Software-Defined Datacenter Network Debugging

Praveen Tammana

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2017

Abstract

Software-defined Networking (SDN) enables flexible network management, but as networks evolve to a large number of end-points with diverse network policies, higher speed, and higher utilization, abstraction of networks by SDN makes monitoring and debugging network problems increasingly harder and challenging. While some problems impact packet processing in the data plane (*e.g.*, congestion), some cause policy deployment failures (*e.g.*, hardware bugs); both create inconsistency between operator intent and actual network behavior. Existing debugging tools are not sufficient to accurately detect, localize, and understand the root cause of problems observed in a large-scale networks; either they lack in-network resources (compute, memory, or/and network bandwidth) or take long time for debugging network problems.

This thesis presents three debugging tools: PathDump, SwitchPointer, and Scout, and a technique for tracing packet trajectories called CherryPick. We call for a different approach to network monitoring and debugging: in contrast to implementing debugging functionality entirely in-network, we should carefully partition the debugging tasks between end-hosts and network elements. Towards this direction, we present CherryPick, PathDump, and SwitchPointer. The core of CherryPick is to *cherry-pick* the links that are key to representing an end-to-end path of a packet, and to embed picked linkIDs into its header on its way to destination.

PathDump is an end-host based network debugger based on tracing packet trajectories, and exploits resources at the end-hosts to implement various monitoring and debugging functionalities. PathDump currently runs over a real network comprising only of commodity hardware, and yet, can support surprisingly a large class of network debugging problems with minimal in-network functionality.

The key contributions of SwitchPointer is to efficiently provide network visibility to end-host based network debuggers like PathDump by using switch memory as a "directory service" — each switch, rather than storing telemetry data necessary for debugging functionalities, stores *pointers* to end hosts where relevant telemetry data is stored. The key design choice of thinking about memory as a directory service allows to solve performance problems that were hard or infeasible with existing designs.

Finally, we present and solve a *network policy fault localization problem* that arises in operating policy management frameworks for a production network. We develop Scout, a fully-automated system that localizes faults in a large scale policy deployment and further pin-points the physical-level failures which are most likely cause for observed faults.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in Chapter 4, Chapter 5, Chapter 6, and Chapter 7 have been published in the following papers:

- CherryPick: Tracing Packet Trajectory in Software-defined Datacenter Networks.
  Praveen Tammana, Rachit Agarwal, and Myungjin Lee.
  In ACM SIGCOMM Symposium on SDN Research (SOSR), 2015.

- Simplifying Datacenter Network Debugging with PathDump.
  Praveen Tammana, Rachit Agarwal, and Myungjin Lee.
  In USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2016.

- Distributed Network Monitoring and Debugging with SwitchPointer.
  Praveen Tammana, Rachit Agarwal, and Myungjin Lee.
  In USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2018.

- Fault Localization in Large-Scale Network Policy Deployment.
  Praveen Tammana, Chandra, Pavan, Ramana Kompella, and Myungjin Lee.
  In IEEE International Conference on Distributed Computing Systems (ICDCS), 2018.

(*Praveen Tammana*)

Table of Contents

Chapter 1

Introduction

Today, data centers are a key computing infrastructure that drives the Internet. Numerous services such as search, online social media, big data analytics, and cloud applications rely on data centers. Information technology giants like Microsoft, Google, Facebook, and Amazon build and maintain data centers that have hundreds of thousands of servers, thousands of network devices, and storage devices [112]. Using this infrastructure, a huge amount of data is generated and consumed every day by billions of users in the globe.

As size of network elements in the data centers grow, Software-Defined Networking (SDN) has emerged as a key technology for managing the network infrastructure. In SDN, network administrators express their desire to meet application demands as a network policy and provide the policy to a centralized controller. The controller further converts policy into low-level per-switch instructions. For example, a routing policy is converted to low-level forwarding rules (see Figure 1.1), security policy to access control list (ACL) rules, load balance policy to equal-cost multi-path (ECMP) configurations, and quality of service (QoS) policy to per-port priority queue configurations. Upon receiving the packet, switch data plane makes forwarding decisions based on the forwarding and ACL rules, and puts the packet into appropriate queue as defined by a load balance and QoS policy.

In a large complex network environment network problems are inevitable. While troubleshooting a network problem, operators need to think of multiple possibilities. A problematic network behavior could be due to many reasons — misconfiguration (*e.g.*, incorrect forwarding rules), malfunction of network devices (*e.g.*, faulty interface), or lack of network resources (*e.g.*, congestion due to buffer overflow), and their combination. The presence of these problems would create inconsistency between network

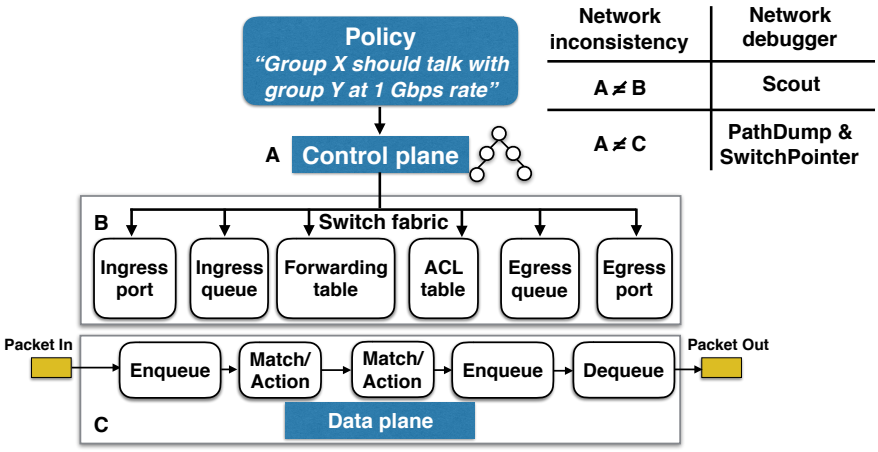


Figure 1.1: A policy is compiled to low-level switch rules and configurations. Based on the rules and configurations, switch data plane makes packet forwarding decisions. However, problems in the data plane (*e.g.*, ECMP collisions, network congestion) cause inconsistency between control plane (*A*) and actual network behavior. Similarly, policy deployment failures due to physical-level faults (*e.g.*, TCAM memory overflow, control channel disruption, etc.) might cause inconsistency between control plane (A) and switch rules and configurations (B).

admin's intent and actual network behavior which in-turn triggers spurious events; *e.g.*, high job completion times, degradation in quality of search query results, etc.

Ideally network administrators want a fully automated debugging tool — a key component in self-driving networks — that consumes minimal in-network resources (*e.g.*, switch cpu, switch memory, and network bandwidth) for monitoring and allows to debug a large class of network problems. Moreover, it should allows to detect, localize, inspect, and fix the problems at fine-time scales in the order of milliseconds to seconds.

Monitoring and debugging network problems is complex. There have been many recent systems (*e.g.*, Trumpet [71], EverFlow [112], PathQuery [75] Marple [74]) for debugging network problems. While each of these works has different approach, debugging problems observed in a large-scale network still remains challenging. Moreover, as data centers evolve to a large number of end-points ($>$ 100k), diverse network policies (routing, security, and QoS), higher utilization (aggregate traffic can exceed 100Tbps), and higher speed (10/40/100G), monitoring and debugging become more challenging.

To better understand the nature of network problems, we have investigated problems handled in many recent works and broadly divided them into two categories. First category has the network problems in the switch data plane that impact packet

processing. Moreover, these problems cannot be detected at a centralized point either by analyzing network policies or corresponding switch configurations, but their presence impact actual network behavior in the data plane (*C* in Figure 1.1). Some example problems are load imbalance due to ECMP poor hashing [12], random packet drops by a faulty interface [112], etc.

Second category has the problems that cause the inconsistency between control plane policy and low-level per-switch rules and configurations. The inconsistency between the policy and the network state could be detected at a centralized controller either by continuously monitoring updates pushed to the network [60], or by periodically collecting and analyzing the rules and configurations [66] such as routing configurations, switch table rules, etc. However, it is hard to localize and find the root cause that creates such inconsistency. Some possible causes are errors in routing or security policy compilation, control channel disruption, device memory overflow, and their combination.

This thesis presents three debugging tools PathDump, SwitchPointer, and Scout, and also a packet trajectory tracing technique called CherryPick. While CherryPick, PathDump and SwitchPointer together enables debugging a large class of network problems in the data plane (first category), Scout is an end-to-end system that localizes failures and also pinpoints the cause for a failure in a large-scale policy deployment (second category). Note, the tools do not target "automated" debugging but rather allows to build a framework to simplify the debugging process, and enables network operators to quickly detect, localize, and understand the cause for a large class of network problems.

1.1 Problems and contributions

This section provides discussion on the two problems addressed in this thesis. It covers recent works in network debugging, their limitations, and outlines the thesis contributions.

Problem 1: *Existing systems are insufficient to monitor and debug network problems in the data plane. Debugging requires both network visibility and resources to capture the visibility.*

Managing large-scale networks is complex. Even short-lived problems due to failures, load imbalance, faulty hardware and software bugs can severely impact performance and revenue [44, 71, 112].

Existing tools to monitor and debug network problems operate at one of the two extremes. On the one hand, proposals for in-network monitoring argue for capturing telemetry data (*e.g.*, flowIDs, packet headers) at switches [13, 63, 107, 64, 50], and querying this data using new switch interfaces [74, 41, 75, 6] and hardware [51, 74]. Such in-network approaches provide visibility into the network that may be necessary to debug a class of network problems; however, these approaches are often limited by data plane resources (switch memory and/or network bandwidth) and thus have to rely on sampling or approximate counters which are not accurate enough for monitoring and diagnosing many network problems (§6.2).

At the other extreme are recent systems [71, 44] that use end-hosts to collect and monitor telemetry data, and to use this data to debug spurious network events. The motivation behind such end-host based approaches is two-fold [71]. First, hosts not only have more available resources than switches but also already need to process packets; thus, monitoring and debugging functionalities can potentially be integrated within the packet processing pipeline with little additional overhead. Second, hosts offer the programmability [71] needed to implement various monitoring and debugging functionalities without any specialized hardware. While well-motivated, such purely end-host based approaches lose the benefits of network visibility offered by in-network approaches.

Contributions. This thesis calls for a radically different approach for network monitoring and debugging: in contrast to implementing the debugging functionalities entirely in-network or at end-hosts, we should carefully partition the debugging tasks between end-hosts and network elements. This approach not only reduces the overhead on network resources, but also allows to debug the problems one can see the other cannot.

Towards this direction, this thesis presents CherryPick, PathDump, and SwitchPointer, all three together integrates the best of two worlds — resources and programmability of end-hosts, and network visibility offered by the network elements. CherryPick is a scalable, yet simple technique for tracing packet trajectories in SDN-enabled data center networks. The main idea is instead of picking every link that a packet traverse towards the destination, CherryPick exploits common data center network topologies like fat-tree, VL2, and selectively chooses the links that are sufficient to represent the end-to-end path. More details of CherryPick can be found in Chapter 4 and §5.3.1.

PathDump is an end-host based network debugger that exploits resources and programmability of end-hosts to collect and store telemetry data necessary for debugging

network problems. In addition, each individual end-host also expose query service that allows PathDump to filter telemetry data stored across multiple end-hosts in a distributed manner. PathDump design is based on tracing packet trajectories using link sampling technique like CherryPick. Visibility at each individual end-host, along with path information provided by CherryPick allows PathDump to debug a large class of network problems (present in *C* of Figure 1.1) with minimal in-network functionality (see Table 5.2 for the supported problems). More details of PathDump can be found in Chapter 5.

Although end-hosts in PathDump know paths of every packet they receive, and debugs problems for which path information is sufficient, PathDump still losses the in-network visibility, thus unable to debug a class of performance problems. For instance, hosts cannot localize (*e.g.*, specific switch in the path) and understand the cause (*e.g.*, contending flows and packets) of a spurious event (*e.g.*, latency, packet drops). SwitchPointer efficiently enable network visibility to end-host based monitoring systems like PathDump by using switch memory as a "directory service" — in contrast to in-network approaches where switches store telemetry data [75, 74, 107] necessary to diagnose network problems, SwitchPointer switches store *pointers* to end-hosts where the relevant telemetry data is stored. The distributed storage at switches thus operates as a distributed directory service; when an end-host triggers a spurious network event, SwitchPointer uses the distributed directory service to quickly filter the data (potentially distributed across multiple end-hosts) necessary to debug the event. The key design choice of thinking about network switch storage as a directory service rather than a data store allows to efficiently solve many problems that are hard or even infeasible for existing systems. More details of SwitchPointer can be found in Chapter 6.

Discussion on the supported network problems, that is, coverage of CherryPick, PathDump and SwitchPointer can be found at `https://github.com/PathDump/Applications`.

Problem 2: *Debugging network policy deployment failures takes time.*

A number of frameworks (*e.g.*., PGA [82] APIC [15], Merlin [92], Frenetic [41], Pyretic [69], GBP [9]) aid network policy management tasks through abstraction, policy composition, and deployment. However, these frameworks are not immune to various faulty situations that can rise from misconfiguration [60], software bugs, hardware failure [112, 44], etc. Many of them incur a flow of instructions from a centralized controller, to a software agent in a network device and finally to ternary content addressable memory (TCAM) in that device. A failure of any element in this data flow

or at physical-level can significantly disturb the network policy deployment process.

Typically, policy management frameworks [82, 15] represent the intent of network admins using policy objects (in short, objects) such as marketing group, DB tier, filter, and so on. When a network policy is not rendered in the network (*e.g.*, a large number of low-level TCAM rules are missing) admins observe a large number of failure notifications. So, admins should first understand which part (set of objects) of the policy has been affected. Today, it is challenging to localize because the low-level rules is the final outcome after compilation of a large number of policy objects and their inter dependencies. So, admins can end up spending lot of time examining tens of thousands of low-level rules to localize a small set of objects that become faulty due to failures — a needle-in-a-haystack problem. Therfore, *admins require a fully-automated means that quickly nail down to the part of the policy they should look into or further diagnose in order to fix a large number of observed failures.*

There have been many recent works [59, 60, 66] on detecting inconsistency between high-level network policy (*A* in Figure 1.1) at the controller and low-level table entries and configurations (*B* in Figure 1.1). But, localization and identifying the root cause for inconsistency between *A* and *B* which is as equally important as detecting inconsistency is understudied. In specific, an ideal debugging tool should localize high-level faulty policy object (in *A*) that the operator should dig deeper (to fix the problem), and also provide most likely root cause (*e.g.*, physical-level failure) for the objects to become faulty.

Contribution. We call the problem of finding out the impaired parts of the policy as a *network policy fault localization problem*. This thesis presents Scout, an end-to-end system that automatically pinpoints not only faulty policy objects, but also physical-level failures; the cause for policy objects becoming faulty. We tackle it via *risk modeling* [62]; risks are modeled as simple bipartite graphs that capture dependencies between risks (*i.e.*, objects) and nodes (*e.g.*, endpoints or end user applications) in low-level rules. We then annotate the risk models for those risks and nodes that are associated with the observed failures. Using those models, we devise a greedy fault localization algorithm that outputs a hypothesis, a minimum set of most-likely faulty policy objects (*i.e.*, risks) that explains most of the observed failures. More details in Chapter 7.

1.2 Thesis organization

This thesis is organized into the following chapters.

Chapter 2 provides background on the data center network environment. It also explains how network problems degrades performance of applications, and technical challenges in debugging those problems.

Chapter 3 discusses related work on network monitoring and debugging. It provides shortcomings of the existing approaches, and how CherryPick, PathDump, SwitchPointer, and Scout fill the gap.

Chapter 4 presents CherryPick, a packet trajectory tracing technique operates in L2/L3 layer. CherryPick works for common data center topologies like fat-tree and VL2 that contains commodity OpenFlow compatible switches, and requires no changes to hardware. This chapter is based on work published in ACM SIGCOMM SOSR, 2015.

Chapter 5 presents PathDump, an end-host based network debugger that simplifies network debugging and enables debugging a large class of network problems with minimal in-network functionality. It explains the details of PathDump design (§5.2), implementation (§5.3), and example applications (§5.4). This chapter is based on the work published in USENIX OSDI, 2016.

Chapter 6 presents SwitchPointer, a distributed network monitoring and debugging tool. The key contribution of SwitchPointer is to enable network visibility to end-host based monitoring approaches like PathDump. It explains the details of SwitchPointer design (§6.4), implementation, and a few SwitchPointer usecases (§6.5). This chapter is based on the work published in USENIX NSDI, 2018.

Chapter 7 presents Scout, an end-to-end system that localizes the policy objects become faulty due to policy deployment failures and also pin-points the root cause for the object become faulty in-terms of physical-level failures. More details on risk models (§7.3), a greedy-based fault localization algorithm (§7.4), and the Scout system (§7.5) can be found in this chapter. This chapter is based on the work to appear in IEEE ICDCS, 2018.

Chapter 8 concludes the thesis. It outlines areas of future work and key contributions of this thesis.

Chapter 2

Background

This chapter discusses the data center network environment, a few physical-level faults observed in production networks, and some challenging problems that draw the attention of the data center networking community.

2.1 The data center environment

Network role in a data center. Many applications deployed in data centers expect non-blocking communication between their compute and storage servers. So, network in a data center moves data traffic between the servers that run both delay-sensitive (*e.g.*, web search) and bandwidth-intensive applications (*e.g.*, big data).

To better understand the network role inside a data center, consider how a search query might work. When a user makes a search query, the query hits a server in a data center. This server might query several other servers, which communicate with several other servers and so on. Responses from the individual servers are collated, and the final search response is sent to the user. For one query, there might be a large number of server to server interactions, and deadline of each interaction could be as small as 10ms [20]. This kind of communication pattern is referred to as scatter-gather or partition-aggregation.

Scatter-gather is not exclusive to the search. Similar communication pattern is observed while loading web pages. For instance, a recent study in Facebook data center [78] while loading a popular web page shows that there are 521 internal requests on average, at 95 percentile, internal requests are over 1700. Further, data centers often run big data analytics tools such as Hadoop, Spark, and Database joins, which all process data to provide a response to queries. These big data processing tools move massive amounts of data around.

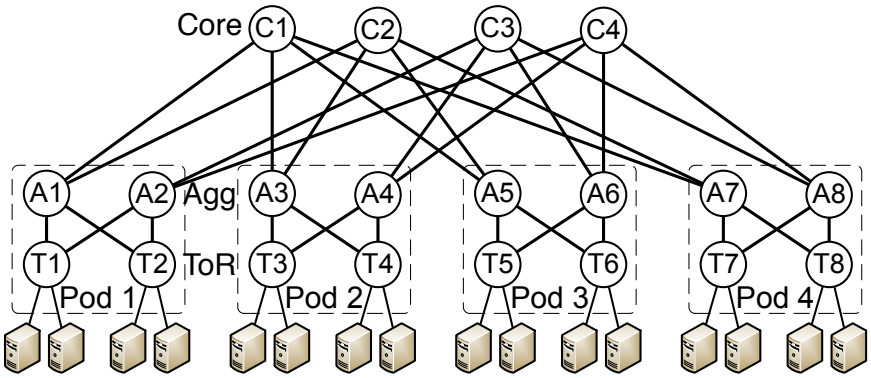


Figure 2.1: 4-ary fat-tree topology

Workloads. Broadly, the applications and their flow sizes can be categorized into three types [20]: query traffic (2KB to 20KB), latency sensitive short messages (1MB to 100MB), and throughput sensitive long flows (longer than 100MB). Some example applications are: ARP, DNS (latency sensitive); hadoop, spark, and database joins (latency and throughput sensitive); VM migration, and large size file transfers (throughput sensitive).

Multi-rooted tree topologies. Data center networks are built based on scale-out architectures that consist of switches in multiple layers. For instance, consider fat-tree [17] network topology. It has switches at three layers: top of rack (ToR), aggregate and core. Hosts are connected to a ToR switch, a group of ToR switches is connected to an aggregate switch, and finally, a core switch connects a group of aggregate switches. A K-ary fat-tree topology has K pods; a pod has $K/2$ ToR and $K/2$ aggregate switches. $K/2$ ports of a ToR switch are connected to end-hosts, and the remaining $K/2$ ports are connected to $K/2$ aggregate switches in the same pod. Furthermore the other $(K/2)$ aggregate switch ports are connected to $(K/2)$ core switches.

To meet application demands, in some cases, switches are configured to forward an end-host pair traffic across multiple paths. For example, in a K-ary fat-tree topology there are $(K/2)^2$ equal length shortest paths between any source-destination pair (for K=48, there are 576 shortest paths [17]).

