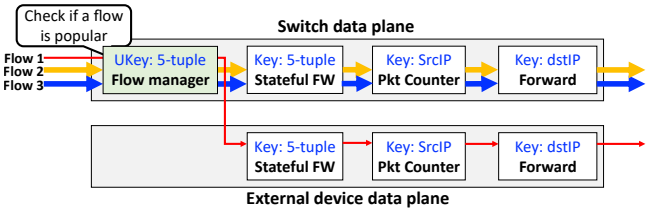


Figure 4: Our runtime environment processes most traffic at the switch and the rest at the external device in a run-to-completion manner. The green box is a per-app ExoPlane flow manager, and *UKey* indicates the app's union key.

device where the packet can be processed as all state entries needed to process the packet will be available.

However, naïvely implementing the packet-pinning model has two potential problems. 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, and it becomes overloaded, limiting the throughput. Second, since an entry miss can happen for an arbitrary object, per-object inter-device processing logic is needed to handle such cases. Such additional logic incurs switch data plane resource overheads.

To tackle this, we propose a *union-key based* state management that enables us to process a majority of traffic for an app at the switch and the remaining at the external device (Fig. 4). We define a union key type (UK) for an app as the union of key types of its constituent objects (*i.e.,* $UK = \cup_i K_{o_i}$). A flow is a set of packets with the same union key value. For example, in the figure, an IP 5-tuple is the union key type and packets with the same IP 5-tuple forms a flow.

Having defined the union key, we can use traffic workload characteristics to enable the switch to serve the majority of traffic for the app. Specifically, we build on the observation that the distribution of flow keys (including the union key) is skewed in typical networking workloads. As an example, we measure the distribution of IP 5-tuple which is the union key of our example app, by analyzing packet traces collected from an Internet backbone [25] and a university data center [27] (Fig. 15 in Appendix C illustrates the distributions). 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 most of the traffic at the switch by placing a few popular union keys (*e.g.,* 516 entries for 80% in the data center trace).

Based on this, we employ a per-app *ExoPlane flow manager* (the green box in Fig. 4 and denoted as o_{FM}) at the switch, which maintains a list of popular union keys for an app 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.,*

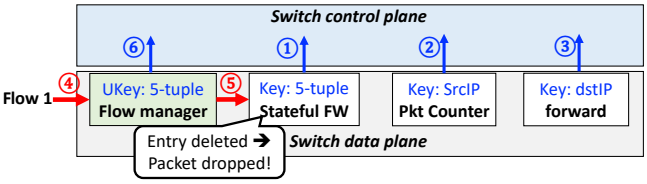


Figure 5: Incorrect state eviction: application's state has been removed while there is a packet being processed.

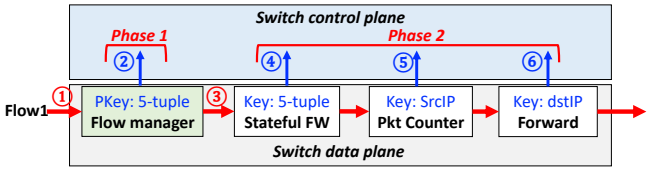


Figure 6: Correct two-phase state eviction.

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 allows to have low overhead as we do not need to have per-object inter-device processing. Taken together, our packet-pinning model and data plane design provides the following correctness property:

Invariant 1 (**Packet-pinning model**). *For each application, if the ExoPlane flow manager (o_{FM}) has the packet's union key ($UK(pkt)$), the constituent objects (o_i) must have entries ($K_{o_i}(pkt)$) for the packet. Formally,*

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

4.2 Handling workload changes

So far we assumed steady state—(1) no new flows and (2) no changes in flow popularity. Next, we discuss how we handle new flows and popularity churn.

Handling new flows. When a packet belonging to the new flow arrives at the switch, and if a miss occurs in the ExoPlane flow manager, it routes the packet to the external device. Note that there are two cases for the miss: (1) first packet of the new flow or (2) a packet of an existing flow for which the flow state is not at the switch. Since these two cases are indistinguishable from the view of ExoPlane flow manager, it always routes packets with misses to the external device. When a packet of the new flow arrives at the external device, it must first be processed by the app's control logic for handling new flow arrivals. In our example, the stateful FW table reports the packet to the control logic that inserts entries for the flow to three objects. At the same time, the control logic asks the control logic running on the switch to initialize the flow state at the switch data plane, which can succeed only when there is a space on every object. Depending on the app logic, the packet can be sent back to the data plane and processed with the new entries. If the flow state has been initialized both at the switch and the external device, the switch will process subsequent packets in the flow. Otherwise, the external device will process them.

Promoting popular flows. In practice, the popularity of flows can change and we need to promote and demote flow states as needed. Suppose we know which flow keys become popular (*i.e.,* their entries are currently not on the switch) and unpopular (*i.e.,* their entries are currently on the switch). We discuss how we track this in §6.

When promoting a new popular flow (*i.e.,* installing state at the switch), there are two possibilities: (1) there is spare space in the ExoPlane flow manager and app's other objects for new entries *vs.* (2) there is no room in the objects. For (1), we can simply insert new entries to the objects. For (2), however, we need to evict some unpopular flow to make a room. Doing so correctly is challenging. Fig. 5 illustrates why via a naïve mechanism can violate Invariant 1. 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 app objects (FW, Counter, and Forward) as well as the ExoPlane flow manager (blue arrows in Fig. 5). However, in the current switch architecture, a set of eviction operations (blue arrows) cannot be executed atomically. Thus, there could be cases where app's state entries have been removed while there are packets being processed in the data plane (⑤), violating Invariant 1. Even if eviction is correct, insertion can be incorrect. That is, during the time the switch control plane tries to insert entries for a flow, packets for the flow arrive and are looked up the ExoPlane flow manager. If the entry exists, the packet must be processed completely at the switch. However, since entries in other objects may not be available, the packet cannot be processed and will get dropped.

Two-phase state update. To address the issues, we adapt a *two-phase state update* mechanism, inspired by classical two-phase update or commit protocols [28, 56]. As illustrated in Fig. 6, when evicting entries for flow 1, in the first phase, the switch control plane evicts an entry from the ExoPlane flow manager. 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. 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 flow manager has been evicted are correctly processed in the switch. Note that when it evicts entries from the app's objects, it ensures that entries for other non-victim flows will remain. The insertion works similarly. To insert entries for a flow, in the first phase, the switch control plane inserts entries to the app's objects, and then in the second phase, it inserts an entry to the ExoPlane flow manager.

4.3 Synchronizing shared stateful objects

The previous discussion considers scenarios with no cross-flow objects that can be updated at runtime, which meant there was no need for objects on an external device and the switch to be synchronized. In practice, apps may have such objects; *e.g.,* per-SrcIP packet counter in our example is shared across flows. Next, we extend the basic ExoPlane protocol to handle such objects.

Consistency modes. P4 programs can have 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. Correspondingly, ExoPlane provides two levels of consistency. 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 a *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) in the data plane 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 *bounded-inconsistency* within a configurable time bound T_ϵ (*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 with a key (*e.g.,* a SrcIP), it updates (or inserts) all entries corresponding to the key existing at the external device and the switch.

Bounded-inconsistency for data-plane updatable objects is more challenging. Consider the per-SrcIP packet counter implemented using a register array in our example. Suppose that for a given SrcIP, there are two copies placed on the switch and the external device that can be updated simultaneously. To achieve bounded-inconsistency, the ExoPlane runtime needs to periodically *merge* values of the copies. Traditional techniques for state merging in server-based network functions (*e.g.,* [36, 55, 63]) are impractical in our context since they rely on buffering incoming packets and pausing processing while combining copies. This is expensive and even infeasible in the switch because packet rates are much higher, and we cannot buffer arbitrary packets.

Our approach for bounded-inconsistency. We devise a state synchronization protocol that achieves bounded inconsistency without needing packet buffering. We do so by combining capabilities of both the switch and external device's control and data plane. We use the control plane's

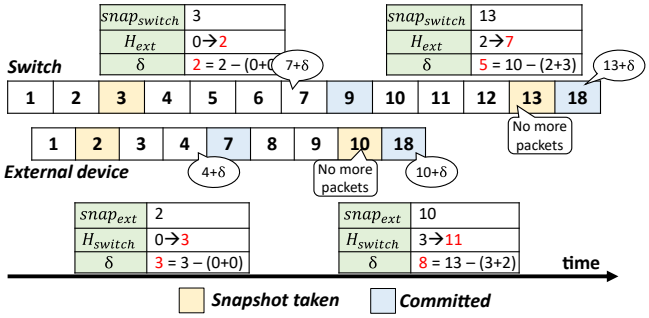


Figure 7: Our state synchronization protocol synchronizes two copies of an entry in the packet counter.

memory to track the history of periodic synchronizations while executing the merge operation in the data plane.

Let us revisit our counter example from Fig. 7. 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_ϵ seconds, the switch control plane initiates synchronization by sending its *Snap* and the H, and the external device's control plane replies it with its snapshot and history; e.g., 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 (δ) after the previous synchronization by subtracting two history values from the received snapshot value. This prevents a potential under or double-counting issue. Lastly, the control plane of both devices injects a special control packet containing δ 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 recovers. 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 to resume the synchronization from the state when it failed.

More generally, our protocol supports objects that can be expressed in a key-value structure. We provide a developer 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 δ and commit the update. For example, a Bloom filter [29] can be expressed as (Key: an integer, Value: {0, 1}) pairs with the binary OR as $\circ^+$ (no subtraction operator is needed). We provide a detailed pseudo-code in Appendix B.

4.4 Scaling to multiple devices

Thus far, we have assumed that there is a single external device. However, in practice, a single device instance may not provide enough processing capacity or resources. To use multiple devices, ExoPlane shards entries in objects across

the devices based on the union key. When an entry miss occurs at the ExoPlane flow manager, it routes a packet based on the union key to a specific external device that has state for the key. However, the skewness in the union key space (§4.1) could result in load imbalance across the devices (*i.e.,* a subset of devices can be overloaded). Fortuitously, the small fraction of popular entries we already have at the switch is helpful for load balancing. Prior analysis in storage systems 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), guarantees uniform load balancing across the servers regardless of the skew [34]. Thus, by placing $\geq O(N \log N)$ popular union keys at the switch, we can provide the cache effect for load balancing.