Equal-cost multi-path (ECMP). Switches configured with ECMP load balance traffic among equal cost paths [18]. In specific, for each incoming packet a switch applies hash on header fields (*e.g.*, 5-tuple flow ID) and generates a key. The Key space is divided equally by the number of possible output ports. In other words, each divided region in the key space is mapped to one output port. If the packet's key falls into one of the subregions, then the port mapped to the region is selected as the next hop.

Software-Defined Networking (SDN). In conventional network switches, the control

logic is co-located with the switching logic [26]. Therefore, a network policy is realized by configuring individual switches. But as the size of the network increases, managing box by box becomes hard. SDN makes network management easier. It decouples the network control plane (brain) from the data plane (forwarding engine) [26]; the control logic is separated from the switches and moved to a centralized controller. This way SDN abstracts underlying network infrastructure complexity, and the SDN applications program the network treating it as a single logical entity (one big switch). Some key network management applications that drive SDN are: 1) VLAN configuration during VM migration [43]; 2) Quick allocation of network resources to meet user application demands; and 3) Enforcement of policies like QoS, isolation, and security [56].

Commodity switches. Large-scale data center networks primarily use low-cost commodity switches that have limited resources [17]. These switches should process packets in nano seconds to keep up with high-speed line rates (*e.g.*, 10Gbps, 40Gbps and more). To enable fast look up, switches store the state required (lookup tables) for forwarding packets in expensive and power-hungry memories (SRAM and TCAM), and these memories are limited in size due to cost and power constraints.

In general, SRAM stores look-up tables that have entries similar to key-value pairs [63]. For example, a set of destination MAC addresses is assigned to a particular interface [43]. When an incoming packet's destination matches any particular entry, it is forwarded to the assigned interface. Similarly, TCAM stores wild-card entries, each entry can have don't care values, either 1 or 0. Typically, access control list (ACL) rules that allow, discard, or rate-limit packets are stored in TCAM. Note, TCAM is more expensive and consumes more power than SRAM.

Despite the fact that switch memory is costly and limited, operators want to fully utilize the memory (store a large number of entries to route traffic). If some applications need different QoS and access control, it requires even more memory. Likewise, monitoring functionality also require memory resources [74, 63] to maintain counters, flow statistics, etc.

Strict limits on per-packet operations and limited available memory make it hard to maintain a set of active flows at the merchant silicon. There is limited time on each packet to spend; only 12 ns to process a 64 byte packet arrive on a 40Gbps port. Modern on chip SRAM has about 1 ns access time. Assuming the whole 12 ns is assigned to monitoring, it is still challenging to do SRAM lookup, perform a few ALU operations, and write back within the given time budget [63].

End-host networking. Many recent works instrumented end-hosts in data centers to enable multiple network related tasks [32, 97, 55]; *e.g.*, congestion control, tunneling, access control, monitoring, and debugging. Some of the key motivations for involving end-hosts are: (1) Availability of computation power, memory, and storage; (2) Higher visibility into the behavior of applications it hosts with little overhead; and (3) Complete control of datacenter-wide network, storage, and compute resources under a single administrative domain [106, 97, 71].

2.2 Network data plane faults

This section describes a few data plane faults observed in the production data centers, and the challenges to diagnose these problems with existing debugging tools. CherryPick, PathDump and SwitchPointer system designs are motivated to address these problems.

Silent packet drops. It is challenging to localize the culprit switch silently dropping packets. This could be because of a faulty interface dropping a fraction of packets at random [112], or bit flaps in a switch fabric module [44]. Such type of faults cause performance degradation (due to high TCP retransmissions) to network-wide flows passing through the faulty switch.

Furthermore, the higher the level of the faulty switch in the network topology, the more severe in-terms of number of flows or applications would suffer. With hundreds of switches in the network and multiple equal-cost paths between a source-destination pair, localizing the faulty switch or link is not trivial. Conventional tools like ping and trace-route may not observe the problems encountered by actual traffic. So, operators have to run the tests from multiple places, infer the rough location, and inspect individual switches in the location. Because there is a large number of links (*e.g.*, 25,000), using these tools to localize the problem has significant overhead and can take several hours [112].

Silent blackhole. It is a type of routing blackhole that does not show up in forwarding tables. For example, a forwarding table entry in the switch TCAM is corrupted [112]. It may cause packets with a specific source-destination IP address pair or source destination port numbers to be dropped [44]. Such faults cannot be detected by examining switch forwarding tables. To illustrate the impact of the fault, suppose that a faulty switch is present in one of the multiple paths between a pair of end-hosts. Operators might observe some requests between the end-hosts are success, while others

fail [112]. Unless the operators look into all possible paths and every switch present in those paths, the faulty switch cannot be easily localized. Therefore, debugging a silent blackhole is as challenging as the silent random packet drop problem.

ECMP load imbalance. ECMP load imbalance due to poor hashing [12] would degrades application performance. Even though, load imbalance can be detected after looking at the difference between interface counters using tools like SNMP link monitor, the coarse-grained interface counters are not sufficient to understand the root cause of load imbalance. Debugging load imbalance requires fine-grained telemetry data (*e.g.*, contending flows, packet headers) in fine time-scales (*e.g.*, order of milliseconds to seconds)

For instance, it is essential for network operators to distinguish between two possible causes for the load imbalance: Is there a hardware bug causing ECMP poor hashing on flow tuple, or is there any particular application that sends a sudden burst of traffic on a particular interface that lead to an imbalance between two interface counters. To find out answers, first the operator needs to know which flows (and their sizes) contribute to those counters. In other words, per-flow statistics (5-tuple flow ID, and its size) at regular intervals is essential to the operator to make a right decision (reconfigure ECMP function or limit application bandwidth rate). However, maintaining flow-level visibility comes at the cost of per-packet operations that require additional CPU and memory resources [75, 74, 63].

Inflated end-to-end latency. Many datacenter applications have tight deadlines on flow completion times (*e.g.*, 1 ms per network tier [112]). But packet delays due to queue build-up at a single switch, or at multiple switches en-route to the destination may miss the completion deadlines. Troubleshooting end-to-end latency is challenging. For instance, consider in-network load balancing technique like ECMP is active. There is no easy way to identify the set of links or switches that a packet has traversed.

Even in a case where the links traversed by a packet are known in advance, a link could be shared by multiple flows. If there is a hash collision, it is hard to tell which flows are contributing to large queue size [74]. Such fine-grained visibility is essential to understand the root cause of the delay. However, conventional tools like traceroute are not sufficient to debug latency problems mainly because of two reasons: First, the RTT inferred from the test packets could be different from what original packet has encountered. Second, since the test packet is handled by the switch CPU, RTT could be inflated by the control plane, therefore inferred RTT could be noisy or unusable [112].

2.3 Flow contention

Many recent studies on application communication patterns showed that multiple flows contend for the same outgoing port. Contention causes congestion that eventually inflates flow completion time (due to packet loss, time outs, etc). There have been numerous works that proposed ways to mitigate contention; *e.g.*, changes to end-host congestion control mechanisms [84], efficient scheduling at switches [68], dynamic path changes [18, 21, 19, 57], and coordination between switches and end-hosts [20]. However, as applications evolve, network operators may see new traffic patterns, and need to update prior contention control techniques. This section provides discussion on three popular application communication patterns that effect application performance and solutions proposed to mitigate the collisions.

ECMP collisions. Despite the load balancing, the main drawback of ECMP technique is collisions due to the stateless behavior of hashing technique [18]. Since data center commodity switches have shallow buffers, collision of big flows (*i.e.*, flows that occupy a significant bandwidth of link capacity) on the same outgoing port quickly fill the queue buffers, in-turn causes packet drops. Moreover, a collision of big flows and latency-sensitive small flows adds more queuing delay to small flow packets. For example, if two flows contending (*i.e.* due to hash collision) for a single output link with capacity 1Gbps, there will be 50% throughput loss for each flow.

One idea is to collect flow information from the switches and avoid ECMP collisions by scheduling heavy hitter flows over other under-utilized paths [18]. An alternative approach is to detect heavy flow from endhost socket logs [32] and alarm intermediate switches about the flow. This technique reduces monitoring overhead on switches and helps to detect heavy flow before it become significant.

TCP in-cast. To handle latency-sensitive queries efficiently, certain data center applications follow partition-aggregation communication pattern. Some example applications are social media, web queries, and big data. Typically, the communication tree for these applications has non-leaf aggregate nodes and leaf worker nodes. Requested queries are first handled by a set of aggregate nodes, which in-turn distribute requests among worker nodes present in next level. If there is any unexpected delay in the response from any worker, a timeout occurs that eventually reduces the quality of final results. Specifically, during the aggregate phase, all flows carrying responses are directed towards a single output port to which the aggregate node is connected. The switch buffer suddenly overflows and the switch drops packets. In the worst case,

packet drops not only degrade application performance, but also under utilize the link capacity. This classic problem is named as TCP in-cast [20], and many works that includes DCTCP [20], CONGA [19], and pFabric [21]) provide solutions to address the problem.

TCP outcast. Another problem observed in application communication pattern is TCP outcast [83]. When a large set of tcp flows and a small set of tcp flows arrive on different switch ports and contend for the same output port, packet loss due to port blackout (common in commodity switches) preferentially cause timeouts for a small set of flows. This is due to the fact that consecutive packet drops have more performance impact on the small set of flows as they lose the tail of entire congestion window and result in a timeout, eventually creating unfairness for the flows present in the small set.

2.4 Summary

This chapter provides background on the data center network environment, explains a representative network faults in the data plane, and describes flow-level contention observed at the network switches. In the next chapter, we present related work on monitoring and debugging network problems introduced in this Chapter and Section 1.1.

Chapter 3

Related work

In this chapter, Section 3.1 discusses in-network debugging systems using complex in-network techniques. Next, Section 3.2 elaborates on why the existing monitoring solutions are either hard or infeasible to debug network problems. Finally, Section 3.3 discusses most related work on fault localization problems in the literature.

3.1 In-network debugging

Many existing debugging tools have incorporated increasingly complex in-network techniques. The idea of in-network techniques is to exploit programmable switches to capture debugging information at switches and then send it to a centralized collector to do further analysis and debug the network problems. This Section illustrates the complexity of the existing in-network techniques.

Data plane snapshots. Several recent works [59, 66, 60] have proposed to take a snapshot of the entire data plane state, build models, and analyze (by checking various conditions) this in the background for network debugging purposes. The data plane state includes switch forwarding tables, ACLs, routing configurations, etc. However, collecting the network state in a consistent manner may require freezing the entire network, take a snapshot and then unfreeze the network, which is hard. Moreover, these works do not deal with debugging problems occur while processing packets in the data plane like network congestion, silent packet drops, silent black holes etc.

Per-switch per-packet logs. Dataplane snapshots can be avoided by collecting packet logs for every packet at every switch [47]. Thus, one can analyze these logs to debug both persistent (*e.g.*, reachability) and transient network problems (*e.g.*, packet drops). Suppose that 1000 byte packet traverses five switches and each log is 40 bytes, then for

each packet, you are generating 200 bytes of data. If traffic is 100Tbps, then log generation requires 20 Tbps of additional network bandwidth. Therefore, this technique has high bandwidth overhead.

Packet sampling and mirroring. Bandwidth overhead can be greatly reduced by selectively sampling packets [85, 112, 11]. We can generate packet logs only for those sampled packets or in fact, there is some technique to completely mirror the entire packet to be able to do a more in-depth analysis. However, the question is which packets to sample? For example, consider one of the network problems we mentioned earlier, the silent random packet drop case. If the switch is dropping one out of 1000 packets in a flow, unless we sample exactly that packet the switch is dropping, we will not be able to debug the problem. Since these are random packet drops, figuring out which packet to sample is not trivial.

Everflow [112] selectively mirrors traffic to reduce storage and processing overhead. In specific, Everflow traces flows by capturing only control packets (*e.g.*, TCP-SYN packet) assuming that subsequent data packets follow the same path of the control packets. Furthermore, it uses a guided probing technique; carefully injects test packets with customized headers to match with a specific match-action rule. However, the functionality implemented at the network elements (precisely the elements that these tools are trying to debug) is even more complex.

Dynamic rule updates. In contrast to previously discussed tools, a class of tools [75] executes SQL-like queries on switches while storing the query state in the packet header. However, it has the complexity of dynamic rule installations in order to execute a query such as adding or deleting flow rules across multiple switches.

Packet probes. Another line of work sends packet probes [16, 109, 44] into the network and infers what went wrong with the original packets from the performance of the probed packets. For instance, pingmesh [44] leverages all servers to launch TCP or HTTP pings to provide maximum latency measurement coverage. The measured latency data is collected and analyzed while troubleshooting performance problems. Similarly, ATPG [109] sends test packets to check liveness properties such as reachability failures (due to link faults, missing forward entry, etc.) and throughput degradation (due to network congestion). However, test packet generation is insufficient to debug transient problems like poor ECMP hash collisions, load imbalance, silent random packet drop, etc., that require per-packet visibility.

Networks are complex because of its scale, complex dependencies, and diverse

network policies to be enforced. Existing in-network debugging tools have incorporated increasingly complex techniques on an already complex network. In contrast, we propose PathDump, a simple end-host based network debugger for data center networks. With PathDump, there is no collection of dataplane snapshots, no per-packet per-switch logs, no packet sampling, no packet mirroring, no active probing, and no dynamic switch rule installation. PathDump, by pushing much of the network debugging functionality to the end-hosts, it gives up on a small class of network debugging problems (more details in Chapter 5), while executing debugging with high accuracy at fine-grained time-scales with low overhead.

3.2 Distributed network monitoring

Traditionally, flow-level network monitoring supports vital network management tasks such as traffic engineering, anomaly detection, accounting, understanding traffic structure, detecting worms, scans and botnet activities. As data centers evolve, to meet network service level agreements (SLAs) of user applications, operator needs to add more management and debugging tasks to the list. This ever-growing list of tasks necessitates fine-grained visibility at packet-level, this is in contrast to sampled flow-level information provided by many popular monitoring tools like NetFlow and sFlow. To this end, there have been many recent proposals that capture telemetry data at packet-level using new data structures at switches [107, 63], and query this data using new switch software or hardware interfaces [74, 75]. But, as network evolves in-terms of speed, utilization, and diverse policies, monitoring and debugging network problems become more challenging.

Switch data structures. Typically, network flow monitoring involves per-packet operations; hash function calculation on 5-tuple, memory (SRAM) lookup, run a few ALU operations, and write to memory (update counters). But, for high-speed interfaces (*e.g.*, 40 Gbps), processor and memory for updating flow entry cannot keep up with the line rate. To be more precise, at 40 Gbps rate, a packet has to be processed in 12 ns. This is challenging, because packet processing involves stages such as packet header parsing, layer 2/3 forwarding, ACL, etc., in addition to monitoring tasks. Therefore, Cisco introduced sampled NetFlow [13] to reduce overhead, where 1 out of N packets is sampled. When using sampled NetFlow, the actual counts are estimated by multiplying with N.

Following this idea and respecting the resource constraints, many sophisticated

inference techniques with some guarantees on accuracy are proposed. Initial works embrace approximation techniques that trade off accuracy for switch memory. These techniques can be broadly divided into two categories: (1) packet sampling [89, 90, 38, 37]; and (2) task-specific measurements [101, 38, 107, 63, 50]. The next two paragraphs discuss these techniques.

The idea of packet sampling is to sample a subset of packets with some probability, aggregate sampled packets, then generate reports. Researchers observed [38] large portion of traffic is occupied by small percentage of large flows, and the key management tasks such as traffic engineering and accounting need such flow statistics. Since, packet sampling is biased towards large flows, the probability of sampling small flow packets is very low, therefore it offers inadequate accuracy to many management tasks that rely on small flow statistics like anomaly detection, scans, etc.

With the goal to improve flow monitoring coverage, cSamp [89] proposed a framework for network-wide flow monitoring. The main idea is to assign a hash range to each switch, such that, if hash value of a flow (based on 5-tuple) key that is extracted from the packet fall in the hash range assigned to the switch, then the switch monitor (maintain counters) that flow. Furthermore, a different hash range is assigned to each switch. Thus, same flow is not sampled by more than one switch. This way the resources distributed across the switches allows to monitor a more number of flows, thus more coverage.

Another way to address the limitations of packet sampling techniques is with task-oriented monitoring. The broad idea is to share available memory among a set of measurement tasks, and only monitor the traffic relevant to the task at line rate. This is possible using data streaming [101, 38] or sketch algorithms [107, 50] designed to address particular tasks such as flow-size distribution, super spreader, heavy hitters, etc.

But task-oriented approaches lack generality and also make it harder for network equipment vendors to realize it in practice. It is due to this fact that, early commitment is necessary for designing hardware and hard to modify in future. Moreover, recent sketch-based solutions [107, 63, 50, 64] have a fundamental trade-off between measurement accuracy and resource usage, being inappropriate to debug problems that require more switch resources and high accuracy.

In contrast to the above techniques, SwitchPointer switches only stored pointers to end-hosts to locate telemetry data for monitoring and debugging. Thus, SwitchPointer requires minimal switch resources (4-6 MB of SRAM and 1-2 Mbps of bandwidth

between the control plane and data plane), but can still debug network problems that are hard or infeasible by sketch or stream-based approaches.

Switch interfaces. A body of work defines programming abstractions [75, 41, 45, 69] for network monitoring. These approaches enable flow monitoring by installing flow level rules either in forwarding tables or separate tables. For instance, PathQuery [75] supports network debugging by dynamically installing switch flow rules, and run SQL-like queries on these switches. But the problem is operators need to know in advance the network problems that they wish to monitor. Without prior knowledge, it is hard to obtain relevant monitoring data.

Switch hardware. Some recent works [6, 51] is quite similar to SwitchPointer (more details in Chapter 6) in that they use network switches to embed telemetry data (input port, output port, queue size, etc) into the packet header. The embedded data is extracted at the end-points (*e.g.*, edge swtiches, end-hosts), used later to debug network problems by executing queries on the data distributed across all end-points. The key difference between these approaches and SwitchPointer is that SwitchPointer can locate the useful telemetry data easily but they have to search for it across all the end-points in the network, unless all the data are centrally collected, which is prohibitively expensive.

Another recent work Marple [74] programs switch hardware that allows to install predicates (*e.g.*, Does a packet observed high queuing delays?). The predicates are checked against each incoming packet directly on the switch data plane. When a predicate is satisfied at a switch, it reports telemetry data necessary for debugging the problem to an analyzer. However, if a network problem occurs due to spurious events (*e.g.*, delay) distributed across multiple switches, then Marple cannot debug the problem. For example, if a packet is delayed for 10 msec distributed across multiple switches in its path, then the switch close to the destination detects the problem and reports to analyzer. But we cannot debug the cause for delay since it requires telemetry data of contending flows and packets at other switches in the packet's path. Moreover, Marple requires processing and memory resources to execute installed predicates. The amount of required resources varies for each predicate.

End-host monitoring. Several recent proposals have advocated moving the monitoring functionality to the edge-points [28, 71, 106]. SNAP [106] logs events (e.g., TCP statistics and socket-calls) at the end-hosts to infer network problems. Hone [97] studied host-network traffic management by deploying user-defined programs onto end-

hosts. These programs monitor local traffic at an end-host, and transmit the required data for management tasks running on a centralized controller. Trumpet [71] proposes to push the debugging functionality to the end-host. Specifically, the end-host agent in trumpet inspects every single packet at line rate, and checks a wide set of events. Finally, Felix [28] proposed a declarative query language for end-host based network measurement.

While end-host based approaches work well in monitoring application performance due to lack of visibility into network core, they are insufficient to accurately debug transient performance problems (*e.g.*, latency, packet loss). On the contrary, SwitchPointer has knowledge of two worlds — application performance at end-hosts, and fine-grained visibility of contending flows and packets in the network at millisecond level timescale. Therefore, operators can relate application performance drop with network events and better understand the root cause. In fact, since SwitchPointer rely on end-host based network monitoring like PathDump and Trumpet, it can benefit by adopting features (*e.g.*, timely triggering mechanisms in Trumpet [71]) from those systems.

3.3 Fault localization in network policy deployment

A large body of research work has been conducted for network fault localization [62, 53, 23, 72, 34, 54, 61, 94]. Most of them focus on failures involving physical components such as fiber-optic cable disruption, interface faults, system crash, etc., in ISP networks where network components are not necessarily under a single administration. At a high level, these fault localization approaches have a trade-off between accuracy [62] and computation overhead [23, 53].

ISP networks. SCORE [62] used risk models and greedy approximation based fault localization algorithms to identify faulty components in ISP networks. The failure data collected from the network can have either noise or lack all necessary information, thus it requires ad hoc thresholds to overcome the negative impact on the accuracy of the proposed algorithms. Unlike SCORE, sherlock [23, 53] captured the noise or missing data with probabilistic dependencies between the graph nodes, and then inference mechanisms such as bayesian [93] or belief propagation [94] are applied to identify the most likely root causes for network-wide faults. [95] provide an extensive study of these inference based approaches. A hybrid approach, Gestalt [72], leverages both approaches (greedy and bayesian) and offers an accurate and lower overhead solution for specific applications.

On the contrary, Scout focus is on fault localization of the network policy configuration process in multi-tenant data center networks driven by network policies. Thus, the context of Scout is quite different from that of these prior works. Also, the information and properties captured by the models address the problems that are different from ISP networks. Unlike Scout, many of these approaches do not scale well beyond a few nodes (more than 50) in the network [81].

Enterprise and data center networks. As mentioned before in Section 3.1, many recent works [58, 66, 59] collect data plane snapshots, checks the consistency between the control plane policy and data plane configurations (*e.g.*, per-switch routing rules, ACL rules, etc). While, these works goal is to detect the inconsistency between the control plane policy and the data plane configurations, Scout goal is to localize, and find the root cause of such inconsistency. In other words, Scout takes output of these systems as one of the inputs, analyzes, and helps the operators in two aspects: (1) nails down to the part of the policy that the operator should look at to fix the observed failures (or inconsistency); and (2) infers the most likely physical-level root cause for the inconsistency.

Most recent work on network provenance systems [111, 27] keeps track of events associated with packets and rules, while Scout only compares network policies with actual rules deployed in the network.

Other works [82, 15, 41, 69, 92, 39, 56] focus on the automation of conflict-free, error-free composition and deployment of network policies to reduce the likelihood of network problem occurrences. While these frameworks are greatly useful in managing network policies, it is hard to completely shield their management plane from failure, which may cause the inconsistency between the policies and the actual network state. Scout can identify the impacted network policies. Thus, Scout can be useful in reinstating the network policies when these frameworks may not work correctly.

3.4 Summary

This chapter presented related work on network monitoring, network debugging, and network fault localization. To summarize, networks are complex, and the tools to debug these networks are even more complex. In-network monitoring approaches offer network visibility, but often limited by available switch data plane resources (*e.g.*, cpu, memory). So, they incorporate increasingly complex in-network techniques which require more network bandwidth resources or make it hard to debug the network prob-

lems. On the contrary, end-host based monitoring loses the benefits offered by in-network monitoring approaches. Existing work on network verification focused on detecting the inconsistency between intent and actual network behavior. In contrast, this thesis focuses on localizing and finding the cause that creates the inconsistency.

Chapter 4

CherryPick: Tracing Packet Trajectory in Software-Defined Datacenter Networks

4.1 Introduction

A particularly interesting problem in SDN debugging is to be able to reason about flow of traffic (*e.g.*, tracing individual packet trajectories) through the network [47, 59, 60, 66, 73, 110]. Such a functionality enables measuring network traffic matrix, detecting traffic anomalies caused by congestion [36], localizing network failures [59, 60, 66], or simply ensuring that forwarding behavior at the data plane matches the policies at the control plane [47]. Note that existing tools for tracing packet trajectories can use one of the two broad approaches. On the one hand, tools like NetSight [47] support a wide range of queries using after-the-fact analysis, but also incur large "out-of-band" data collection overhead. In contrast, "in-band" tools (*e.g.*, PathQuery [73] and PathletTracer [110]) significantly reduce data collection overhead at the cost of supporting a narrower range of queries.

We present CherryPick, a scalable, yet simple "in-band" technique for tracing packet trajectories in SDN-enabled datacenter networks. CherryPick is designed with the goal of minimizing two data plane resources: the number of switch flow rules and the packet header space. Indeed, existing approaches to tracing packet trajectories in SDN trade off one of these resources to minimize the other. At one end of the spectrum is the most naïve approach of assigning each network link a unique identifier and switches embedding the identifier into the packet header during the forwarding pro-

	CherryPick			PathletTracer			Naïve		
Path length	**4**	**6**	**8**	**4**	**6**	**8**	**4**	**6**	**8**
#Flow rules	48	48	48	576	1.2M	1.7B	48	48	48
#Header bits	11	22	33	10	21	31	24	36	48

Table 4.1: CherryPick achieves the best of the two existing techniques for tracing packet trajectories — the minimal number of switch flow rules required by the naïve approach and close to the minimal packet header space required by PathletTracer [110]. These results are for a 48-ary fat-tree topology; M and B stand for million and billion respectively. See §4.3 for details.

cess. This minimizes the number of switch flow rules required, but has high packet header space overhead especially when the packets traverse along non-shortest paths (*e.g.*, due to failures along the shortest path). At the other end are techniques like PathletTracer [110] that aim to minimize the packet header space, but end up requiring a large number of switch flow rules (§4.3); PathQuery [73] acknowledges a similar limitation in terms of switch resources.

CherryPick minimizes the number of switch flow rules required to trace packet trajectories by building upon the naïve approach — each network link is assigned a unique identifier and switches simply embed the identifier into the packet header during the forwarding process. However, in contrast to the naïve approach, CherryPick minimizes the packet header space by *selectively picking a minimum number of essential links to represent an end-to-end path*. By exploiting the fact that datacenter network topologies are often well-structured, CherryPick requires packet header space comparable to state-of-the-art solutions [110], while retaining the minimal switch flow rule requirement of the naïve approach. For instance, Table 4.1 compares the number of switch flow rules and the packet header space required by CherryPick against the above two approaches for a 48-ary fat-tree topology.

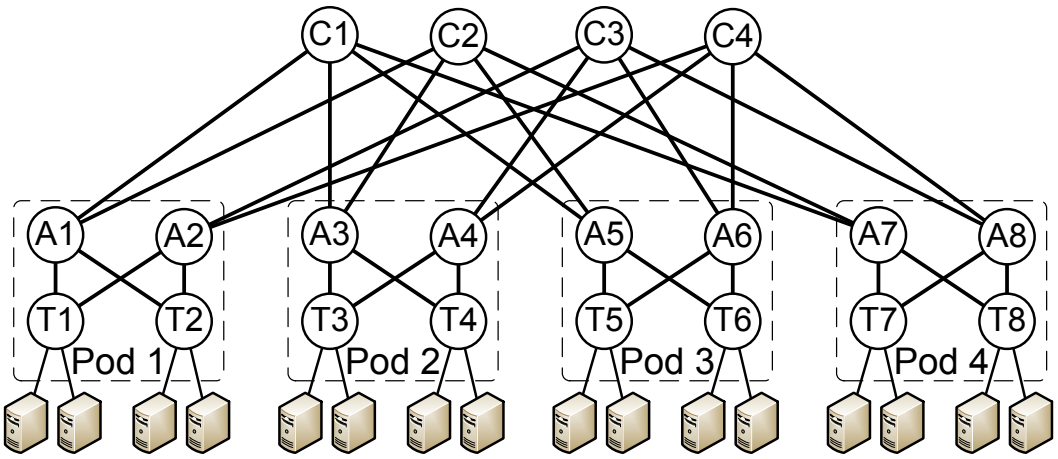


Figure 4.1: A 4-ary fat-tree topology.

In summary, this chapter makes three contributions:

- We design CherryPick, a simple and scalable packet trajectory tracing technique for SDN-enabled datacenter networks. The main idea in CherryPick is to exploit the structure in datacenter network topologies to minimize number of switch flow rules and packet header space required to trace packet trajectories. We apply CherryPick to a fat-tree topology in this chapter to demonstrate the benefits of the technique (§4.2).

- We show that CherryPick can trace all 4- and 6-hop paths in an up-to 72-ary fat-tree with no hardware modification by using IEEE 802.1ad double-tagging (§4.2).

- We evaluate CherryPick over a 48-ary fat-tree topology. Our results show that CherryPick requires minimal number of switch flow rules while using packet header space close to state-of-the-art techniques (§4.3).

4.2 CherryPick

This section describes the CherryPick design in detail with a focus on enabling L2/L3 packet trajectory tracing in a fat-tree network topology.

4.2.1 Preliminaries

The fat-tree topology. A k-ary fat-tree topology contains three layers of k-port switches: Top-of-Rack (ToR), aggregate (Agg) and core. A 4-ary fat-tree topology is presented in Figure 4.1 (ToR switches are nodes with letter T, Agg with letter A and core with letter C). The topology consists of k pods, each of which has a layer of $k/2$ ToR switches

and a layer of $k/2$ Agg switches. $k/2$ ports of each ToR switch are directly connected to servers and each of remaining $k/2$ ports connected to $k/2$ Agg switches. The remaining $k/2$ ports of each Agg switch are connected to $k/2$ core switches. There are a total of $k^2/4$ core switches where port i on each core switch is connected to pod i.

To ease the following discussion, links present in the topology are grouped into two categories: *i) intra-pod link* and *ii) pod-core link*. An intra-pod link is one connecting a ToR and an Agg switch, whereas a pod-core link is one connecting an Agg and a core switch. Note that there are $k^2/4$ intra-pod links within a pod and a total of $k^3/4$ pod-core links. In addition, *affinity core segment*, or affinity segment, is a group of core switches that can be directly reached by each Agg switch at a particular position in each pod. In Figure 4.1, $C1$ and $C2$ in affinity segment 1 are directly reached by Agg switches at the lefthand side in each pod (*i.e.*, $A1$, $A3$, $A5$, and $A7$).

Routing along non-shortest paths. While datacenter networks typically use shortest path routing, packets can traverse along non-shortest paths due to several reasons. First, node and/or link failures can enforce routing of packets along non-shortest paths [30, 76, 104]. Consider, for instance, the topology in Figure 4.1 where a packet is being routed between Src and Dst along the shortest path Src → T1 → A1 → C1 → A3 → T3 → Dst. If link C1 → A3 fails upon the packet arrival, the packet will be forced to traverse a non-shortest path. Second, recently proposed techniques reroute packets along alternate (potentially non-shortest) paths [83, 100, 108] to avoid congested links. Finally, misconfiguration may also create similar situations. The term "detour" is defined to collectively refer to situations that force packets to traverse along a non-shortest path. The ability to be able to trace packet trajectories in case of packet detours is, thus, important since this may reveal network failures and/or misconfiguration issues.

However, packet detours complicate trajectory tracing due to the vastly increased number of possible paths even in a medium-size datacenter. For instance, given a 48-ary fat-tree topology, the number of shortest paths (*i.e.*, 4-hop paths) between a host pair in different pods is just 576. On the other hand, there exist almost 1.31 million 6-hop paths for the same host pair. As shown in §4.3, techniques that work well for tracing shortest paths [110] do not necessarily scale to the case of packet detours.

Detour model. CherryPick is designed to work with arbitrary routing schemes. However, to ease the discussion, this work focused on a simplified detour model where once the packet is forwarded by the Agg switch in the source pod, it does not traverse

any ToR switch other than the ones in the destination pod. For instance, in Figure 4.1, consider a packet traversing from Src in Pod 1 to Dst in Pod 2. Under this simplification, the packet visits none of ToR switches $T5$, $T6$, $T7$ and $T8$ once it leaves $T1$. Of course, in practice, packets may visit ToR switches in non-destination pods. In such a case, packet following a path longer than 6 hops, which is not common, is forwarded to controller. Then, controller verify routing policies, and take the necessary actions (*e.g.* re-injects the packet into network, raises the alarm).

4.2.2 Overview of CherryPick

This section gives a high-level description of CherryPick design. Consider the naïve approach that embeds in the packet header an identifier (ID) for each link that the packet traverses. For a 48-port switch, it is easy to see that this approach requires $\lceil \log(48) \rceil = 6$ bits to represent each link. Indeed, the header space requirement for this naïve approach is far higher than the theoretical bound, $\log(P)$ bits, where P is the number of paths between any source-destination pair. For tracing 4-hop paths, the naïve scheme requires 24 bits whereas only 10 bits are theoretically required since P is 576 ($P = k^2/4$).

CherryPick builds upon the observation that data center network topologies are often well-structured and allow reconstructing the end-to-end path without actually storing each link as the packet traverses. CherryPick, thus, *cherry-picks a minimum number of links essential to represent an end-to-end path.* For instance, for the fat-tree topology, it suffices to store the ID of the pod-core link to reconstruct any 4-hop path. To handle a longer path, in addition to picking a pod-core link, CherryPick selects one extra link every additional 2 hops. Hence, tracing any n-hop path ($n \geq 4$) requires only $(n-4)/2+1$ links worth of header space[1]. However, the cherry-picking of links makes it impossible to use local port IDs as link identifiers and using global link IDs requires a large number of bits per ID due to the sheer number of links in the topology. CherryPick, thus, assigns link IDs in a manner that each link ID requires fewer bits than a global ID and that the end-to-end path between any source-destination pair can be reconstructed without any ambiguity. Section 4.2.3, discuss how CherryPick reconstructs end-to-end paths using cherry-picking the links along with a careful assignment of non-global link IDs.

CherryPick leverages VLAN tagging to embed chosen links in the packet header.

[1] A 2-hop path is the shortest path between servers in the same pod, for which CherryPick simply picks one intra-pod link at Agg.

Port	IP Src	IP Dst	Action
3	10.*pod*.0.0/16	10.*pod*.0.0/16	`write_metadata`: 0x0/0x0
4	10.*pod*.0.0/16	10.*pod*.0.0/16	`write_metadata`: 0x0/0x0
3	10.*pod*.0.0/16	*	`push_vlan_id`: linkID(3)
4	10.*pod*.0.0/16	*	`push_vlan_id`: linkID(4)

(a) ToR switch

Port	IP Src	IP Dst	Action
1	*	10.*pod*.0.0/16	`push_vlan_id`: linkID(1)
2	*	10.*pod*.0.0/16	`push_vlan_id`: linkID(2)

(b) Aggregate switch

Port	IP Src	IP Dst	Action
1	*	*	`push_vlan_id`: linkID(1)
2	*	*	`push_vlan_id`: linkID(2)
3	*	*	`push_vlan_id`: linkID(3)
4	*	*	`push_vlan_id`: linkID(4)

(c) Core switch

Figure 4.2: OpenFlow table entries at each switch layer for the 4-ary fat-tree. In this example, the address follows the form of 10.*pod*.*switch*.*host*, where *pod* denotes pod number (where switch is), *switch* denotes position of switch in the pod, *host* denotes the sequential ID of each host. These entries are stored in a separate table which will be placed at the beginning of a table pipeline. In (a), 3 and 4 are port numbers connected to Agg layer. In (b), 1 and 2 are port numbers connected to ToR layer.

While the OpenFlow standard [79] does not dictate how many VLAN tags can be inserted in the header, typically commodity SDN switches only support IEEE 802.1ad double-tagging. With two tags, CherryPick can keep track of all 1.31 million 6-hop paths in the 48-ary fat-tree while keeping switch flow memory overhead low. As hardware programmability in SDN switch increases [25, 51], we expect that the issue raised by the limited number of tags can be mitigated.

4.2.3 Design

This section discuss CherryPick design in depth. The focus is on three aspects of the design: (1) selectively picking links that allow reconstructing the end-to-end path and configuring switch rules to enable link picking; (2) careful assignment of link IDs to further minimize the packet header space; and (3) the path reconstruction process using the link IDs embedded in the packet header.

Picking links. Consider a packet arriving at an input port. The link attached to the input port is called as *ingress link*. For each packet, every switch has a simple link selection mechanism that can be easily converted into OpenFlow rules. If the packet matches one of the rules, the ingress link is picked; and its ID is embedded into the packet using VLAN tag.

The following describes the link selection mechanism at each switch level, and Figure 4.2 shows flow rules derived from the mechanisms:

- *ToR:* If a ToR switch receives the packet from an Agg switch and if the packet's source belongs to the same pod, the switch picks the ingress link connected to the Agg switch that forwarded the packet. However, if both source and destination are in the same pod, the switch ignores the ingress link (we use `write_metadata` command to implement the "do nothing" operation). For all other cases, no link is picked.
- *Aggregate:* If an Agg switch receives the packet from a ToR switch and if the packet's destination is in the same pod, the ingress link is chosen. Otherwise, no link is picked.
- *Core:* Core switch always picks the ingress link.

Using the above set of rules, CherryPick selects the minimum number of links required to reconstruct the end-to-end path of any packet. For ease of exposition, four examples are present in Figure 4.3. First, Figure 4.3(a) illustrates the baseline 4-hop scenario. In this scenario, core switch $C2$ only picks an ingress link and other switches (*e.g.*, $A1$, $A3$ and $T3$) do nothing. In case of one detour at source pod (Figure 4.3(b)), $T2$ and $C2$ will choose each ingress link of a packet while others not. A similar picking process is undertaken in case of one detour at destination pod (Figure 4.3(c)). When one detour occurs between aggregate and core switch (Figure 4.3(d)), only core switches which see the detoured packet pick the ingress links.

Link ID assignment. CherryPick assigns IDs for intra-pod links and pod-core links

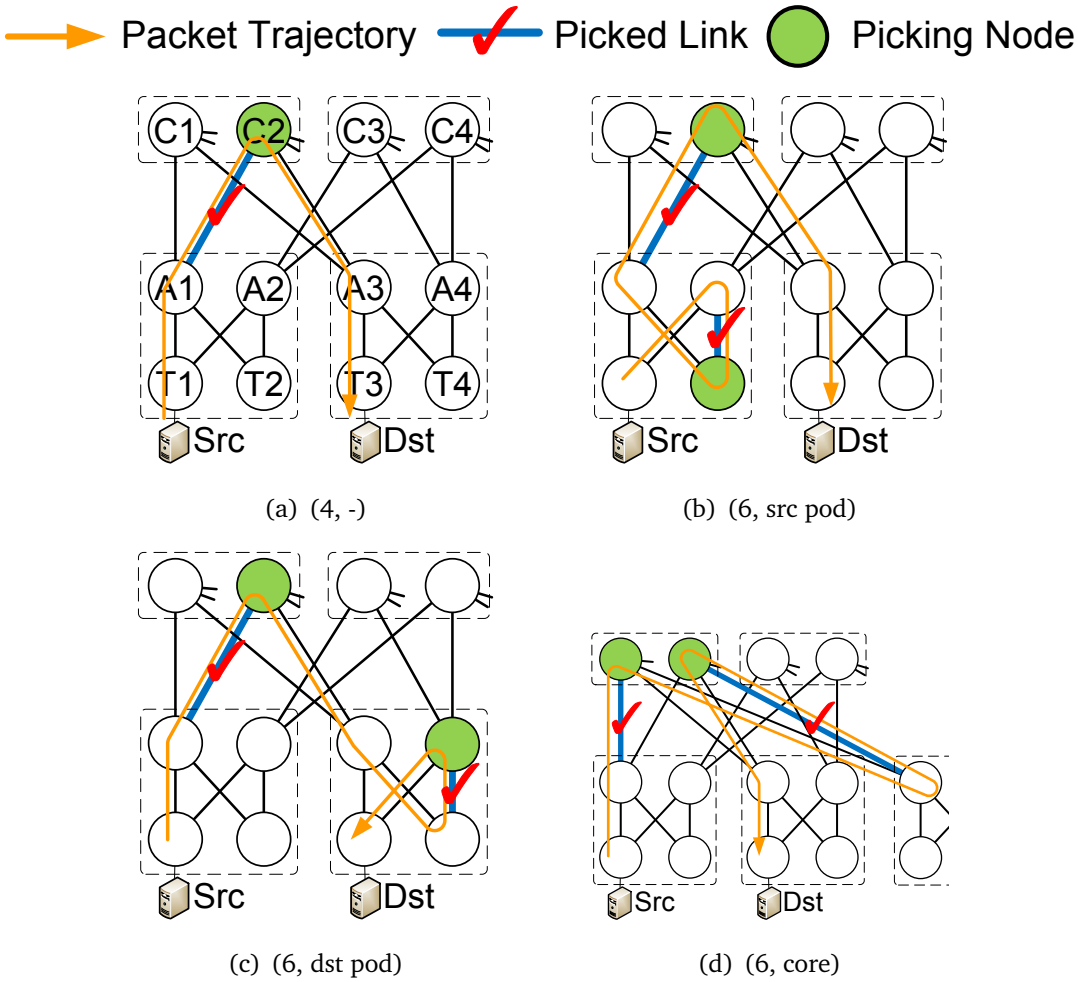
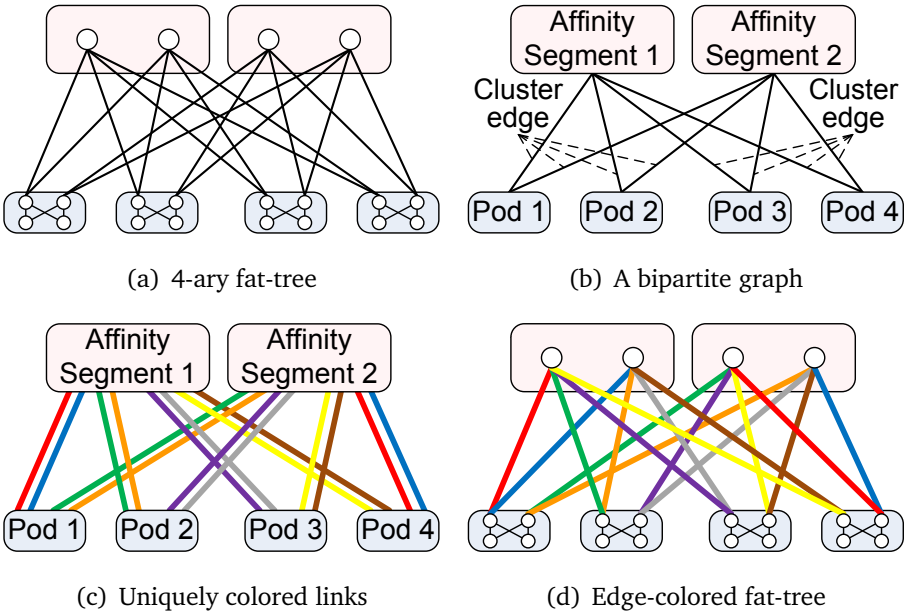


Figure 4.3: An illustration of link *cherry-picking* in CherryPick. (x, y) means x number of hops and detour at location y.

separately.

i) Intra-pod links: Since pods are separated by core switches, the same set of links IDs is used across all pods. Since each pod has $(k/2)^2$ intra-pod links, we need $(k/2)^2$ IDs. Links at the same position across pods have the same link ID.

ii) Pod-core links: Assigning IDs to pod-core links such that the correct end-to-end path can be reconstructed while minimizing the number of bits required to represent the ID is non-trivial. Indeed, one way to assign IDs is to consider all pod-core links, and assign each link a unique ID. However, there are $k^3/4$ pod-core links and such an approach would require $\lceil \log(k^3/4) \rceil$ many bits per link ID. It can be significantly reduced by viewing the problem as an edge coloring problem of a complete bipartite graph. In this problem, the goal is to assign colors to the edges of the graph such that adjacent edges of a vertex have different colors. Edge-coloring a complete bipartite graph requires α different colors where α is the graph's maximum degree.