4.5 Handling external device failures

Application state loss due to failures can affect the performance or correctness of apps [46]. Specifically, we consider the failure model where an external device (or its hosting machine) fails or a network link between the switch and the device fails. To deal with state loss due to such failures, the ExoPlane runtime environment replicates each flow state to at least one additional external device when initiating flow entries, and when the primary device fails, it falls back to a replica. It does so by managing the logical to physical external device ID mapping at the switch, where the primary and replicas share the same logical ID. However, even if there is a replica, if we 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 (which is typically used for diagnosis), similar to previous work [45]. We configure the engine to generate a packet when ports go down. By processing the generated packet, the runtime environment updates the external device ID table in the data plane. Using the table, it can route subsequent packets to a replica.

5 ExoPlane planner

Next, we tackle the issue of sharing resources across multiple apps while meeting the performance objectives given by developers and operators. The resulting ExoPlane planner consists of a resource allocator and an application merger. The resource allocator finds an optimal resource allocation using inputs from the developers and the network operator. The application merger generates a merged P4 program based on the optimal allocation decision. Fig. 8 illustrates example inputs for an ensemble of applications.

Inputs. Developers provide a set of P4 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). Optionally, they can also specify a high,

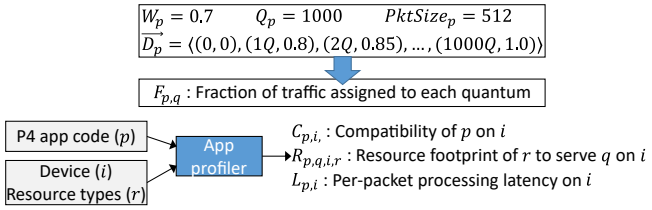


Figure 8: Example inputs for an application p.

medium or low switch affinity to the switch for each app. If the affinity of an app is set to high (or low), the app will run entirely at the switch (or at external devices). The network operator provides cross-app and per-app traffic information, which includes the fraction of all 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 union key space. While using a fraction of traffic served by *each key* provides the most fine-grained information, we find that it could make the search space for resource allocation too large. Instead, we use the distribution discretized into a larger quantum size denoted as Q_p. Based on D_p, we compute the estimated fraction of traffic served by each quantum q ($F_{p,q}$). The operator also provides resource information (r) for devices (i). This includes SRAM, TCAM, hash units, and SALUs for a Tofino-based switch and compute units, SRAM, and DRAM for NPU-based NICs.[6]

Profiler. Based on the inputs, we generate per-app profiles consisting of a resource footprint, per-packet processing latency, and compatibility matrix for each device type. The profiler estimates resource footprint of r for p serving q on i denoted as $R_{p,q,i,r}$. Since blackbox compilers determine the resource usage using proprietary heuristics, our preprocessor compiles p to determine $R_{p,q,i,r}$. For each q, it updates the size of each object specified in app code and compiles it using vendor-provided compilers. Then it extracts the resource usage from compiler outputs. If the compilation fails due to insufficient resources, it sets the resource usage to infinite. We use constants $Cap_{i,r}$ to represent the total amount of r available on i. The profiler also estimates a per-packet processing latency of p on i, $L_{p,i}$. Specifically, it instruments the switch to record two timestamps on a custom packet header field when a packet enters and leaves the rack. Then it injects $PktSize_p$-sized packets to the rack and estimates the latency based on the timestamps in returned packets.

Finally, some vendor-provided P4 compilers for external devices may not support certain switch hardware features or P4 constructs[7] used by apps. Because of this, if an app uses a feature that is not supported by an external device, the device cannot run the app. To consider the compatibility of the app on devices, our profiler generates a compatibility matrix

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

[7] *e.g.,* Packet recirculation and P4 registers

($C_{p,i}$) 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 vendor's compiler manual. For the second set, the profiler analyzes the app code to extract used features.

Resource Allocation. Given these inputs, we can formulate the problem of finding an optimal resource allocation satisfying per-app and cross-app requirements. In our formulation, we assume that the resource usage of multiple apps can be estimated by accumulating the resource usage of each app. We use binary decision variables $d_{p,q,i}$ to indicate whether q for p is assigned to i. There are four constraints types imposed:

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

$$\forall p, q, i : d_{p,q,i} \leq C_{p,i} \tag{2}$$

$$\forall i, r : \sum_p \sum_q d_{p,q,i} \times R_{p,q,i,r} \leq Cap_{i,r} \tag{3}$$

$$\forall p : lat_p = \sum_q \sum_i d_{p,q,i} \times F_{p,q} \times L_{p,i} \tag{4}$$

First, q must be assigned to a unique i (Eq. 1). Second, q can be assigned to i if and only if p is compatible with i (Eq. 2). Third, the amount of r consumed by q on i must be less than or equal to the total amount of r on i (Eq. 3). Last, the expected latency of p is the sum of per-packet processing latency of p on i weighted by $F_{p,q}$ (Eq. 4).

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:

$$\text{Minimize} \sum_p W_p \times lat_p \tag{5}$$

Other commonly used fairness metrics such as maximizing the minimum expected throughput can be used as well. By solving the ILP, the ExoPlane resource allocator finds an optimal assignment of q to i for p, and the size of each object and ExoPlane flow manager for p accordingly, which are used as input for the application merger, as we describe next.