(a) 4-ary fat-tree (b) A bipartite graph

(c) Uniquely colored links (d) Edge-colored fat-tree

Figure 4.4: Edge-coloring pod-core links in a 4-ary fat-tree.

To view a fat-tree as a complete bipartite graph with two disjoint sets, a pod is treated as a vertex in the first set and an affinity core segment as a vertex in the second set (compare Figures 4.4(a) and 4.4(b)). Then, edges (links) from an Agg switch are grouped to an affinity core segment and supersede them by one edge named as *cluster edge* (Figure 4.4(b)). Since the maximum degree is k (i.e., the number of cluster edges at an affinity core segment), we need k different colors to edge-color this bipartite graph. Note that one cluster edge is a collection of all $k/2$ links. Therefore, we need k different color sets such that each color set has $k/2$ different colors and any two color sets are disjoint (Figure 4.4(c)). Thus in total $k(k/2)$ different colors are required. The actual color assignment is done by applying a near-linear time algorithm [31]. Figure 4.4(d) shows an accurate color allocation.

Putting it together, the number of unique IDs required is $3k^2/4$ ($(k/2)^2$ for the intra-pod links and $k(k/2)$ for the pod-core links). Thus, CherryPick requires a total of $\lceil \log(3k^2/4) \rceil$ bits to represent each link. For 48-ary and 72-ary fat-tree, CherryPick requires just 11 and 12 bits respectively to represent each link. CherryPick can thus support an up-to 72-ary fat-tree topology using the 12 bits available in VLAN tag.

Path reconstruction. With this scheme, when a packet reaches its destination, it contains a minimum set of link IDs necessary to reconstruct a complete path. To keep it simple, suppose that those link IDs in the header are extracted and stored in the order that they were selected. At the destination, a network topology graph is given and each link in the graph is annotated with one ID. Path reconstruction process begins from

source ToR switch in the graph. Initially, a list S contains the source ToR switch. Until all the link IDs are consumed, the following steps are executed: i) take one link ID (say, l) from the ID list and find, from the topology graph, a link whose ID is l (if l is in the pod ID space, search for the link in either source or destination pod depending on whether pod-core link is consumed; otherwise, search for it in the current affinity segment); ii) identify two switches (say, s_a and s_b) that form the link; iii) out of the two, choose one (say, s_a) closer to the switch (say, s_r) that was most recently added to S; iv) find a shortest path (which is simple because it is either 1-hop or 2-hop path) between s_a and s_r and add all intermediate nodes (those closer to s_r first) and s_a later to S; v) add the remaining switch s_b to S. After all link IDs are consumed, we add to S the switches that form a shortest path from the switch included last in S to the destination ToR switch. Finally, we obtain a complete path by enumerating switches in S.

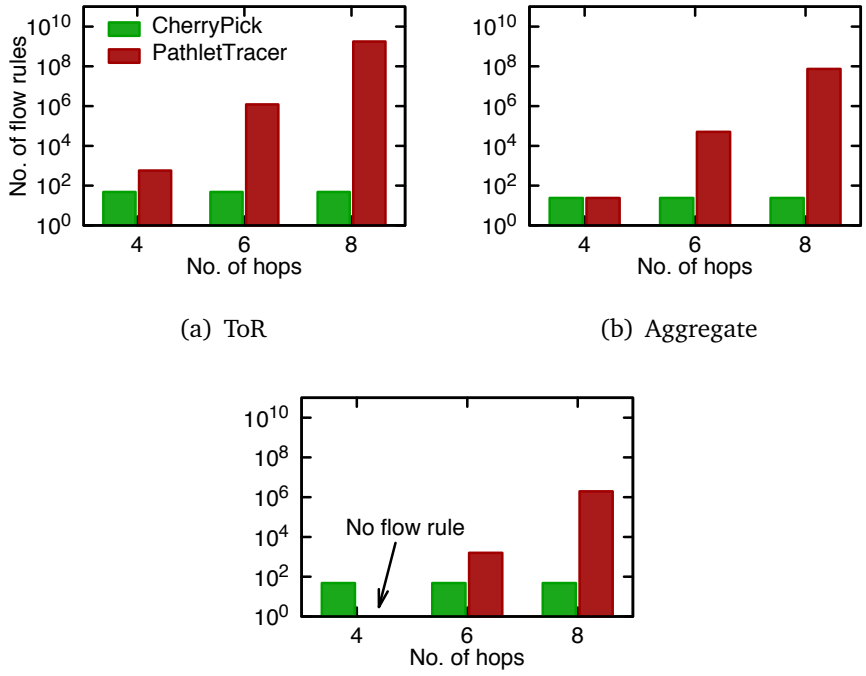
4.3 Evaluation

This section presents preliminary evaluation results for CherryPick, and compare it against PathletTracer [110] over a 48-ary fat-tree topology. The two schemes are evaluated in terms of number of switch flow rules (§4.3.1), packet header space (§4.3.2) and end-host resources (§4.3.3) required for tracing packet trajectories. While preliminary, evaluation suggests that:

- CherryPick requires minimal number of switch flow rules to trace packet trajectories. In particular, CherryPick requires as many rules as the *number of ports per switch*. In contrast, PathletTracer requires number of switch flow rules linear in *the number of paths* that the switch belongs to. For tracing 6-hop paths in a 48-ary fat-tree topology, for instance, CherryPick requires three orders of magnitude fewer switch rules than PathletTracer while supporting similar functionality.

- CherryPick requires packet header space close to state-of-the-art techniques. Compared to PathletTracer, CherryPick trades off slightly higher packet header space requirements for significantly improved scalability in terms of number of switch flow rules required to trace packet trajectories.

- CherryPick requires minimal resources at the end hosts for tracing packet trajectories. In particular, CherryPick requires as much as three orders of magnitude fewer entries at the destination when compared to PathletTracer for tracing 6-hop paths on a 48-ary fat-tree topology.



(a) ToR

(b) Aggregate

(c) Core

Figure 4.5: CherryPick requires number of switch flow rules comparable to PathletTracer for tracing shortest paths. However, for tracing non-shortest paths (*e.g.*, packets may traverse such paths in case of failures), the number of switch flow rules required by PathletTracer increases super-linearly. In contrast, the number of switch flow rules required by CherryPick remains constant.

4.3.1 Switch flow rules

CherryPick, for any given switch, requires as many flow rules as the number of ports at that switch. In contrast, for any given switch, PathletTracer requires as many switch flow rules as the number of paths that contain that switch. Since the latter depends on the layer at which the switch resides, number of switch flow rules are plotted for CherryPick and PathletTracer across each layer separately (see Figure 4.5).

It is observed that for tracing shortest paths only, the number of switch flow rules required by PathletTracer is comparable to those required by CherryPick. However, if one desires to trace non-shortest paths (*e.g.*, in case of failures), the number of switch flow requirement of PathletTracer grows super-linearly with the number of hops constituting the paths[2]. For tracing 6-hop paths, for instance, PathletTracer requires over a million rules on ToR switch, tens of thousands of rules on Aggregate switch, and

[2]The number of switch flow rules required by PathletTracer could be reduced by tracing "pathlets" (a sub-path of an end-to-end path) at the cost of coarser tracing compared to CherryPick.

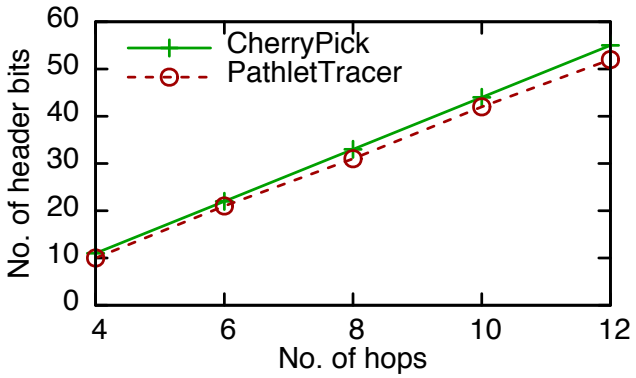


Figure 4.6: CherryPick requires packet header space comparable to PathletTracer for tracing packet trajectories. In particular, for tracing 6-hop paths in a 48-ary fat-tree topology, CherryPick requires 22 bits while PathletTracer requires 21 bits worth of header space.

thousands of rules on Core switch. This means that PathletTracer would not scale well for path tracing at L3 layer because packets may follow non-shortest paths. In contrast, CherryPick requires a small number of switch flow rules independent of path length.

4.3.2 Packet header space

We now evaluate the number of bits in the packet header required to trace packet trajectories (see Figure 4.6). Recall from §4.2 that to enable tracing of any n-hop path in a fat-tree topology, CherryPick requires embedding $(n-4)/2+1$ links in the packet header. The number of bits required to uniquely represent each link increases logarithmically with number of ports per switch; for a 48-ary fat-tree topology, each link requires 11 bits worth of space. PathletTracer requires $\log(P)$ bits worth of header space, where P is the number of paths between the source and the destination. We observe that CherryPick requires slightly higher packet header space than PathletTracer (especially for longer paths); however, as discussed earlier, CherryPick trades off slightly higher header space requirement with significantly improved scalability in terms of switch flow rules.

4.3.3 End host resources

Finally, performance of CherryPick is compared against PathletTracer in terms of the resource requirements at the end host. Each of the two schemes stores certain entries at the end host to trace the packet trajectory using the information carried in the packet header. Processing the packet header requires relatively simple lookups into the stored entries for both schemes, which is a lightweight operation. We hence only quantify the

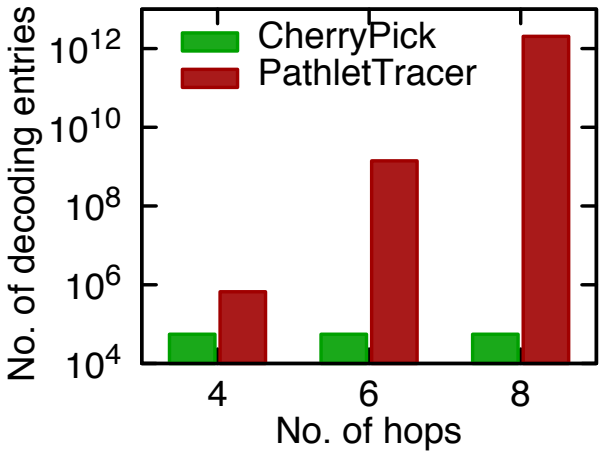


Figure 4.7: CherryPick requires significantly fewer entries than PathletTracer at each end host compared to trace packet trajectories. In particular, for tracing 6-hop paths in a 48-ary fat-tree topology, CherryPick requires less than 1MB worth of entries while PathletTracer requires approximately 12GB worth of entries at each end host.

number of entries required for both schemes.

CherryPick stores, at each destination, the entire set of network links, each annotated with a unique identifier. PathletTracer requires storing a "codebook", where each entry is a code assigned to each of the unique path. For fair comparison, we assume that individual hosts in PathletTracer store an equal-sized subset of the codebook that is only relevant to them.

Figure 4.7 shows that PathletTracer needs to store non-trivial number of entries if non-shortest paths need to be traced. For instance, PathletTracer requires more than 10^9 entries *at each end host* to trace 6-hop paths, which translates to approximately 12GB worth of entries as each entry is about 12 bytes long. As discussed earlier, this overhead for PathletTracer could be reduced at the cost of tracing at coarser granularities. In contrast, since CherryPick stores only the set of links constituting the network, it requires a small, fixed number of entries (∼56K entries per host for a 48-ary fat-tree topology).

4.4 Summary

This chapter presents CherryPick, a simple yet scalable technique for tracing packet trajectories in SDN-enabled datacenter networks. The core idea in CherryPick is that structure in datacenter network topologies enable reconstructing end-to-end paths using a few essential links. To that end, CherryPick "cherry-picks" a subset of links along the packet trajectory and embeds them into the packet header. CherryPick is applied to a fat-tree topology in this chapter and showed that it requires minimal switch flow rules to enable tracing packet trajectories, while requiring packet header space close to state-of-the-art techniques.

Chapter 5

Simplifying Data center Network Debugging with PathDump

5.1 Introduction

Data center networks are complex. Networks are built based on scale-out topologies, that consists of tens of thousands of network devices, and aggregate traffic can easily exceed 100 Tbps [112]. Such a large-scale network also involves complex dependencies [112, 44] among these many network components. Despite this complexity, users continue to demand very high performance for their applications. Performance degradation of these services, even for short period of time, can cause millions of dollars revenue loss. For instance, Amazon reports [1] that every 100 ms delay costs 1% of its e-commerce revenue. The network becomes even more complex, with enforcement of policies like security and isolation. Increasingly, many data centers are using programmable hardware [10]. In these data centers, network switches can be dynamically programmed to achieve various goals. This programmability adds to the complexity of networks.

In such a large, complex and dynamic environment, network problems in the data plane are inevitable. For example, failures or bugs can trigger forwarding rule updates and temporarily create loops in the network. A faulty interface drops packets at random and does not update respective counters, thus creating a silent random packet drop problem [112]. In the presence of such data plane problems, what is happening in the network may not match with the network operator's intention, which could result in performance degradation [109, 112] or even network outages [59]. Ideally, the operator wants to debug and fix such network problems in real time using debugging tools. To

this end, there have been lots of recent efforts in building debugging tools [36, 107, 96, 112, 75, 66, 60, 47, 63].

Network debuggers are even more complex. Over the years, the class of network problems supported by existing network debugging tools has grown significantly. Accordingly, the tools have incorporated increasingly complex techniques — collecting data plane snapshots [66, 60, 59, 58, 40], collecting per-packet per-switch packet logs [112, 77, 47, 86, 107, 96, 36, 13], selective mirroring of packets [112], packet sampling [107, 96, 36, 13], using active probe packets [16, 109, 112], replaying traffic [103], dynamic rule installation [75, 46] — and this list barely scratches the surface of all the sophisticated techniques used in existing network debugging tools.

Our goal is not add to the impressive collection of network debugging techniques. Instead, we ask whether there are a non-trivial number of network debugging problems that could obviate the need for sophisticated in-network techniques. Thus, our focus is *not* to try to beat existing tools in either generality or in performance, but to help them focus on a smaller subset of nails (debugging problems) that we need a hammer (debugging techniques) for. The hope is that by focusing on a smaller subset of problems, the already complex networks[1] and the debugging tools for these networks can be kept as simple as possible.

PathDump design. We present PathDump, an end-host based network debugger that demonstrates our approach by enabling a large class of debugging problems with minimal in-network functionality. PathDump design is based on tracing packet trajectories and comprises of the following:

- Switches are simple; they neither require dynamic rule updates nor perform any packet sampling or mirroring. In addition to its usual operations, a switch checks for a condition before forwarding a packet; if the condition is met, the switch embeds its identifier into the packet header (*e.g.*, with VLAN tags).
- An edge device, upon receiving a packet, records the list of switch identifiers in the packet header on a local storage and query engine; a number of entries stored in the engine (used for debugging purposes) are also updated based on these switch identifiers.
- Entries at each edge device can be used to trigger and debug anomalous network behavior; a query server can also slice-and-dice entries across multiple edge

[1] as eloquently argued in [112]; in fact, our question about simpler networks and debugging tools was initially motivated by the arguments about network complexity in [112].

devices in a distributed manner (*e.g.*, for debugging functionalities that require correlating entries across flows).

PathDump's design, by requiring minimal in-network functionality, presents several benefits as well as raises a number of interesting challenges. The benefits are rather straightforward. PathDump not only requires minimal functionality to be implemented at switches, but also uses minimal switch resources; thus, the limited switch resources [33, 70] can be utilized for exactly those tasks that necessitate an in-network implementation[2]. PathDump also preserves flow-level locality — information about all packets in the same flow is recorded and analyzed on the same end-host. Since PathDump requires little or no data transfer in addition to network traffic, it also alleviates the bandwidth overheads of several existing in-network debuggers [47, 86, 112].

PathDump challenges. PathDump resolves several challenges to achieve the above benefits. First challenge is regarding generality — what class of network problems can PathDump debug with minimal support from network switches? To get a relatively concrete answer in light of numerous possible network debugging problems, we examined all the problems discussed in several recent works [47, 51, 75, 112] (see Table 5.2). Interestingly, we find that PathDump can already support more than 85% of these problems. For some problems, network support seems necessary; however, we show that PathDump can help "pinpoint" these problems to a small part of the network. We discuss the design, implementation and evaluation of PathDump for the supported functionality in §5.2.3 and §5.4.

PathDump also resolves the challenge of packets not reaching the edge devices (*e.g.*, due to packet drops or routing loops). A priori, it may seem obvious that PathDump must not be able to debug such problems without significant support from network switches. PathDump resolves the packet drop problem by exploiting the fact that datacenters typically perform load balancing (using ECMP or packet spraying [35]); specifically, we show that the difference between number of packets traversing along multiple paths allows identifying spurious packet drops. PathDump can in fact debug routing loops by leveraging the fact that commodity SDN switches recognize only two VLAN

[2]As PathDump matures, we envision it to incorporate (potentially simpler than existing) in-network techniques for debugging problems that necessitate an in-network implementation. As network switches evolve to provide more powerful functionalities (*e.g.*, on-chip sampling) and/or larger resource pools, partitioning the debugging functionality between the edge devices and the network elements will still be useful to enable capturing network problems at per-packet granularity — a goal that is desirable and yet, infeasible to achieve using today's resources. Existing in-network tools that claim to achieve per-packet granularity (*e.g.*, Everflow [112]) have to resort to sampling to overcome scalability issues and thus, fail to achieve per-packet granularity.

tags in hardware and processing more than two tags involves switch CPU (§5.4.5).

Finally, PathDump carefully optimizes the use of data plane resources (*e.g.*, switch rules and packet header space) and end-host resources (*e.g.*, CPU and memory). PathDump extends our prior work, CherryPick (Chapter 4), for per-packet trajectory tracing using minimal data plane resources. For end-host resources, we evaluate PathDump over a wide range of network debugging problems across a variety of network testbeds comprising of commodity network switches and end-hosts; our evaluation shows that PathDump requires minimal CPU and memory at end-hosts while enabling network debugging over fine-grained time scales.