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 our deployment model for multiple apps, described in §3.3. Our merger supports programs written in P4-16 [11] (Fig. 16 in Appendix C illustrates it works). First, for each app, the merger renames the main control block [11] to avoid naming conflicts between apps. Second, it specifies the size of each object (*e.g.,* number of entries in a table) based on the decision made by our resource allocator. Third, it inserts an ExoPlane flow manager. Finally, in the merged P4 code, it instantiates an instance of each app and inserts execution logic. The merged P4 code is compiled using the vendor-provided compiler and loaded on the switch and external devices. Sometimes, the compilation process fails due to its proprietary heuristics for resource allocation. If so, we repeat the process with a tighter resource constraint).

In summary, ExoPlane planner allocates resources across apps based on inputs from developers and the operator 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 frequently (*e.g.,* once every hour). While this module is not on the critical path, performance results are available in Appendix D.

6 Implementation

Data plane. The data plane components of the runtime environment implemented in P4-16 consists of: (1) ExoPlane flow manager implemented using a match-action table and (2) global logical to physical external device ID mapping implemented using a register array (on the switch).

Tracking flow popularity. We implement a flow popularity tracking mechanism on external devices using the count-min sketch [32] that tracks the frequently accessed flow keys. When it detects a new popular key, it reports the key to the external device's control plane that maintains a list of flow keys and corresponding entries, and they are reported to the switch control plane. On the switch, we enable the *aging* supported by the switch ASIC for the ExoPlane flow manager. If a certain key has not been accessed for a timeout period (T_{idle}), a callback function registered at the switch control plane is triggered, and it evicts the entry corresponding to the idle key. In our prototype, we set T_{idle} to 2 seconds.

Control plane. We implement the control plane components of the runtime environment in Python and C++. The main capability is to initializing new flow entries and promoting new popular flows' entries on the switch. On the switch side, we use Barefoot Runtime APIs to access the stateful object in the switch data plane. On the smart NIC side, we use Netronome Thrift APIs [13] 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.

Resource allocator. We implement the resource allocator in C++ based on the Gurobi C++ API [15] to encode and solve our resource allocation ILP.

Application profiler and merger. We extend the open-source P4 compiler [20] to analyze input P4 programs. Using its frontend, we extract information from each program including an entry size of each object. We implement the application merger in C++, which takes an IR generated by the compiler frontend, and produces a merged P4 program.

Supporting other hardware platforms. While our prototype uses a Tofino-based programmable switch and Netronome smart NICs, ExoPlane can be extended to other platforms. For example, ExoPlane can be applied to other types of programmable switches (*e.g.,* Nvidia Spectrum-2 ASICs [19])

Applications	States
Per-VM NAT	Per-flow address mapping for each VM.
Per-VM Stateful FW + Packet counter	Established TCP connection list.
Per-VM SYN proxy	Per-flow sequence number translation table.
NetCache [40]	Key-value store cache.

Table 2: Switch programs written in P4 used in the evaluation in addition to ones introduced in Table 1.

and FPGA or ASIC-based smart NICs (*e.g.,* Xilinx and Intel FPGA NICs [1, 7] as external devices for P4-programmable switches.

7 Evaluation

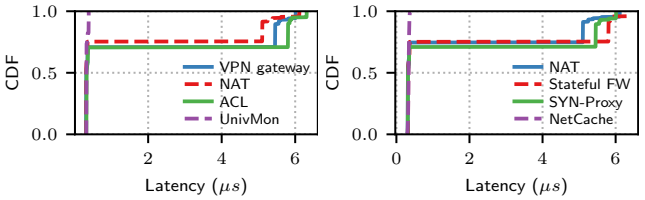
We evaluate ExoPlane on a testbed consisting of a programmable switch and servers equipped with a Netronome Agilio smart NIC using various workloads. Our key findings are:

- In steady-state, ExoPlane provides predictable per-packet latency (*e.g.,* 273–384 *ns* at the switch) and scalable throughput with more external devices while the app-pinning model achieves a limited throughput (§7.1).
- Even under dynamic workloads, ExoPlane can process packets with the correct state and sustain high throughput with multiple devices (§7.2).
- In case of an external device failure, ExoPlane can recover an end-to-end TCP throughput within 200 *ms* (§7.4).
- ExoPlane provides the above benefits with small control plane (*e.g.,* a few tens MB) and switch ASIC resource overheads (*e.g.,* less than 4.5% of ASIC resources) (§7.5).

Testbed setup. We build an on-rack resource augmentation architecture consisting of Wedge100BF-32X Tofino-based programmable switches [10] and 4 servers equipped with Netronome 40 Gbps smart NICs [8]. We use 4 additional servers with 100 Gbps regular NICs to generate traffic workloads. All servers are equipped with an Intel Xeon Silver 4110 CPU and 128 GB DRAM, running Ubuntu 18.04. We repeat each experiment 100 times unless otherwise noted.

Traffic workloads. We use packet traces collected from a real data center [2], the Internet backbone [25], 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 parameters (α=0.9, 0.95, 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. We replay the traces using DPDK-pktgen [3] or run iperf [6] for TCP workloads.