PathDump contributions. Overall, PathDump makes three main contributions:

- Make a case for partitioning the network debugging functionality between the edge devices and the network elements, with the goal of keeping network switches free from complex operations like per-packet log generation, dynamic rule updates, packet sampling, packet mirroring, etc.
- Design and implementation of PathDump[3], a network debugger that demonstrates that it is possible to support a large class of network management and debugging problems with minimal support from network switches.
- Evaluation of PathDump over operational network testbeds comprising of commodity network hardware demonstrating that PathDump can debug network events at fine-grained time-scales with minimal data plane and end-host resources.

This chapter is organized as follows: Section 5.2 gives overview of PathDump, Section 5.3 has details of PathDump's implementation, Section 5.4 discusses various debugging applications enabled using PathDump, and Section 5.5 presents the PathDump system's evaluation.

5.2 Overview

We start with an overview of the PathDump interface (§5.2.1) and PathDump design (§5.2.2). We then provide several examples of using the PathDump interface for debugging network problems (§5.2.3, §5.2.4).

[3]Available at: https://github.com/PathDump.

Host API	Description
`getFlows(linkID, timeRange)`	Return list of `flows` that traverse `linkID` during specified `timeRange`.
`getPaths(flowID, linkID, timeRange)`	Return list of `Paths` that include `linkID`, and are traversed by `flowID` during specified `timeRange`.
`getCount(Flow, timeRange)`	Return packet and byte counts of a `flow` within a specified `timeRange`.
`getDuration(Flow, timeRange)`	Return the duration of a `flow` within a specified `timeRange`.
`getPoorTCPFlows(Threshold)`	Return the `flowIDs` for which `protocol = TCP` and the number of consecutive packet retransmissions exceeds a threshold.
`Alarm(flowID, Reason, Paths)`	Raise an alarm regarding `flowID` with a reason code (e.g., TCP performance alert (`POOR_PERF`)), and corresponding list of `Paths`.
Controller API	**Description**
`execute(List⟨HostID⟩,Query)`	Execute a `Query` once at each host specified in list of `HostIDs`; a `Query` could be any of the ones from Host API.
`install(List⟨HostID⟩,Query,Period)`	Install a `Query` at each host specified in list of `HostIDs` to be executed at regular `Periods`. If the `Period` is not set, the query execution is triggered by a new event (*e.g.*, receiving a packet).
`uninstall(List⟨HostID⟩,Query)`	Uninstall a `Query` from each host specified in list of `HostIDs`

Table 5.1: PathDump Interface. See §5.2.1 for definitions and discussion.

5.2.1 Interface

PathDump exposes a simple interface for network debugging; see Table 5.1. We assume that each switch and host has a unique ID. We use the following definitions:

- A `linkID` is a pair of adjacent `switchIDs` ($\langle S_i, S_j \rangle$);
- A `Path` is a list of `switchIDs` ($\langle S_i, S_j, \ldots \rangle$);
- A `flowID` is the usual 5-tuple ($\langle$srcIP, dstIP, srcPort, dstPort, protocol$\rangle$);
- A `Flow` is a ($\langle$flowID, Path$\rangle$) pair; this will be useful for cases when packets from the same `flowID` may traverse along multiple `Paths`.
- A `timeRange` is a pair of `timestamps` ($\langle t_i, t_j \rangle$);

PathDump supports wildcard entries for `switchIDs` and `timestamps`. For instance, ($\langle \star, S_j \rangle$) is interpreted as all incoming links for switch S_j and ($\langle t_i, \star \rangle$) is interpreted as "since time t_i".

Note that each host exposes the host API in Table 5.1 and returns results for "local" flows, that is, for flows that have this host as their `dstIP`. To collect the results distributed across PathDump instances at individual end-hosts, the controller may use the controller API — to `execute` a query, to `install` a query for periodic execution, or to `uninstall` a query.

5.2.2 Design Overview

The central idea in PathDump is to trace packet trajectories. To achieve this, each switch embeds its switchID in the packet header before forwarding the packet. However, naïvely embedding all the switchIDs along the packet trajectory requires large packet header space, especially when packets may traverse a non-shortest path (*e.g.*, due to failures along the shortest path) [98]. PathDump uses the link sampling idea from CherryPick (see §4.2) to trace packet trajectories using commodity switches. However, CherryPick supports commonly used datacenter network topologies (*e.g.*, FatTree, VL2, etc.) and does not work with arbitrary topologies. Note that this limitation on supported network topologies is merely an artifact of today's hardware — as networks evolve to support larger packet header space, PathDump will support more general topologies without any modification in its design and implementation.

An edge device, upon receiving a packet, extracts the list of switchIDs in the packet header and records them on a local storage and query engine (along with associated

metadata, *e.g.*, flowID, timestamps, number of packets, number of bytes, etc.). Each edge device stores:

- A list of flow-level entries that are used for debugging purposes; these entries are updated upon each event (*e.g.*, receiving a packet).
- A static view of the datacenter network topology, including the statically assigned identifiers for each switch. This view provides PathDump with the "ground truth" about the network topology and packet paths.
- And, optionally, network configuration files specifying forwarding policies. These files are also used for monitoring and debugging purposes (*e.g.*, ensuring packet trajectories conform to specified forwarding policies). The operator may also push these configuration files to the end-hosts dynamically using the `Query` installation in controller API.

Finally, each edge device exposes the API in Table 5.1 for identifying, triggering and debugging anomalous network behavior. The entries stored in PathDump (within an edge device or across multiple edge devices) can be sliced-and-diced for implementing powerful debugging functionalities (*e.g.*, correlating entries across flows going to different edge devices). PathDump currently disregards packet headers after updating the entries to avoid latency and throughput bottlenecks in writing to persistent storage; extending PathDump to store and query at per-packet granularity remains an intriguing future direction.

5.2.3 Example applications

We now discuss several examples for network debugging applications using PathDump API.

Path conformance. Suppose the operator wants to check for policy violations on certain properties of the path taken by a particular `flowID` (*e.g.*, path length no more than 6, or packets must avoid `switchID`). Then, the controller may `install` the following query at the end-hosts:

```
Paths = getPaths(flowID, <*, *>, *)
for path in Paths:
  if len(path)>=6 or switchID in path:
    result.append (path)
if len(result) > 0:
  Alarm (flowID, PC_FAIL, result)
```

PathDump executes the query either upon each packet arrival, or periodically when a `Period` is specified in the query; an `Alarm()` is triggered upon each violation.

Load imbalance. Understanding why load balancing works poorly (as explained in Section 2.2) is of interest to operators because uneven traffic splits may cause severe congestion, thereby hurting throughput and latency performance. PathDump helps diagnose load imbalance problems, independent of the underlying scheme used for load balancing (*e.g.*, ECMP or packet spraying). The following example constructs flow size distribution for each of two egress ports (i.e., links) of interest on a particular switch:

```
result = {}; binsize = 10000
linkIDs = (l1, l2); tRange = (t1, t2)
for lID in linkIDs:
  flows = getFlows (lID, tRange)
  for flow in flows:
    (bytes, pkts) = getCount (flow, tRange)
    result[lID][bytes/binsize] += 1
return result
```

Through cross-comparison of the flow size distributions on the two egress ports, the operator can tell the degree of load imbalance. Even finer-grained diagnosis on load balancing is feasible; *e.g.*, when packet spraying is used, PathDump can identify whether or not the traffic of a flow in question is equally spread along various end-to-end paths. We demonstrate these use cases in §5.4.2.

Silent random packet drops. This network problem occurs when some faulty interface at switch drops packets at random (§2.2) without updating the discarded packet counters at respective interfaces. It is a critical network problem [112] and is often very challenging to localize.

PathDump allows a network operator to implement a localization algorithm such as MAX-COVERAGE [61]. The algorithm, as input, requires logs or observations on a network problem (that is, failure signatures). Using PathDump, a network operator can `install` a TCP performance monitoring query at the end-hosts for periodic monitoring (*e.g.*, period set to be 200 ms):

```
flowIDs = getPoorTCPFlows()
for flowID in flowIDs:
  Alarm (flowID, POOR_PERF, [])
```

Every time an alarm is triggered, the controller sends the respective end-host (by parsing flowID) the following query and collects failure signatures (that is, path(s) taken by the flow that suffers serious retransmissions):

```
flowID = (sIP, sPort, dIP, dPort, 6)
linkID = (*, *); tRange = (t1, *)
paths = getPaths (flowID, linkID, tRange)
return paths
```

The controller receives the query results (that is, paths that potentially include faulty links), locally stores them, and runs the MAX-COVERAGE algorithm implemented as only about 50 lines of Python code. This procedure repeats whenever a new alert comes up. As more path data of suffering TCP flows get accumulated, the algorithm localizes faulty links more accurately.

Traffic measurement. PathDump also allows to write queries for various measurements such as traffic matrix, heavy hitters, top-*k* flows, and so forth. The following query computes top-1000 flows at a given end-host:

```
h = []; linkID = (*, *); tRange = (t1, t2)
flows = getFlows (linkID, tRange)
for flow in flows:
  (bytes, pkts) = getCount (flow, tRange)
  if len(h) < 1000 or bytes > h[0][0]:
    if len(h) == 1000: heapq.heappop (h)
    heapq.heappush (h, (bytes, flow))
return h
```

To obtain top-*k* flows from multiple end-hosts, the controller can `execute` this query at the desired subset of the end-hosts.

5.2.4 Reducing debugging space

As discussed in §5.1, some network debugging problems necessitate an in-network implementation. One such problem is network switches incorrectly modifying the packet header — for some corner case scenarios, it seems hard for any end-host based system to be able to debug such problems.

One precise example in case of PathDump is switches inserting incorrect switchIDs in the packet header. In case of such network anomalies, PathDump may not be able to identify the problem. For instance, consider the path conformance application from §5.2.3 and suppose we want to ensure that packets do not traverse a

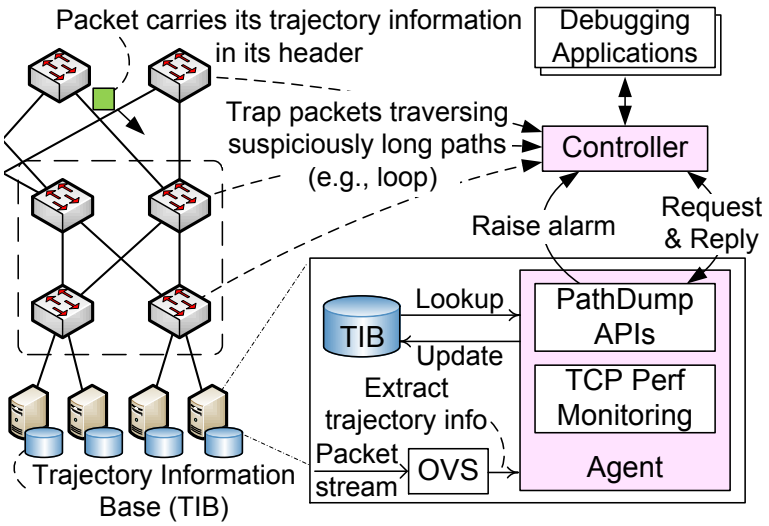


Figure 5.1: PathDump system overview

switch s_1 (that is, switchID=s_1 in the example). Suppose the packet trajectory {src, s_1, s_2, ..., dst} actually involves s_1 and hence, PathDump must raise an alarm.

The main problem is that if s_1 inserts a wrong switchID, say s_1', then PathDump will not raise an alarm. However, in many cases, the trajectory {src, s_1', s_2, ..., dst} in itself will be infeasible — either because s_1' is not one of the switchIDs or because the switch with ID s_1' does not connect directly to either src or s_2. In such cases, PathDump will be able to trigger an alarm stating that one of the switches has inserted incorrect switchID; this is because PathDump continually compares the extracted packet trajectory to the ground truth (network topology) stored in PathDump.

5.3 Implementation

PathDump implementation comprises of three main components (Figure 5.1):

- In-network implementation for tracing packet trajectories using packet headers and static network switch rules (§5.3.1); PathDump's current implementation relies entirely on commodity OpenFlow features for packet trajectory tracing.
- A server stack that implements a storage and query engine for identifying, triggering and debugging anomalous network behavior (§5.3.2); we use C/C++ and Python for implementing the stack.
- A controller running network debugging applications in conjunction with the server stack (§5.3.3). The current controller implementation uses Flask [4] — a

micro framework supporting a RESTful web service — for exchange of query-responses messages between the controller and the end-hosts.

We describe each of the individual components below. As mentioned earlier, PathDump implementation is available at https://github.com/PathDump.

5.3.1 Tracing packet trajectory

PathDump traces packet trajectories at per-packet granularity by embedding into the packet header the IDs of switches that a packet traverses. To achieve this, PathDump resolves two related challenges.

First, the packet header space is a scarce resource. The naïve approach of having each switch embed its switchID into the header before forwarding the packet would require large packet header space, especially when packets can traverse non-shortest paths (*e.g.*, due to failures along the shortest path). For instance, tracing a 8-hop path on a 48-ary FatTree topology would require 4 bytes worth of packet header space, which is not supported using commodity network components[4]. PathDump traces packet trajectories using close to optimal packet header space by using the link sampling idea presented in our preliminary work, CherryPick §4.2. Intuitively, CherryPick builds upon the observation that most frequently used datacenter network topologies are very structured (*e.g.*, FatTree, VL2) and this structure enables reconstructing an end-to-end path by keeping track of a few carefully "sampled" links along any path. We provide more details below.

The second challenge that PathDump resolves is implementation of packet trajectory tracing using commodity off-the-shelf SDN switches. Specifically, PathDump uses the VLAN and the MPLS tags in packet headers along with carefully constructed network switch rules to trace packet trajectories. One key challenge in using VLAN tags is that the ASIC of SDN switch (*e.g.*, Pica8 P-3297) typically offers line rate processing of a packet carrying up to two VLAN tags (*i.e.*, QinQ). Hence, if a packet somehow carries three or more tags in its header, a switch attempting to match TCP/IP header fields of the packet would trigger a rule miss and usually forward it to the controller. This can hurt the flow performance. We show that PathDump can enable per-packet trajectory tracing for most frequently used datacenter network topologies (*e.g.*, FatTree and VL2), even for non-shortest paths (*e.g.*, up to 2 hops in addition

[4]We believe networks will evolve to support larger packet header space. We discuss how PathDump could exploit this to provide even stronger functionality. However, we do note that even with availability to larger packet header space, ideas in PathDump may be useful since this additional packet header space will be shared by multiple applications.

to the shortest path), using just two VLAN tags. Note that these limitations on supported network topologies and path lengths are merely an artifact of today's hardware — PathDump achieves what is possible with today's networks, and as networks evolve to support larger packet header space, PathDump will support more general topologies (*e.g.*, Jupiter network [91]) and/or longer path lengths without any modification in its design and implementation.

However, not all non-shortest paths need to be saved and examined at end-hosts. In particular, when a path is *suspiciously long*, instant inspection at the controller is desirable while packets are on the fly; it may indeed turn out to be a serious problem such as routing loop. PathDump allows the network operator to define the number of hops that would constitute a suspiciously long path (we use 4 hops in addition to the shortest path length as default because packets rarely traverse such a long path in datacenter networks).

For the ease of understanding, we briefly review the ideas from CherryPick below; we refer the readers to §4.2 for more detailed discussion and evaluation. We then close the subsection with a discussion on identifying and trapping packets traversing a suspiciously long path.

Tracing technique. The need for techniques like CherryPick is clear; a naïve approach of embedding link ID of each hop into the packet header simply does not work (more details in §4.2). Assuming 48-port switches, embedding a 6-hop path requires 36 bits in the header space whereas two VLAN tags only allow 24 bits.

The core idea of CherryPick is to sample links that suffice in representing an end-to-end path. One key challenge is that sampling links makes a local identifier inapplicable. Instead, each link should be assigned a global identifier. Clearly, the number of physical links is far more than that of available link IDs (c.f., 4,096 unique link IDs expressed in a 12 bit VLAN identifier vs. 55,296 physical links in a 48-ary fat-tree topology).

In addressing the issue, the following observation is used: aggregate switches between different lower level blocks (e.g., pods) must be interconnected only through core switches. Therefore, instead of assigning global IDs for the links in each pod, it becomes possible to share the same set of global IDs across pods. In addition, the scheme efficiently assigns IDs to core links by applying an edge-coloring technique [31]. The following describes how the links should be picked for fat-tree and VL2:

• *Fat-tree:* A key observation in it is that given any 4-hop path, when a packet reaches a core switch, the ToR-aggregate link it traversed becomes easily known, and there is only a single route to destination from the core switch. Hence, to build the end-to-end path, it is sufficient to pick one aggregate-core link that the packet traverses. When the packet is diverted from its original shortest path, the technique selects one extra link every additional 2 hops. Thus, two VLAN tags make it feasible to trace any 6-hop path. The mechanism is easily converted into OpenFlow rules (see §4.2). The number of rules at switch grows linearly over switch port density.

• *VL2:* VL2 requires to sample three links for tracing any 6-hop path. Hence, we additionally use DSCP field. However, because the field is only 6-bits long, we use it in order to sample an ToR-aggregate link in pod where there are only k links. After the DSCP field is spent, VLAN tags are being used over a subsequent path. If a packet travels over a 6-hop path, it carries one DSCP value and two VLAN tags at the end. In this way, rule misses on data plane is prevented for packets traversing a 6-hop path. We need two rules per ingress port: one for checking if DSCP field is unused, and the other to add VLAN tag otherwise, thus still keeping low switch rule overheads.

Given a 12-bit link ID space (*i.e.*, 4,096 link IDs), the scheme supports a fat-tree topology with 72-port switches (about 93K servers). Since DSCP field is additionally used for VL2, the scheme can support a VL2 topology with 62-port switches (roughly 19K servers).

Instant trap of suspiciously long path. PathDump by design supports identifying and trapping packets traversing a suspiciously long path. When a packet traverses one such path, it cannot help but carry at least three tags. An attempt to parse IP layer for forwarding at switch ASIC would cause a rule miss and the packet is sent to the controller. The controller then can immediately identify the suspiciously long path. We leverage this ability of PathDump to implement a real-time routing loop detection application (see §5.4.5).

5.3.2 Server stack

The modules in the server stack conduct three tasks mainly. The first is to extract and store the path information embedded in the packet header. Next, a query processing module receives queries from the controller, consumes the stored path data and provides responses. The final task is to do active monitoring of flows' performance and prompt raise of alerts to the controller.

Trajectory information management. The trajectory information base (TIB) is a

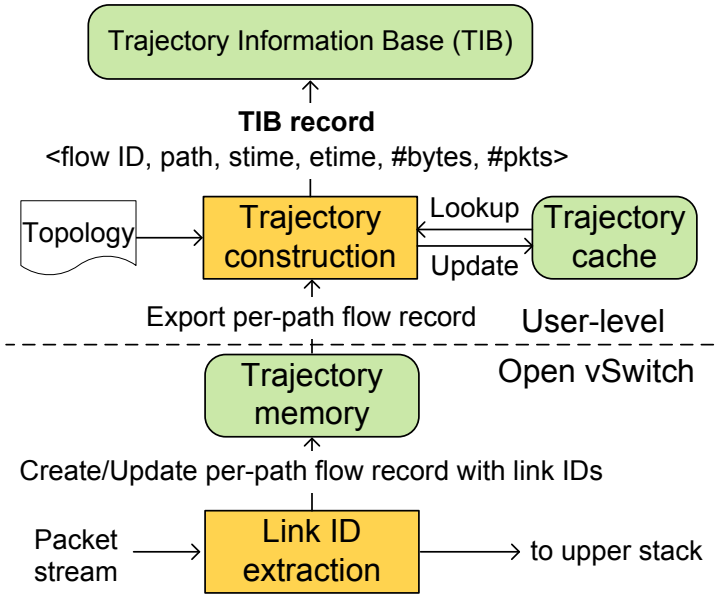


Figure 5.2: Trajectory information update procedure.

repository where packet trajectory information is stored. Because storing path information of individual packets can waste too much disk space, we do per-path aggregation given a flow. In other words, we maintain unique paths and their associated counts for each flow. First, a packet is classified based on the usual 5-tuple flow ID (*i.e.*, <srcIP, dstIP, srcPort, dstPort, proto>). Then, a path-specific classification is conducted. Figure 5.2 illustrates an overall procedure of updating TIB.

When a packet arrives at a server, we first retrieve its metadata (flow ID, path information (*i.e.*, link IDs) and bytes). Because the path information is irrelevant to the upper layer protocols, we strip it off from the packet header in Open vSwitch (OVS) before it is delivered to the upper stack via the regular route. Next, using the flow ID and link IDs together as a key, we create or update a per-path flow record in trajectory memory. Note that link IDs do not represent a complete end-to-end path yet. Each record contains flow ID, link IDs, packet and byte counts and flow duration. That is, one per-path flow record corresponds to statistics on packets of the same flow that traversed the same path. Thus, at a given point in time, more than one per-path flow record can be associated with a flow. Similar to NetFlow, if FIN or RST packet is seen or a per-path flow record is not updated for a certain time period (*e.g.*, 5 seconds), the flow record is evicted from the trajectory memory and forwarded to the trajectory construction sub-module.