Deployment scenarios. We use two scenarios with multiple P4 apps: (1) at the data center gateway, 4 apps in Table 1 and (2) at the leaf of the network, 4 apps from Table 2. Given packet traces, we synthesize inputs for the resource allocator in ExoPlane planner (*e.g.,* per-app affinity and a flow key distribution). For example, we set the affinity level for the UnivMon [50] and NetCache [40] to high so that workloads for these apps are always processed at the switch.



(a) Data center gateway. **(b) Data center leaf.**

Figure 9: Per-packet processing latency distribution of apps concurrently running on ExoPlane in steady state.

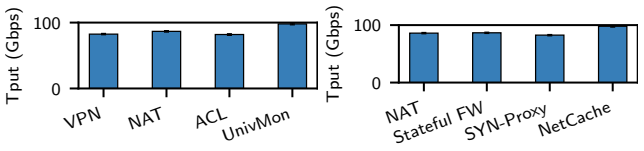
7.1 Performance in steady state

First, we evaluate the per-packet processing latency and throughput of apps running on ExoPlane in steady state (*i.e.,* no new flows, no changes in flow popularity, and no device failures). Here, we pre-populate popular flow entries at the switch and assume that the traffic is equally distributed across the apps (*i.e.,* $W_p = 0.25$ for all apps).

Packet processing latency. We define the packet processing latency as the time difference between when a packet first arrives at the switch from a sender and when it is sent to a receiver after processing. We instrument the P4 program running on the switch to record two timestamps (48-bits each) to our custom packet header fields of each packet so that the receiver can compute the processing latency for a packet. From the sender, we replay the backbone packet traces, each of which contains more than 6M flows.

Fig. 9 shows the CDF of the per-packet latency distribution for each app. For the apps that are assigned to the *high* affinity (UnivMon and NetCache), every packet is processed at the switch in 273–384 *ns*, depending on packet sizes. For other apps, the distributions vary depending on packet sizes and 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 gateway scenario (Fig. 9a), at the switch, the ACL processes ≈70% of its traffic whereas the NAT processes ≈75% of its traffic. At the external device, packets are processed in 5.1–6.1 μs depending on an app. 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 app throughput, we replay the synthetic packet trace that consists of 1500 B packets at line rate (98.6 Gbps in our testbed). We use four sender nodes, each of which generates traffic for each of four apps (*e.g.,* node 1 generates traffic for app 1). We start with a single external device to demonstrate the impact of the number of external devices to the throughput.



(a) Data center gateway. (b) Data center leaf.

Figure 10: Throughput of each app running on ExoPlane in steady-state with a single external device (Apps are running concurrently).

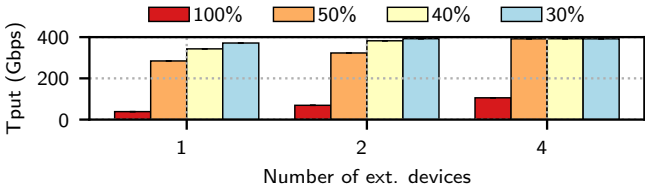


Figure 11: Scalable throughput with multiple devices with different fraction of traffic offloaded to external devices.

Ensemble of apps	App-pinning	ExoPlane
VPN	98.6 Gbps	98.6 Gbps
VPN+NAT	137.1 Gbps	197.2 Gbps
VPN+NAT+ACL	174.6 Gbps	295.6 Gbps
VPN+NAT+ACL+UnivMon	271.3 Gbps	394.1 Gbps

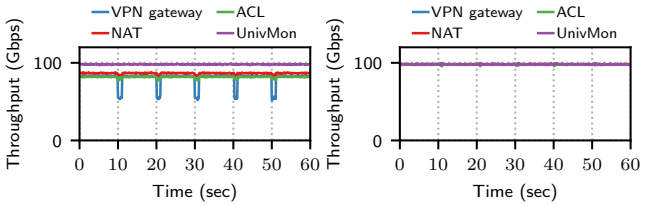
Table 3: Aggregate throughput of 4 apps running on the app-pinning model and ExoPlane with 4 external devices.

Fig. 10 shows the throughput of each app. The apps that run entirely on the switch (UnivMon and NetCache) process traffic at line rate without dropping any packets. However, we observe that the others cannot process their traffic at line rate. This is because the aggregate amount of traffic across the apps, which needs to be processed at the external device (≈81 Gbps in the gateway case) exceeds the processing capacity of the single device (≈39 Gbps).

Scaling throughput with multiple devices. By adding more devices, ExoPlane can support higher throughput. To demonstrate this, we measure the aggregate throughput of the four apps in the gateway scenario (max. traffic rate in our testbed is ≈394 Gbps) while varying the fraction of traffic offloaded to external device(s)[8] and the number of external devices. Fig. 11 shows the results. In the case of 30, 40, 50% of the traffic being offloaded to external devices, we see the throughput effectively increases with more devices. In contrast, when 100% of traffic is offloaded, adding more external devices is not effective due to load imbalance. This results show the load balancing effect of serving popular flows at the switch, described in §4.4.