The sub-module then constructs an end-to-end path with link IDs in a per-path flow record. It first looks up the trajectory cache with srcIP and link IDs. If there is a cache hit, it immediately converts the link IDs into a path. If not, the module maps link IDs

to a series of switches by referring to a physical topology, and builds an end-to-end path. It then updates the trajectory cache with (srcIP, link IDs, path). In this process, a "static" physical network topology graph suffices, and there is no need for dynamically updating it unless the topology changes physically. Finally, the module writes a record (<flow ID, path, stime, etime, #bytes, #pkts>) to TIB.

We add to OVS about 150 lines of C code to support the trajectory extraction and store function, and run the modified OVS on DPDK [3] for high-speed packet processing (e.g., 10 Gbps). The module is implemented with roughly 600 lines of C++ code. We build TIB using MongoDB [7].

Query processing. PathDump maintains TIB in a distributed fashion (across all servers in the datacenter). The controller sends server agents a query, composed of PathDump APIs (§5.2.1), which in turn processes the TIB data and returns results to the controller. The querying mechanism is composed of about 640 lines of Python code.

Depending on debugging applications, the controller needs to consult more than one TIB. For instance, to check path conformance of a packet or flow, accessing only one TIB is sufficient. On the other hand, some debugging queries (*e.g.*, load imbalance diagnosis; see §5.4.2) need path information from all distributed TIBs.

To handle these different needs properly, we implement two types of query mechanisms: (i) direct query and (ii) multi-level query. The former is a query that is directly sent to one specific TIB by the controller. Inspired by Dremel [67] and iMR [65], we design a multi-level query mechanism whereby the controller creates a multi-level aggregation tree and distributes it alongside a query. When a server receives query and tree, it performs two tasks: (i) query execution on local TIB and (ii) redistribution of both query and tree.

In general, multi-level data aggregation mechanisms including ours can be ineffective in improving response times when the data size is not large and there is no much data reduction during aggregation along the tree. In §6.6, we present the tradeoff through two multi-level queries—flow size distribution and top-k flows.

Finally, when a query is executed, the latest TIB records relevant to the query may reside in the trajectory memory, yet to be exported to the TIB. We handle this by creating an IPC channel and allowing the server agent to look up the trajectory memory. Not all debugging applications require to access the trajectory memory. Instead, the alerts raised by `Alarm()` trigger the access to the memory for debugging at even finer-grained time scales.

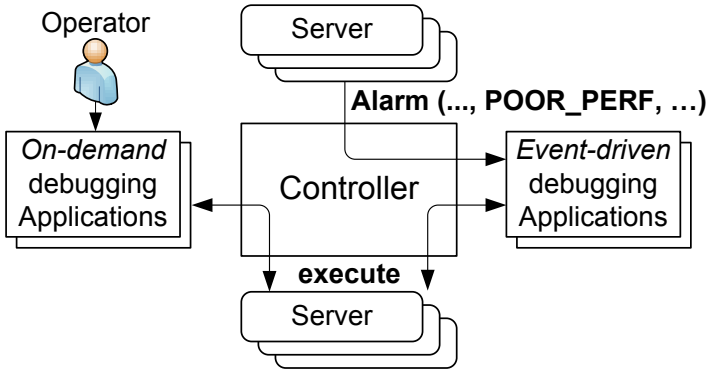


Figure 5.3: Workflow of PathDump.

Active monitoring module. Timely triggering of a debugging process requires fast detection of symptoms on network problems. Servers are a right vantage point to instantly sense the symptoms like TCP timeouts, high retransmission rates, large RTT and low throughput.

We thus implement a monitoring module at server that checks TCP connection performance passively, and promptly raises alerts to the controller in the advent of abnormal TCP behavior. Specifically, by using `tcpretrans` script in perf-tools[5], the module checks the packet retransmission of individual flows at regular intervals (configured by installing a query). If packet retransmissions are observed more than a configured frequency, an alert is raised to the controller, which can subsequently take actions in response. Thus, this active TCP performance monitoring allows fast troubleshooting. We exploit the alert functionality to expedite debugging tasks such as silent packet drop localization (§5.4.3), blackhole diagnosis (§5.4.4) and TCP performance anomaly diagnosis (§5.4.6).

In addition, network behavior desired by operators can be expressed as network invariants (*e.g.*, maximum path length), which can be installed on end-hosts using `install()`. This module uses `Alarm()` to inform any invariant's violation as depicted in §5.2.3.

5.3.3 PathDump controller

PathDump controller plays two roles: installing flow rules on switches and executing debugging applications.

It installs flow rules in switches that append link IDs in the packet header (using `push_vlan` output action) in order to enable packet trajectory tracing. This is one-time task when the controller is initialized, and the rules are not modified once they are

[5] https://github.com/brendangregg/perf-tools

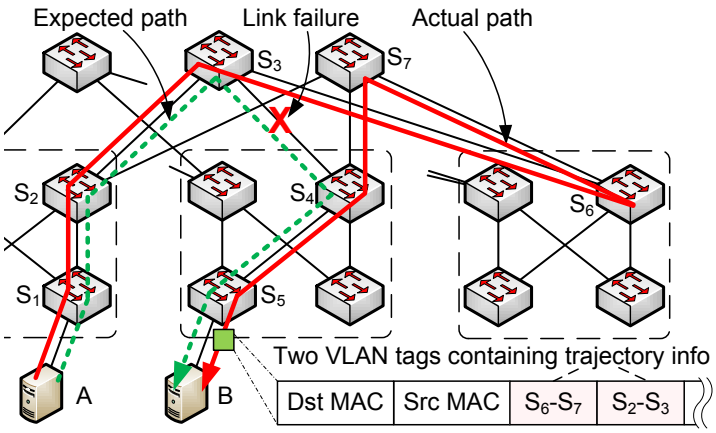


Figure 5.4: An example of path conformance check. The dotted green line is an expected path and the red line is an actual path that packet traverses.

installed. We use switches that support a pipeline of flow tables and that are therefore compatible with OpenFlow specification v1.3.0.

Debugging applications can be executed under two contexts as depicted in Figure 5.3: (i) event-driven, and (ii) on-demand. It is event-driven when the controller receives alerts from the active monitoring module at end-hosts. The other, obvious way is that the operator executes debugging applications on demand. Queries and results are exchanged via direct query or multi-level query. The controller consists of about 650 lines of Python code.

5.4 Applications

PathDump can support various debugging applications for datacenter network problems including both persistent and transient ones (see Table 5.2 for a comprehensive list of debugging applications). In this section, we highlight a subset of those applications.

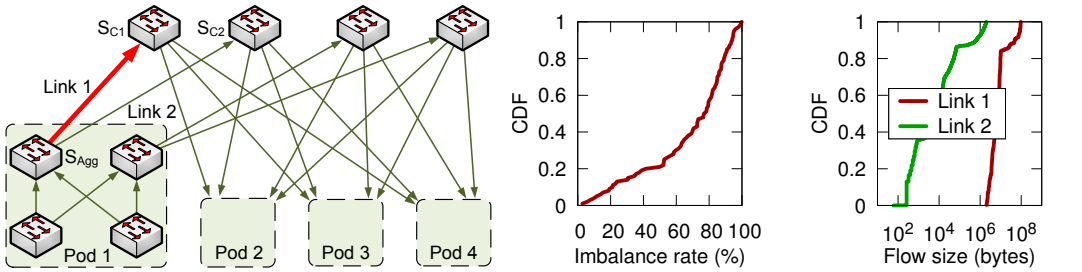
5.4.1 Path conformance check

A path conformance test is to check whether an actual path taken by a packet conforms to operator policy. To demonstrate that, we create an experimental case shown in Figure 5.4. In the figure, the intended path of a packet is a 4-hop shortest path from server A to B. However, a link failure between switches S_3 and S_4 makes S_3 forward the packet to S_6 (we implement a simple failover mechanism in switches with a few flow rules). As a result, the packet ends up traversing a 6-hop path. The PathDump agent in B is configured with a predicate, as a query (as depicted in §5.2.3), that a 6-hop or longer path is a violation of the path conformance policy. The agent detects

Application	Description	PathDump	PathQuery[75]	Everflow[112]	NetSight[47]	TPP[51]
Loop freedom [47]	Detect forwarding loops	✓	✓	✓	✓	?
Load imbalance diagnosis [112]	Get fine-grained statistics of all flows on set of links	✓	✓	✓	✓	✓
Congested link diagnosis [75]	Find flows using a congested link, to help rerouting	✓	✓	✓	✓	✓
Silent blackhole detection [112, 75]	Find switch that drops all packets silently	✓	✓	✓	✓	✗
Silent packet drop detection [112]	Find switch that drops packets silently and randomly	✓	✓	✓	✓	✗
Packet drops on servers [112]	Localize packet drop sources (network vs. server)	✓	✓	✓	✓	✓
Overlay loop detection [112]	Loop between SLB and physical IP	✗	✓	✓	✓	?
Protocol bugs [112]	Bugs in the implementation of network protocols	✓	✓	✓	✓	?
Isolation [47]	Check if hosts are allowed to talk	✓	✓	✓	✓	✓
Incorrect packet modification [47]	Localize switch that modifies packet incorrectly	✗	✓	?	✓	✗
Waypoint routing [47, 75]	Identify packets not passing through a waypoint	✓	✓	✓	✓	✓
DDoS diagnosis [75]	Get statistics of DDoS attack sources	✓	✓	✓	✓	✓
Traffic matrix [75]	Get traffic volume between all switch pairs in a switch	✓	✓	✓	✓	✓
Netshark [47]	Nework-wide path-aware packet logger	✓	✓	✓	✓	✓
Max path length [47]	No packet should exceed path length of size n	✓	✓	✓	✓	✓

Table 5.2: Debugging applications supported by existing tools and PathDump. The table assumes that Everflow performs per-switch per-packet mirroring. Of course, this will have much higher bandwidth requirements than the network traffic itself. If Everflow uses the proposed sampling to minimize bandwidth overheads, many of the above applications will not be supported by Everflow.

such packets in real time and alerts the controller to the violation along with the flow key and trajectory.



(a) S_{Agg} poorly load-balances traffic (b) Load imbalance rate (c) Flow size distribution

Figure 5.5: Load imbalance diagnosis. (a) illustrates a load imbalance case. (b) shows, as reference, the load imbalance rate between links 1 and 2. (c) shows the flow size distribution built by querying all TIBs.

5.4.2 Load imbalance diagnosis

Datacenter networks employ load-balancing mechanisms such as ECMP and packet spraying [35] to exploit numerous equal-cost paths. However, when these mechanisms work poorly, uneven load splits can hurt throughput and flow completion time. PathDump can help narrow down the root causes of load imbalance problems, which we demonstrate using two load-balancing mechanisms: (i) ECMP and (ii) packet spraying.

ECMP load-balancing. This scenario (Figure 5.5(a)) assumes that a poor hash function always creates collisions among large flows. For the scenario, we configure switch S_{Agg} in pod 1 such that it splits traffic based on flow size. Specifically, if a flow is larger than 1 MB in size, it is pushed onto link 1. If not, it is pushed onto link 2. Based on the web traffic model in [21], we generate flows from servers in pod 1 to servers in the remaining pods. As a metric, we use imbalance rate, $\lambda = (L_{max}/\overline{L} - 1) \times 100$ (%) where L_{max} is the maximum load on any link and $\overline{L}$ is the mean load over all links [80].

Figure 5.5(b) shows the load imbalance rate between the two links measured every 5 seconds for 10 minutes. During about 80% of the time, the imbalance rate is 40% or higher. With the load imbalance diagnosis application in §5.2.3, PathDump issues a multi-level query to all servers and collects byte counts of flows that visited those two links. As shown in Figure 5.5(c), flow size distributions on the two links are sharply divided around 1 MB. With flow IDs and their sizes in the TIBs, operators can reproduce this load imbalance scenario for further investigation.

This scenario illustrates how PathDump handles a persistent problem. The appli-

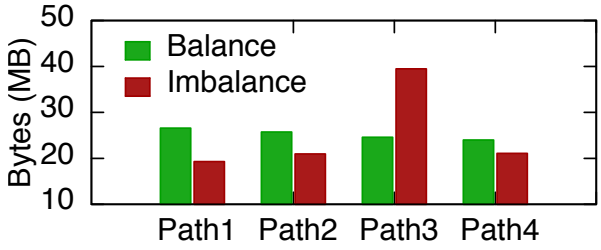
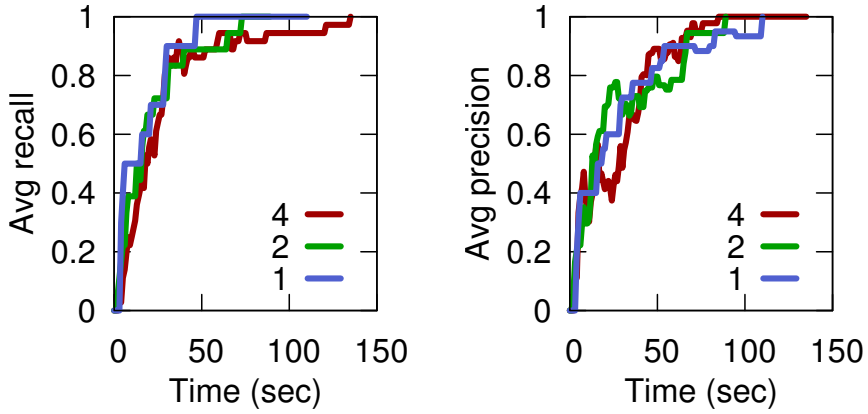


Figure 5.6: Traffic distribution of a flow along four different paths under balanced and imbalanced cases.

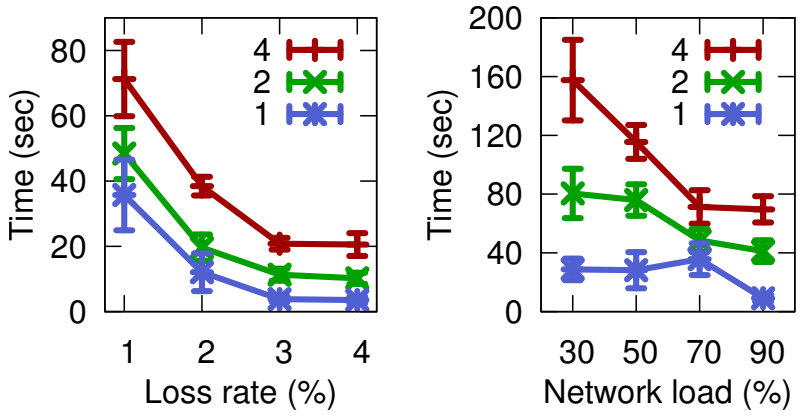


(a) Recall (b) Precision

Figure 5.7: Performance of the silent random packet drop debugging algorithm. Average recall and precision are presented over 10 runs. The network load is set to 70% and each faulty interface drops packets at 1% rate. The numbers (i.e., 1, 2 and 4) in legend denote the number of faulty interfaces.

cation can be easily extended for tackling transient ECMP hash collisions among long flows by exploiting the TCP performance alert function.

Packet spraying. In this scenario, packets of a flow are split among four possible equal-cost paths between a source and destination. For demonstration, we create two cases: (i) a balanced case and (ii) an imbalanced case. In a balanced case, the split process is entirely random, thereby ensuring fair load-balance, whereas in an imbalanced case, we configure switches so that more packets are deliberately forwarded to one of the paths (*i.e.*, Path 3 in Figure 5.6). The flow size is set to 100 MB. Figure 5.6 is drawn using per-path statistics of the flow obtained from the destination TIB. As shown in the figure, operators can check whether packet spraying works well or not. In case of poor load-balancing, they can tell which path (more precisely, which link) is under- or over-utilized. The per-packet path tracing ability of PathDump allows this



(a) Network load = 70% (b) Loss rate = 1%

Figure 5.8: Time taken to reach 100% recall and precision. The numbers (i.e., 1, 2 and 4) in legend denote the number of faulty interfaces. The error bar is standard error, i.e., $\sigma/\sqrt{n}$ where σ is standard deviation and n is the number of runs ($=10$).

level of detailed analysis. For real-time monitoring, it is sufficient to install a query (using `install()`) that monitors the traffic amount difference among subflows.

5.4.3 Silent random packet drops

We implement the silent packet drop debugging application as described in §5.2.3 and conduct experiments in a 4-ary fat-tree topology, where each end-host generates traffic based on the same web traffic model. We configure 1-4 randomly selected interfaces such that they drop packets at random. We run the MAX-COVERAGE algorithm and evaluate its performance based on two metrics: recall and precision. Recall is $\frac{\#TPs}{\#TPs+\#FNs}$ while precision is $\frac{\#TPs}{\#TPs+\#FPs}$ where true positive is denoted as TP, false negative as FN, and false positive as FP.

In our experiment, as time progresses, the number of alerts received by the controller increases; so does the number of failure signatures. Hence, from Figure 5.7, we observe the accuracy (both recall and precision) also increases accordingly; the recall increases faster than the precision. It is clear from Figure 5.8, as loss rate or network load increase, the controller receives alerts from end-hosts at higher rate, and thus the algorithm takes less time to obtain 100% recall and precision, making it possible to debug the silent random packet drops fast and accurately.

5.4.4 Blackhole diagnosis

We demonstrate how PathDump reduces a debugging search space with a blackhole scenario in the network with a 4-ary fat-tree topology where packet spraying is deployed. Again, we generate the same background traffic used in §5.4.3 to create noises in the debugging process. We create a 100 KB TCP flow and its packets are randomly routed through four possible paths and test two cases.

Blackhole at an aggregate-core link. Obviously, the subflow traffic passing the blackhole link is all dropped. The controller receives an alarm from PathDump agent at sender in 1 sec, immediately retrieves all TIB records for the flow and finds one record for the dropped subflow missing. While examining the paths found in TIB records, it finds that one path did not appear in the TIB. Since only one path (hence, one subflow) was impacted, it produces three switches as a potential culprit: core switch, source and destination aggregate switches (thus avoiding the search of all 10 switches in the four paths).

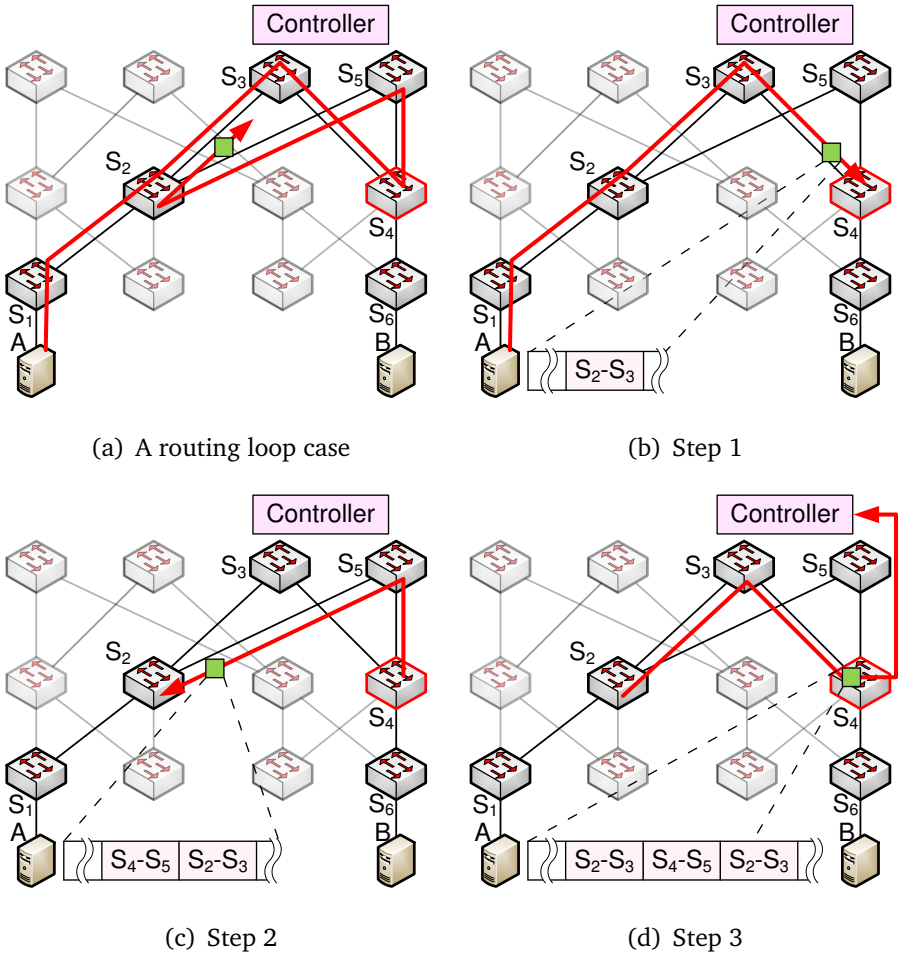
Blackhole at a ToR-aggregate link in the source pod. This blackhole impacts two subflows. The controller identifies two paths that impacted the two subflows using the same way as before. By joining the two paths, the controller can pick four common switches, which should be examined with higher priority.