Comparison with the app-pinning model. We evaluate the benefit of ExoPlane over the app-pinning model (described in §3.2) while running 4 apps from Table 1. In this model, we place an app along with its entire state at the switch if there is a room. Otherwise, we place it to one of the external devices, which has the largest remaining capacity. Table 3 compares the aggregate throughput when running an ensemble of apps. While ExoPlane provides the maximum throughput for each ensemble, the app-pinning model achieves up to 69.3% lower throughput. This is because while ExoPlane allows an app to effectively utilize available resources across different devices, the app-pinning model fixes an app to a device.

[8]In this experiment, we control the fraction of traffic offloaded to external devices by manually assigning the affinity of each app. UnivMon is still pinned to the switch.



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

Figure 12: Throughput changes due to workload changes.

7.2 Performance under dynamic workload

Packet processing latency. As mentioned in §4.2, workload changes happen due to new flows arriving or changes in flow popularity. Handling new flows in ExoPlane can increase packet processing latency because the first packet of a new flow can be processed only after initiating necessary state for both the ExoPlane flow manager and app objects. 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 §7.1. Thus, for each app, we measure the processing latency of the first packet of each flow. We observe that the median latency for the first packet of a new flow is 32 *ms*, which is an order of magnitude higher than that of an external device in steady state. There are two factors here. First, the Netronome Thrift API takes a few tens of *ms* to insert new entries to objects, which is not an ExoPlane-specific overhead. Second, since ExoPlane replicates entries for new flows to one another external device, it incurs additional latency when handling new flows.

Application throughput. The changes in flow popularity can impact the app throughput. To measure the throughput changes, we use the same setup as the previous measurement in steady state, but for every 10 second, we alter the most popular top 10 flows for the VPN gateway of the gateway scenario. Fig. 12 shows the throughput changes over time. Again, we first use a single external device. As shown in Fig. 12a, when the popularity changes, there is a sharp drop in the throughput of the VPN gateway. 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

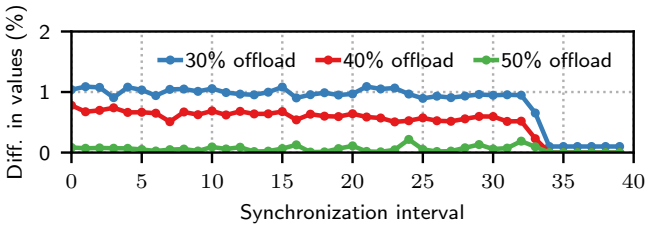


Figure 13: Difference in shared object values on the switch and external devices; the trace ends at epoch 32.

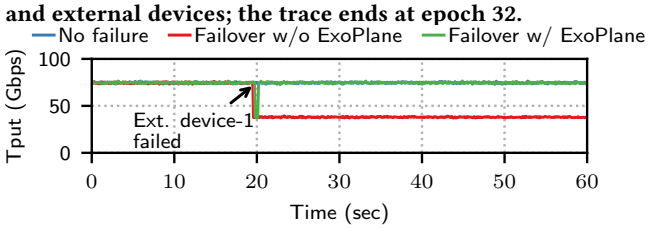


Figure 14: TCP throughput during failover and recovery.

processing capacity. On the other hand, as shown in Fig. 12b, with 4 external devices, there is no such performance drop because there is enough processing capacity at the external devices to handle the traffic during the transient period.

7.3 Shared stateful object synchronization

Next, we evaluate the effectiveness of our state synchronization protocol (§4.3) using the per-SrcIP packet counter in the stateful FW. Here, the metric of interest is the difference between the shared counter entries maintained by each device at each synchronization interval (T_ϵ=1 sec.). We measure this by recording the values at each device right after executing the merge operation in the data plane while injecting 1500 B packets for 60 seconds at 98.6 Gbps. We vary the fraction of traffic offloaded to external devices. In our setting, there are 1000 entries shared between the switch and at least one of the external devices, and we get the median of the differences. Fig. 13 shows the result with three different fractions of offloaded traffic. When the switch and external devices process the same amount of traffic (*i.e.,* 50% offload), there is almost no difference. When there is a gap between the amounts of traffic (*i.e.,* 30% or 40% offload), there are differences because incoming packets keep updating the counter at each devices during the synchronization, affecting the measured values. However, we see that the variance of the difference is small across the synchronization intervals regardless of the gap, showing that our mechanism synchronizes the values. We also confirm that after the packet transmission is done, copies at each device are synchronized with the same value as the total number of packets.

7.4 Failover

In Fig. 14, we use a NAT as an example and run iperf to measure TCP throughput changes. There are 4 TCP connections, and we configure two of them to be processed at the switch while the remaining is processed at an external device. There are two external devices enabled, and we compare changes in TCP throughput when (1) there is no failure and (2) one of the external devices fails with and without ExoPlane. We emulate the failure by disabling a port connected to the external device. At around 20 sec., when the external device-1 goes down, our failover mechanism generates a control packet that modifies the logical to physical device ID mapping in the switch data plane without involving the control plane. Then, subsequent packets are routed to the replica device. We see that the TCP throughput is recovered to its original rate within a 200 ms[9] whereas without ExoPlane, it cannot be recovered.

7.5 Runtime resource overheads

Control plane resource overhead. The control plane component of ExoPlane runtime environment maintains metadata for app states, including a mapping between union keys and devices and a history of each shared object entry on other devices. Each of them consumes the control plane memory. In our scenarios, the union keys to device mapping consumes 12.5 MB per app and the history metadata consumes 1.5 MB per shared object. Our state synchronization protocol consumes management network bandwidth as it periodically exchanges the information between devices, which contains a snapshot and a history of each entry. In our setting, the bandwidth consumption is 24.4 Mbps per shared object, which increases in proportion to the number of devices, the sync. interval, and the number of entries.

Switch ASIC resource usage. The data plane component of ExoPlane runtime environment consumes some switch ASIC resources. Since we implement it using an exact-match table with the aging feature and a register array, it consumes SRAM, SALUs, hash bits, MAP RAM, and match crossbar,[10] whose usage increases proportionally to the number of popular flows maintained (except for SALUs). In our setting where 10240 popular flow entries are managed, it consumes 4.4% of the SRAM, 2.1% of SALUs, 3.5% of the hash bits, 3.8% of the MAP RAM, and 3.6% of the match crossbar, leaving ample resources to application logics.

8 Conclusions

Limited on-chip resources today block the deployment of multiple feature-rich stateful apps on a switch. On-rack switch resource augmentation is an affordable and incrementally expandable solution to this dilemma. To realize this vision, we argued the need for OS-like abstractions and addressed key challenges in realizing these. Our evaluation shows that ExoPlane provides low and predictable latency, scalable throughput, and fast failover, and achieves these

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

[10]MAP RAMs are used for the aging feature and match crossbars are used for implementing the 'matching' part of match-action tables.

with a small resource footprint and few/no modifications to apps. Thus, ExoPlane can be a practical basis for enabling in-network computing workloads for future apps, workloads, and emerging data plane hardware.

References

[1] Netcope P4. https://www.intel.com/content/www/us/en/programmable/solutions/partners/partner-profile/netcope-technologies--a-s-/ip/netcope-p4.html.

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

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

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

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

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

[7] P4-SDNet User Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1252-p4-sdnet.pdf, 2018.

[8] Agilio CX SmartNICs - Netronome. https://www.netronome.com/products/agilio-cx/, 2019.

[9] 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.

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

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

[12] Alveo U25 SmartNIC Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u25.html, 2020.

[13] Apache Thrift. https://thrift.apache.org/, 2020.

[14] 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.

[15] Gurobi - C++ API Overview. https://www.gurobi.com/documentation/9.1/refman/cpp_api_overview.html, 2020.

[16] 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, 2020.

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

[18] NPL Specifications. https://nplang.org/npl/specifications/, 2020.

[19] Nvidia mellanox spectrum-2. https://www.mellanox.com/files/doc-2020/pb-spectrum-2.pdf, 2020.

[20] p4c: a reference P4 compiler. https://github.com/p4lang/p4c, 2020.

[21] P4 DPDK backend. https://github.com/p4lang/p4c/tree/master/backends/dpdk, 2020.

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

[23] The Software Switch Pipeline. https://doc.dpdk.org/guides/prog_guide/packet_framework.html#the-software-switch-swx-pipeline, 2020.

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

[25] The CAIDA UCSD Anonymized Internet Traces. https://www.caida.org/data/passive/passive_dataset.xml, 2021.

[26] Thomas Anderson and Michael Dahlin. *Operating Systems: Principles and Practice*, volume 1. Recursive books, 2014.

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

[28] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

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

[30] 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.

[31] Cisco. Cisco Global Cloud Index: Forecast and Methodology, 2016–2021 White Paper, 2018.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] David Hancock and Jacobus Van der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. In *ACM CoNEXT*, 2016.

[38] 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. 2021.

[39] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. The p4-> netfpga workflow for line-rate packet processing. In *ACM/SIGDA FPGA*, 2019.

[40] 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.

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

[42] Georgios P Katsikas, Tom Barbette, Dejan Kostic, Rebecca Steinert, and Gerald Q Maguire Jr. Metron: Nfv service chains at the true speed of the underlying hardware. In *USENIX NSDI*, 2018.

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

[44] Elie F Kfoury, Jorge Crichigno, and Elias Bou-Harb. An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. 2021.

[45] 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.

[46] 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.

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

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. Pga: Using graphs to express and automatically reconcile network policies. *ACM SIGCOMM Computer Communication Review*, 45(4):29–42, 2015.

[54] 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*, 2019.

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

[56] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. 42(4):323–334, 2012.

[57] 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*, 2021.

[58] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. Composing dataplane programs with μp4. In *ACM SIGCOMM*, 2020.

[59] 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*, 2021.

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

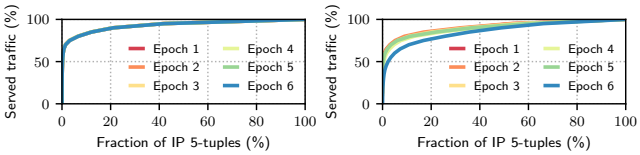
[61] William Tu, Fabian Ruffy, and Mihai Budiu. Linux network programming with p4. In *Linux Plumb. Conf.*

[62] 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*, 2017.

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

[64] 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*, 2017.