Note that if more number of flows (and their subflows) are impacted by the blackhole, PathDump can localize the exact source of the blackhole.

5.4.5 Routing loop debugging

PathDump debugs routing loop in *real-time* by trapping a suspiciously long path in the network. As discussed in §5.3.1, a packet carrying more than two tags is automatically directed to the controller. This feature is a foundation of *making routing loops naturally manifest themselves* at the controller. More importantly, the fact that the controller has a direct control over suspicious packets makes it possible to detect routing loops *of any size*.

Real timeliness. We create a 4-hop routing loop as shown in Figure 5.9(a). Specifically, switch S_4 is misconfigured and all core switches are configured to choose an alternative egress port except the ingress port of a packet. In the figure, switches from S_2 to S_5 constitute the loop. Under this setup, it takes about **47 ms** on average until the controller detects the loop. When the packet trapped in this loop ends up carrying three tags (see Figures 5.9(b)–5.9(d)) and appears at the controller, two of the tags have the



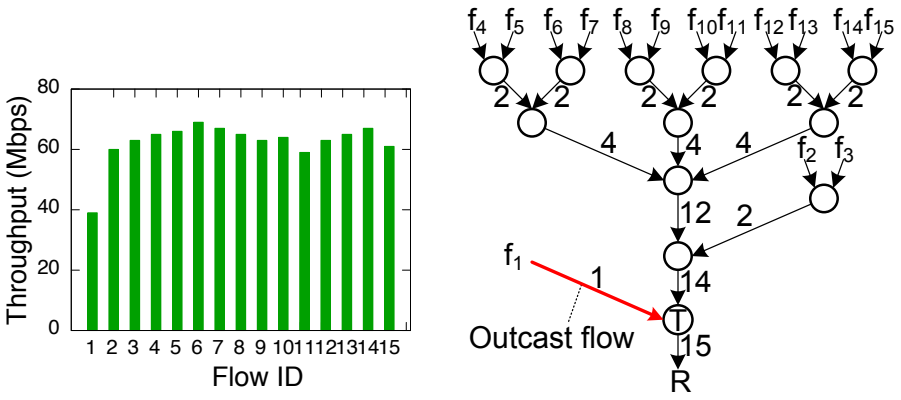
(a) A routing loop case (b) Step 1

(c) Step 2 (d) Step 3

Figure 5.9: Debugging a routing loop. (a) A routing loop is illustrated. (b) A packet carries a VLAN tag whose value is an ID for link S_2-S_3 appended by S_3. (c) S_4 bounces the packet to S_5; S_5 forwards the packet to one remaining egress port (to S_2) while appending an ID for link S_4-S_5 to the packet header. (d) S_3 appends a third tag of which the value is a ID for link S_2-S_3; at S_4, the packet is *automatically* forwarded to the controller since ASIC in switches only recognizes two VLAN tags whilst the packet carries three; at this stage, the controller immediately detects the loop by finding the repeated link S_2-S_3 from the packet header.

same link ID (S_2-S_3 in Figure 5.9(d)). Hence, the loop is detected immediately at this stage.

Detecting loops of any size. In this scenario, we create a 6-hop routing loop (not shown for brevity). The controller finds no repeated link IDs from three tags when it sees the packet for the first time. The controller locally stores the three tags, strips them off from the packet header, and sends the packet back to the switch. Since the



(a) (b)

Figure 5.10: Diagnosis of TCP outcast. Unfairness of throughput is shown in (a). In (b), the communication graph is mapped onto a physical topology, and edge weight is the number of flows arriving at an input port. Both data sets are made available from TIB.

packet is trapped in the 6-hop loop, it will have another set of three tags and be forwarded to the controller. This time, comparing link IDs in previous and current tags, the controller observes that there is at least one repeated link ID and detects the loop. The whole process took **∼115 ms**. Detecting even larger loops involves exactly the same procedure.

5.4.6 TCP performance anomaly diagnosis

PathDump can diagnose incast [29] and outcast [83] problems in a fine-grained manner although they are transient. In particular, we test a TCP outcast scenario. For a realistic setup, we generate the same type of TCP background traffic used in §5.4.4. In addition to that, 15 TCP senders send data to a single receiver for 10 seconds. Thus, as shown in Figure 5.10(b), a flow from f_1 and 14 flows from $f_2 - f_{15}$ arrive on two different input ports at switch T. They compete for the same output port at the switch toward receiver R. As a result, these flows experience the port blackout phenomenon, and the flow from f_1 sees the most throughput loss (see [83] for more details).

Every 200 ms (default TCP timeout value) the server agents run a query that generates alerts when their TCP flows repeatedly retransmit packets. The diagnosis application at the controller starts to work when it sees a minimum of 10 alerts from different sources to a particular destination. Since all alerts specify R as receiver, the application requests flow statistics (*i.e.*, bytes, path) from R and diagnoses the root cause for high alerts. It first analyzes the throughput for each sender (Figure 5.10(a)) and constructs a

path tree for all 15 flows (Figure 5.10(b)). It then identifies that the flow from f_1 (one closest to the receiver) is most highly penalized. PathDump concludes the TCP unfairness stems from the outcast because these patterns fit the outcast's profile. We observe that the application initiates its diagnosis in 2-3 seconds since the onset of flows and finishes it within next 200 ms.

5.5 Evaluation

We first study the performance of direct and multi-level queries in terms of response time and data overheads. We then evaluate CPU and memory overheads at end-host in processing packet stream and in executing queries.

5.5.1 Experimental setup

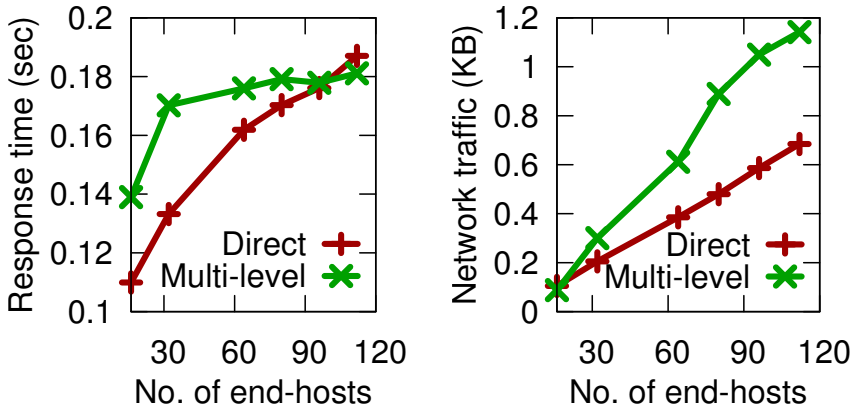
We build a real testbed that consists of 28 physical servers; each server is equipped with Xeon 4-core 3.1 GHz CPU and a dual-port 1 GbE card. Using the two interfaces, we separate management channel from data channel. The controller and servers communicate with each other through the management channel to execute queries. Each server runs four docker containers (in total, 112 containers). Each container is assigned one core and runs a PathDump agent to access TIB in it. In this way, we test up to 112 TIBs (*i.e.*, 112 end-hosts). We only refer to container as end-host during the query performance evaluation. Each TIB has 240K flow entries, which roughly corresponds to the number of flows seen at a server for about an hour. We estimate the number based on the observation that average flow inter-arrival time seen at server is roughly 15 ms (∼67 flows/sec) [55].

For multi-level query execution, we construct a logical 4-level aggregation tree with 112 end-hosts. Our PathDump controller sits on the top of the tree (level 0). Right beneath the controller are 7 nodes or end-hosts (level 1). Each first-level node has, as its child, four nodes (level 2), each of which has four nodes at the bottom (level 3).

For the packet progressing overhead experiment, we use another server equipped with a 10 GbE card. In this test, we forward packets from all other servers to a virtual port in DPDK vSwitch via the physical 10GbE NIC.

5.5.2 Query performance

We compare the performance of direct query with that of multi-level query. To understand which type of query suits well to a debugging application, we measure two key metrics: i) end-to-end response time, and ii) total data volume generated. We test two



(a) Response time (b) Traffic amount

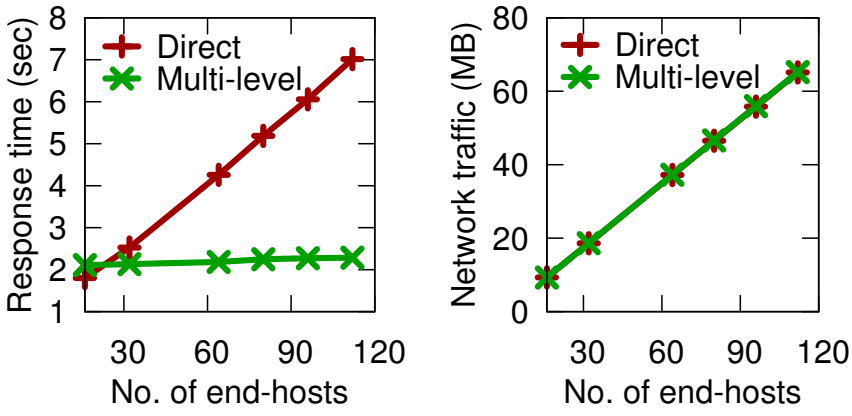
Figure 5.11: Average end-to-end response time and traffic amount of a flow size distribution query.

queries—flow size distribution of a link and top-k flows. For the top-k flows query, we set k to 10,000. Results are averaged over 20 runs.

Results. Through these experiments, we make two observations (confirmed via Figures 5.11 and 5.12) as follows.

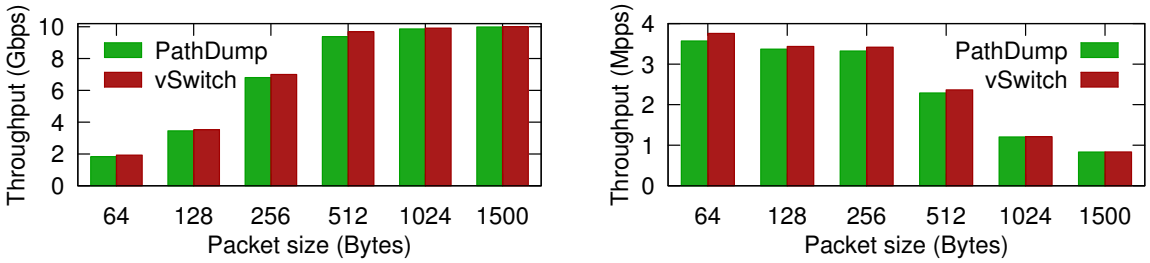
1) When more servers are involved in a query, multi-level query is in general better than direct query. Figure 5.11(a) shows that multi-level query initially takes longer than direct query. However, the response time gap between the two gets smaller as the number of servers increases. This is due to three reasons. First, the aggregation time (the time to aggregate responses at the controller) of direct query is always larger than that of multi-level query. Second, the aggregation time of direct query linearly grows in proportion to the number of end-hosts whereas that of multi-level query gradually grows. Lastly, network delays of both queries change little regardless of the number of servers.

2) If aggregation reduces response data amount substantially, multi-level query is more efficient than direct query. When multi-level query is employed for computing the top-k flows, $(n_i - 1) \cdot k$ number of key-value pairs are discarded at level $i-1$ during aggregation where n_i is the number of nodes at level i ($i < 3$). A massive data reduction occurs through the aggregation tree. Hence, the data amount exchanged in multi-level query is similar to that in direct query (Figure 5.12(b)). Moreover, the computation overhead for aggregation is distributed across multiple intermediate servers. On the contrary, in direct query, the controller alone has to process a large number of key-value pairs (*i.e.*, $k \cdot n_3$ where n_3 is the total number of servers used). Hence, the



(a) Response time (b) Traffic amount

Figure 5.12: Average end-to-end response time and traffic amount of a top-10,000 flows query.



(a) Throughput in Gbits per second (b) Throughput in million-packets per second

Figure 5.13: Forwarding throughput of PathDump and vSwitch. Each bar represents an average over 30 runs.

majority of the response time is attributed to computation at the controller, and the response time grows linearly as the number of servers increases (Figure 5.12(a)). Due to the horizontal scaling nature of multi-level query, its response times remain steady regardless of the number of servers. In summary, these results suggest that multi-level query can scale well even for a large cluster and direct query is recommended when a small number of servers are queried.

5.5.3 Overheads

Packet processing. We generate traffic by varying its packet size from 64 to 1500 bytes. Each packet carries 1-2 VLAN tags. While keeping about 4K flow records (roughly equivalent to 100K flows/sec at a rack switch connected to 24 hosts) in the trajectory memory, PathDump does about 0.8–3.6M lookups/updates per second (0.8M for 1500B packets and 3.6M for 64B). Under these conditions, we measure average throughput

in terms of bits and packets per second over 30 runs.

From Figure 5.13, we observe that PathDump introduces a maximum of 4% throughput loss compared to the performance of the vanilla DPDK vSwitch. The figure omits confidence intervals as they are small. In all cases, the throughput difference is marginal. Note that due to the limited CPU and memory resources allocated, DPDK vSwitch itself suffers throughput degradation as packet size decreases. Nevertheless, it is clear that PathDump introduces minimal packet processing overheads atop DPDK vSwitch.

Query processing. We measure CPU resource demand for continuous query processing at end-host. The controller generates a mix of direct and multi-level queries continuously in a serialized fashion (i.e., a new query after receiving response for previous one). We observe that less than 25% of one core cycles is consumed at end-host. As datacenter servers are equipped with multi-core CPUs (e.g., 18-core Xeon E5-2699 v3 processor), the query processing introduces relatively less overheads.

Storage. PathDump only needs about 10 MB of RAM at a server for packet trajectory decoding, trajectory memory and trajectory cache. It also needs about 110 MB of disk space to store 240K flow entries (roughly equivalent to an hour's worth of flows observed at a server).

5.6 Limitations

Debugging network problems. PathDump does not require network switches to perform complex operations, yet it supports a large class of debugging problems. Table 5.2 shows a list of debugging applications supported by PathDump, except two; Overlay loop detection and incorrect packet modification. Overlay loop could be formed in the following way; a packet's destination server decaps the outer header (VXLAN) and mistakenly injects the packet back into the network, then a network load balancer encaps the VXLAN header, and again forwards to the destination server. With the packet trajectory tracing technique like CherryPick, decoder at an end-host may not have all the information to reconstruct the loop path. Eventually, PathDump could not pin-point the origin of overlay loop (*i.e.*, load balancer).

PathDump allows to view a sub-set of all properties of a packet at each hop. For instance, it can tell switch input and output port that the packet traversed, but not the packet header values and the matched flow table version. Suppose, one of the switches in the path modifies the packet headers incorrectly, then PathDump does not give a clue

of which is the problematic switch in the packet's path.

Impact of packet drops. PathDump end-hosts reconstruct the packet trajectory from the link IDs present in the packet header. However, a packet may not reach the destination for a multitude of reasons, including packet drops due to network congestion, a black hole, or a faulty interface. PathDump resolves the packet drop problem by exploiting the fact that many datacenters typically perform load balancing (using ECMP or packet spraying). In particular, we show in §5.4.3 that the correlation of paths of all impacted flows allows to localize the culprit switch, or link, and the localization accuracy improves as path data of poor flows accumulates. However, in the absence of sufficient data (*e.g.*, only few flows observed the problem), the accuracy would be very low.

End-host agent. User applications are not aware of the link IDs embedded into the packet headers. End-host agent strips link IDs off from the packet headers, updates per-path flow statistics (flowID and path together as a key), then forwards the packet to upper layers. Currently, the agent is implemented and evaluated in OpenVswitch (OVS), both kernel and DPDK versions. For completeness, we should have tested the agent overhead in other virtual switch platforms such as mSwitch [48], Hyper-V virtual switch [5], etc.

Switch flow rules. In section §4.2.3, we have only presented OpenFlow table entries at each switch layer for tracing packet trajectory. In a heterogeneous network setup that has both traditional (operate in L2/L3 mode) and OpenFlow-compatible switches from different switch vendors, enabling packet trajectory tracing may require a different set of table entries to be installed.

5.7 Summary

This chapter presents PathDump, an end-host based network debugger that carefully partitions the debugging functionality between the edge devices and the network switches (in contrast to an entirely in-network implementation used in existing tools). PathDump does not require network switches to perform complex operations like dynamic switch rule updates, per-packet per-switch log generation, packet sampling, packet mirroring, etc., and yet helps debug a large class of network problems over fine-grained time-scales. Evaluation of PathDump over operational network testbeds comprising of commodity network switches and end-hosts show that PathDump requires minimal data plane resources and end-host resources.

Chapter 6

Distributed Network Monitoring and Debugging with SwitchPointer

6.1 Introduction

Debugging network problems require resources like compute, memory, and network bandwidth to collect and monitor telemetry data. As networks evolve to a large number of end-points, higher speeds, and higher utilizations, the amount of resources required to monitor the telemetry data also increases. Moreover, debugging performance problems (*e.g.*, delays, packet drops) need to inspect every packet and collect telemetry data at packet level [112]. This is in contrast to sampled flow-level information provided by existing monitoring tools such as NetFlow, sFlow. However, packet-level monitoring requires even more resources. Increasingly, many data centers need real-time monitoring systems to detect thousands of network events within a few milliseconds [71]. For example, we can use these systems to install predicates that are checked against each packet and report telemetry data of those packets violating the predicates. Enabling real time diagnosis adds to the resources required for network debugging.

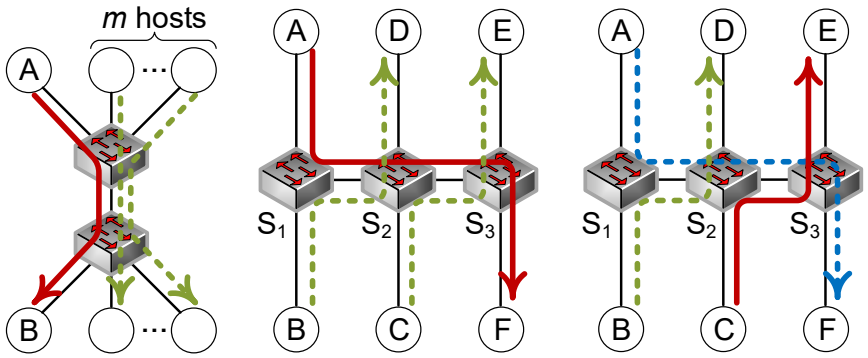
Given the ever growing resource requirements, where should the telemetry data necessary to debug the problems should be captured? On the one hand we have in-network monitoring approaches that collect and monitor telemetry data at switches [13, 63, 107, 64, 50], and query this data using new software or hardware interfaces [74, 41, 75, 6, 51]. Though they provide high network visibility, but often limited by available data plane resources. For example commodity switches have only 10's of Mbits of SRAM for monitoring [74] and strict limits on types of per-packet operation [63]. Limited by these resources, these approaches have to rely on highly aggregated data [74], or

selectively sampled network traffic [112], or approximate counters [107] which sometimes may not accurate enough to diagnose network problems (§6.2).

At the other extreme, we have end-host based monitoring and debugging approaches (*e.g.*, PathDump, Trumpet [71], Pingmesh [44]). These approaches exploit the resources to collect and monitor telemetry data, and use this data to debug network problems. Also, hosts offer the programmability needed to implement various monitoring and debugging functionalities, without need for specialized hardware. But, they lose the benefits of network visibility offered by in-network monitoring approaches.

We present SwitchPointer, a network monitoring and debugging system that integrates the best of the two worlds — resources and programmability of end-host based approaches, and the visibility of in-network approaches. SwitchPointer exploits end-host resources and programmability to collect and monitor telemetry data, and to trigger spurious network events (*e.g.*, using existing end-host based systems like PathDump (Chapter 5). The key contribution of SwitchPointer is to efficiently enable network visibility for such end-host based systems by using switch memory as a "directory service" — in contrast to in-network approaches where switches store telemetry data necessary to diagnose network problems, SwitchPointer switches store *pointers* to end-hosts where the relevant telemetry data is stored. The distributed storage at switches thus operates as a distributed directory service; when an end-host triggers a spurious network event, SwitchPointer uses the distributed directory service to quickly filter the data (potentially distributed across multiple end-hosts) necessary to debug the event.