[65] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *ACM CoNEXT*, 2018.



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

Figure 15: Skewness in flow key (IP 5-tuple): For both Internet backbone and data center case, a few popular keys serve the most of the traffic. This is consistent across measurement epochs.

A Skewness of traffic traces

We measure the distribution of IP 5-tuple as the union key by analyzing packet traces collected from an Internet backbone [25] and a university data center [27]. Fig. 15 shows the union key distributions of the two data sets.

B State synchronization algorithm

Algorithm 1: State synchronization – Switch

S_{switch} : The current state of the value on the switch
Ext: a set of external device IDs
$Snap_{switch}$: The latest snapshot of the value on the switch
$H_{ext}[1 \ldots N]$: The latest information received from each external device
Upon the snapshot timer expires:
foreach $ext_i \in Ext$ **do**
    /* Send an initiate message to ext_i */
    send $(Snap_{switch}, I_{switch}[ext_i])$;
    /* Receive a response from ext_i */
    $(Snap_{ext_i}, I_{ext_i})$ = **recv** ();
foreach $ext_i \in Ext$ **do**
    /* Adjust snapshot values and merge them */
    $\delta = Snap_{ext_i} \circ^{-} (I_{switch}[ext_i] \circ^{+} I_{ext_i})$;
    /* Update the information for ext_i */
    $I_{switch}[ext_i] = Snap_{ext_i} \circ^{-} I_{ext_i}$
/* Merge ($\circ^{+}$) the adjusted value with the current state in the data plane */
$S_{switch} = S_{switch} \circ^{+} \delta$

In §4.3, we describe our state synchronization protocol to synchronize entries in a data-plane updatable object. Algorithm 1 and Algorithm 2 describe the detailed algorithm running on the switch and external devices, respectively.

C Details of Application Merger

Fig. 16 illustrates how our application merger works for a set of P4 applications as described in §5.

D Performance of ExoPlane Planner

We evaluate the performance of the ExoPlane planner. In this experiment, we measure the elapsed time for finding

Algorithm 2: State synchronization – External device

$Snap_{ext}$: The latest snapshot of the value on the external device
S_{ext} : The current state of the value on the external device
I_{ext} : The latest information received from the switch
Upon receiving a message from the switch $(Snap_{switch}, I_{switch})$:
/* Send a response to the switch */
send $(Snap_{ext}, I_{ext})$;
/* Adjust snapshot values and merge them */
$\delta = Snap_{switch} \circ^{-} (I_{ext} \circ^{+} I_{switch})$;
/* Update the history for the switch */
$I_{ext} = Snap_{switch} \circ^{-} I_{switch}$;
/* Merge the adjusted value with the current state */
$S_{ext} = S_{ext} \circ^{+} \delta$

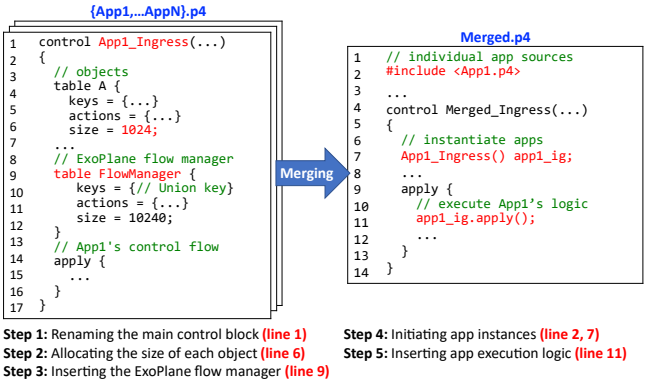
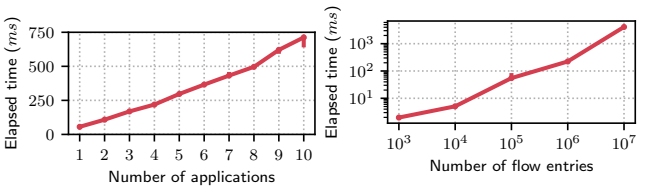


Figure 16: Merging multiple P4 programs into a single program.



(a) Different number of apps. **(b) Different number of flow entries.**

Figure 17: Elapsed time for the resource allocator.

optimal resource allocations and generating a merged P4 program on a server in our testbed. For the two sets of apps and the hardware configuration used in our evaluations, our resource allocation takes 54.5 *ms* and merging the program takes 642 *ms*, which is reasonable since the orchestrator needs to run this process on the hours or days timescale. To further understand the impact of different parameter values including the number of apps and traffic workload sizes, we synthesize inputs for the resource allocator and measure the elapsed time. First, we fix the number of external devices to 16 (to support a large number of apps) and the number of

union key-based flow entries to 1 million for each app. Then, we vary the number of flow entries while fixing the number of apps to 4 and the number of devices same as the above. As illustrated in Fig. 17a, the resource allocation time grows linearly up to 712 *ms* as the number of app increases. Also, as shown in Fig. 17b, as the number of flow entries increases, the elapsed time also increases up to 4.1 second when each app needs to handle 10 million flow entries. This experiment illustrates the ExoPlane orchestrator takes longer time as we add more apps and increases the workload size, which can be up to a few seconds, it is still with in the reasonable timescale under our deployment model.