SwitchPointer design. The key design choice of thinking about network switch storage as a directory service rather than a data store allows SwitchPointer to efficiently solve many problems that are hard or even infeasible for existing systems. For instance, consider the network problems shown in Figure 6.1. We provide an in-depth discussion in §6.2, but note here that existing systems are insufficient to debug the reasons behind high latency, packet drops or TCP timeout problems for the red flow since this requires maintaining temporal state (that is, flow IDs and packet priorities for all flows that the red flow contends with in Figure 6.1(a)), combining state distributed across multiple switches (required in Figure 6.1(b)), and in some cases, maintaining state even for flows that do not trigger network events (for the blue flow in Figure 6.1(c)).



(a) Too much traffic (b) Too many red lights (c) Traffic cascades

Figure 6.1: Three example network problems. Green, blue and red flows have decreasing order of priority. Red flow observes high latency (or even TCP timeout due to excessive packet drops) due to: (a) contention with many high priority flows at a single switch; (b) contention with multiple high priority flows across multiple switches; and (c) cascading problems — green flow (highest priority) delays blue flow, resulting in blue flow contending with and delaying red flow (lowest priority). Please see more details in §6.2.

SwitchPointer is able to solve such problems using a simple design (detailed discussion in §6.4):

- Switches divide the time into *epochs* and maintain a pointer to all end-hosts to which they forward the packets in each epoch;
- Switches embed their switchID and current epochID into the packet header before forwarding a packet;
- End-hosts maintain a storage and query service that allows filtering the headers for packets that match a (switchID, epochID) pair; and,
- End-hosts trigger spurious events, upon which a controller (or an end-host) uses pointers at the switches to locate the data necessary to debug the event.

SwitchPointer challenges. While SwitchPointer design is simple at a high-level, realizing it into an end-to-end system requires resolving several technical challenges. The first challenge is to decide the epoch size — too small an epoch would require either large storage (to store pointers for several epochs) or large bandwidth between the data plane and the control plane (to periodically push the pointers to persistent storage); too large an epoch, on the other hand, may lead to inefficiency (a switch may forward packets to many end-hosts). SwitchPointer resolves this challenge using a hierarchical

data structure, where each subsequent level of the hierarchy stores pointers over exponentially larger time scales. We describe the data structure in §6.4.1.1, and discuss how it offers a favourable tradeoff between switch memory and bandwidth, and system efficiency.

The second challenge in realizing the SwitchPointer design is to efficiently maintain the pointers at switches. The naïve approach of using a hash table for each level of the hierarchy would either require large amount of switch memory or would necessitate one hash operation *per level* per packet for the hierarchical data structure, making it hard to achieve line rate even for modest size packets. SwitchPointer instead uses a *perfect hash function* [2, 42] to efficiently store and update switch pointers in the hierarchical data structure. Perfect hash functions require only 2.1 bits of storage per end-host per-level for storing pointers and only one hash operation per packet (independent of number of levels in the hierarchical data structure). We discuss storage and computation requirements of perfect hash functions in §6.4.1.2.

The final two challenges in realizing SwitchPointer design into an end-to-end system are: (a) to efficiently embed switchIDs and epochIDs into packet header; and (b) handle the fact that switch and end-host clocks are typically not synchronized perfectly. For the former, SwitchPointer can of course use clean-slate approaches like INT [6]; however, we also present a design in §6.4.1.3 that allows SwitchPointer to embed switchIDs and epochIDs into packet header using commodity switches (under certain assumptions). SwitchPointer resolves the latter challenge by exploiting the fact that while the network devices may not be perfectly synchronized, it is typically possible to bound the difference between clocks of any pair of devices within a data center. This allows SwitchPointer to handle asynchrony by carefully designing epoch boundaries in its switch data structures.

SwitchPointer contribution. We have implemented SwitchPointer into an end-to-end system that currently runs over a variety of network testbeds comprising commodity switches and end-hosts. Evaluation of SwitchPointer over these testbeds (§6.5, §6.6) demonstrates that SwitchPointer can monitor and debug network events at sub-second timescales while requiring minimal switch and end-host resources.

This chapter is organized as follows: Section 6.2 presents the problems that motivates the need for SwitchPointer, Section 6.4 elaborates on SwitchPointer design, and §6.5 presents network problems supported by SwitchPointer which are hard or infeasible to debug for existing systems. Finally, Section 6.6 has SwitchPointer evaluation results.

6.2 Motivation

In this section we discuss several network problems that motivate the need for SwitchPointer.

6.2.1 Too much traffic

The first class of problems are related to priority-based and microburst-based contention between flows.

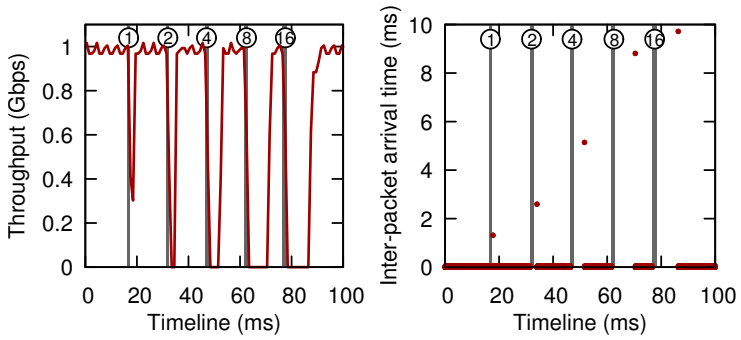
Priority-based flow contention. Consider the case of Figure 6.1(a), where a low-priority flow competes with many high-priority flows on an output port. As a result, the low priority flow may observe throughput drop, high inter-packet arrival times, or even TCP timeouts.

To demonstrate this problem, we set up an experiment. We create a low-priority TCP flow between two hosts A and B that lasts for 100ms. We then create 5 batches of high-priority UDP bursts; each burst lasts for 1ms and has increasingly larger number of UDP flows (m in Figure 6.1(a)) all having different source-destination pairs. We use Pica8 P-3297 switches in our experiment; the switch allows us to delay processing of low-priority packets in the presence of a high-priority packet.

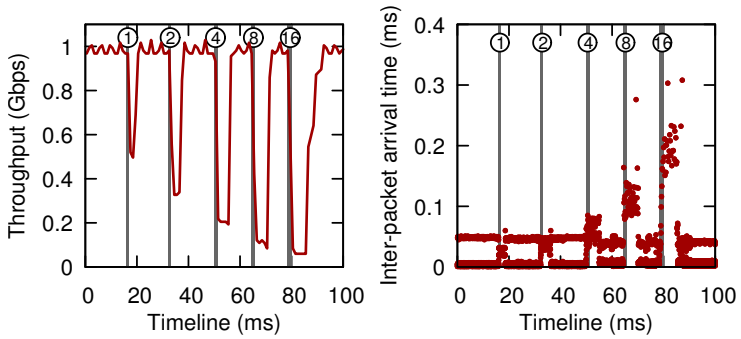
Figure 6.2(a) demonstrates that high-priority UDP bursts hurt the throughput and latency performance of the TCP flow significantly. With increasingly larger number of high-priority flows in the burst, the TCP flow observes increasingly more throughput drop eventually leading to starvation (*e.g.*, 0 Gbps for $\sim$10 ms in case of 16 UDP flows). The figure also shows that increasing number of high-priority flows in the burst results in increasingly larger inter-arrival times for packets in the TCP flow. The reduced throughput and increased packet delays may, at the extreme, lead to TCP timeout.

Microburst-based flow contention. We now create a microburst based flow contention scenario, where congestion lasts for short periods, from hundreds of microseconds to a few milliseconds, due to bursty arrival of packets that overflows a switch queue. To achieve this, we use the same set up as priority-based flow contention with the only difference that we use a FIFO queue instead of a priority queue at each switch (thus, all TCP and UDP packets are treated equally). The results in Figure 6.2(b) show a throughput drop similar to priority-based flow contention, but a slightly different plot for inter-packet arrival times — as expected, the increase in inter-packet delays is not as significant as in priority-based flow contention since all packets get treated equally.

Limitations of existing techniques. The two problems demonstrated above can be



(a) Throughput (left) and inter-packet arrival time (right) of a low-priority TCP flow under priority-based flow contention.



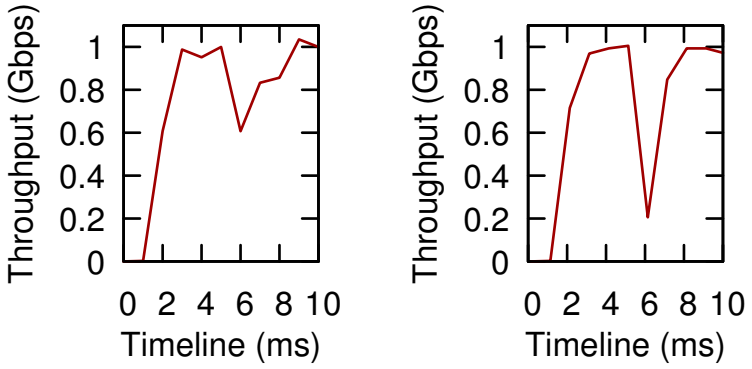
(b) Throughput (left) and inter-packet arrival time (right) of a TCP flow under microburst-based flow contention.

Figure 6.2: Too much traffic problem depicted in Figure 6.1(a). Five UDP burst batches are introduced with a gap of 15 ms between each other. The gray lines highlight the five batches, all of which last for 1 ms. The number in circle denotes the number of UDP flows used in each batch.

detected and diagnosed using specialized switch hardware and interfaces [74]. Without custom designed hardware, these problems can still be detected at the destination of the suffering flow(s), but diagnosing the root cause is significantly more challenging. Packet sampling based techniques would miss microbursts due to undersampling; switch counter based techniques would not be able to differentiate between the priority-based and microburst-based flow contention; and finally, since diagnosing these problems requires looking at flows going to different end-hosts, existing end-host based techniques [71, 99] are insufficient since they only provide visibility at individual end-hosts.

6.2.2 Too many red lights

We now consider the network problem shown in Figure 6.1(b). Our set up uses a low-priority TCP flow from host A to host F (the red flow) that traverses switches S_1, S_2



(a) Throughput of flow A-F at S_1 (b) Throughput of flow A-F at S_2

Figure 6.3: Too many red lights problem depicted in Figure 6.1(b). UDP is used for flows B-D and C-E and TCP for flow A-F.

and S_3. The TCP flow contends with two high-priority UDP flows (B-D and C-E), each lasting for 400μs in a sequential fashion (that is, flow C-E starts right after flow B-D finishes). Consequently, the TCP flow gets delayed for about 400μs at S_1 due to UDP flow B-D and another 400μs at S_2 due to UDP flow C-E.

The result is shown in Figure 6.3. The destination of the TCP flow sees a sudden throughput drop almost down to 200 Mbps. This is a consequence of performance degradation accumulated across two switches S_1 and S_2 — Figures 6.3(a) and 6.3(b) show that the throughput is around 600Mbps at S_1 and around 200 Mbps at S_2 (at around 6 ms time point). In fact, the problem is not limited to reduced throughput for the TCP flow — taken to the extreme, adding more "red lights" can easily result in a timeout for the TCP flow.

Limitations of existing techniques. The too many red lights problem highlights the importance of combining in-network and end-host based approaches to network monitoring and debugging.

Indeed, it is hard for purely in-network techniques to detect the problem — switches are usually programmed to collect relevant flow- or packet-level telemetry information if a predicate (*e.g.*, throughput drop is more than 50% or queuing delay is larger than 1ms) is satisfied, none of which is the case in the above phenomenon. Since the performance of the TCP flow degrades gradually due to contention across switches, the net effect becomes visible closer to the end-host of the TCP flow.

On the other hand, existing end-host based techniques allow detecting the throughput drop (or for that matter, the TCP timeout); however, these techniques do not provide the network visibility necessary to diagnose the gradual degradation of throughput

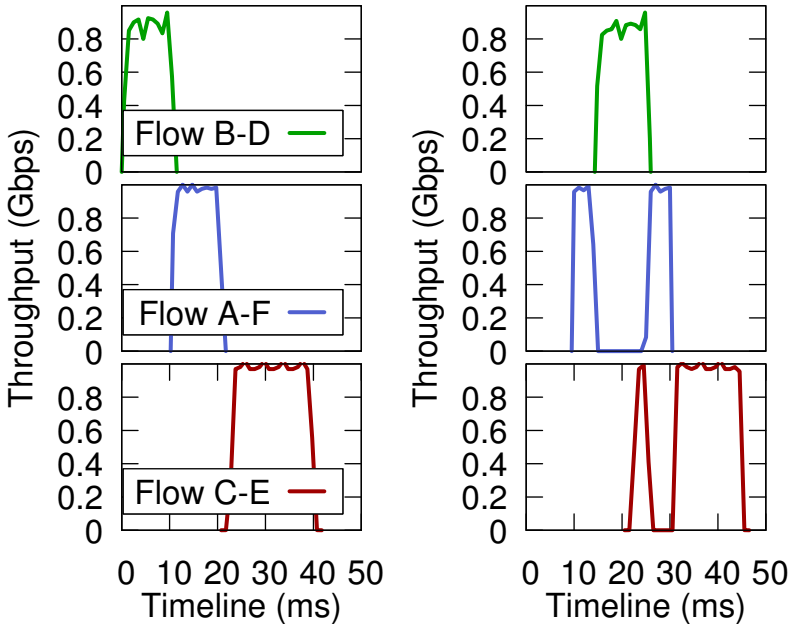


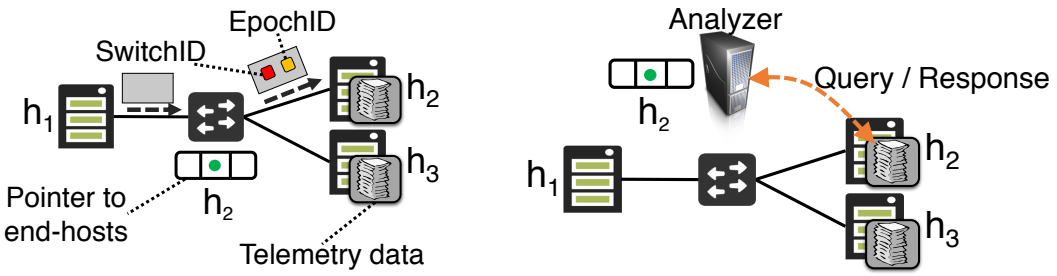
Figure 6.4: Traffic cascades problem depicted in Figure 6.1(c). Throughput of flows (a) without traffic cascades; (b) with traffic cascades. UDP is used for flows B-D and A-F, and TCP for flow C-E.

across switches in the too-many-red-lights phenomenon.

6.2.3 Traffic cascades

Finally, we discuss the traffic cascade phenomenon from Figure 6.1(c). Here, we have three flows, B-D, A-F and C-E, with flow priorities being high, middle and low, respectively. Flows B-D and A-F use UDP and last for 10ms each whereas flow C-E uses TCP and transfers 2MB of data. A cascade effect happens when the high-priority flow B-D affects the middle-priority flow A-F which subsequently affects the low-priority flow C-E. Specifically, if flow B-D and flow A-F do not contend at switch S_1, the flow A-F will depart from switch S_2 before flow C-E arrives resulting in no flow contention in the network (Figure 6.4(a)). However, due to contention of flow B-D and flow A-F at switch S_1 (for various reasons, including B-D being rerouted due to failure on a different path), flow A-F is delayed at switch S_1 and ends up reducing the throughput for flow C-E at switch S_2 (Figure 6.4(b)).

Limitations of existing techniques. Diagnosing the root cause of the traffic cascade problem is challenging for both in-network and for end-hosts based techniques. It not only requires capturing the temporal state (flowIDs and packet priorities for all contending flows) across multiple switches, but also requires to do so even for flows that



(a) (b)

Figure 6.5: (a) Switch divides time into epochs and for each epoch, switch maintains pointer to end-hosts seen in that epoch. In addition, switch also embeds telemetry data (*e.g.*, switchID, epochID) within the packet header. (b) Analyzer uses pointer to end-hosts at switches to identify end-hosts that has relevant telemetry data to debug network problems.

do not observe any noticeable performance degradation (*e.g.*, the B-D flow). Existing in-network and end-host based techniques fall short of providing such functionality.

6.2.4 Other SwitchPointer use cases

There are many other network monitoring and debugging problems for which in-network techniques and end-host based techniques, in isolation, are either insufficient or inefficient (in terms of data plane resources). We have compiled a list of such network problems along with a detailed description of how SwitchPointer is able to monitor and diagnose such problems in [14].

6.3 Overview

SwitchPointer integrates the benefits of end-host based and in-network approaches into an end-to-end system for network monitoring and debugging. To that end, the SwitchPointer system has three main components. This section provides a high-level overview of these components and how SwitchPointer uses these components to monitor and debug network problems. Figure 6.5 gives an overview of SwitchPointer system.

SwitchPointer Switches. The first component runs at network switches and is responsible for three main tasks: (1) embedding the telemetry data into packet header; (2) maintaining pointers to end hosts where the telemetry data for packets processed by the switch are stored; and (3) coordinating with an analyzer for monitoring and debugging network problems.

SwitchPointer switches embed at least two pieces of information in packet headers before forwarding a packet. The first is to enable tracing of packet trajectory, that is, the set of switches traversed by the packet; SwitchPointer uses solutions similar to [98, 99] for this purpose. The second piece of information is to efficiently track contending packets and flows at individual ports over fine-grained time intervals. To achieve this, each SwitchPointer switch divides (its local view of) time into epochs and embeds into the packet header the epochID at which the packet is processed. SwitchPointer can of course use clean-slate approaches like INT [6] to embed epochIDs into packet headers; however, we also present a design in §6.4.1.3 that extends the techniques in [98, 99] to efficiently embed these epochIDs into packet headers along with the packet trajectory tracing information.

Embedding path and epoch information within the packet headers alone does not suffice to debug network problems efficiently. Once a spurious network event is triggered, debugging the problem requires the ability to filter headers contributing to that problem (potentially distributed across multiple end hosts); without any additional state, filtering these headers would require contacting all the end hosts. To enable efficient filtering of headers contributing to the triggered network problem, SwitchPointer uses distributed storage at switches as a directory service — switches store "pointers" to destination end hosts of the packets processed by the switch in different epochs. Once an event is triggered, this directory service can be used to quickly filter out headers for packets and flows contributing to the problem.

Using epochs to track contending packets and flows at switches, and storing pointers to destination end-hosts for packets processed in each epoch leads to several design and performance tradeoffs in SwitchPointer. Indeed, too large an epoch size is not desirable — with increasing epoch size, a switch may forward packets to increasingly many end-hosts within an epoch, leading to inefficiency (at an extreme, this would converge to trivial approach of contacting all end-hosts for filtering relevant headers). Too small an epoch size is also undesirable since with increasing number of epochs, each switch would require either increasingly large memory (SRAM for storing the pointers) or increasingly large bandwidth between the data plane and the control plane (for periodically transferring the pointers to persistent storage).

SwitchPointer achieves a favorable tradeoff between switch memory, bandwidth between the data plane and the control plane, and the efficiency of debugging network problems using a hierarchical data structure, where each subsequent level of the hierarchy stores pointers over exponentially larger time scales. This data structure enables

both real-time (potentially automated) debugging of network problems using pointers for more recent epochs, and offline debugging of network problems by transferring only pointers over coarse-grained time scales from the data plane to the control plane. We discuss this data structure in §6.4.1.1. Maintaining a hierarchy of pointers also leads to challenges in maintaining an updated set of pointers while processing packets at line rate; indeed, a naïve implementation that uses hash tables would require one operation per packet per level of hierarchy to update pointers upon each processed packet. We present, in §6.4.1.2, an efficient implementation that uses perfect hash functions to efficiently maintain updated pointers across the entire hierarchy using just one operation per packet (independent of number of levels in the hierarchical data structure).

SwitchPointer End-hosts. SwitchPointer, similar to recent end-host based monitoring systems [99, 71], uses end hosts to collect and monitor telemetry data carried in packet headers, and to trigger spurious network events. SwitchPointer uses PathDump (Chapter 5) to implement its end-host component; however, this requires several extensions to capture additional pieces of information (*e.g.*, epochIDs) carried in SwitchPointer's packet headers and to query headers. We describe SwitchPointer's end-host component design and implementation in §6.4.2.

SwitchPointer Analyzer. The third component of SwitchPointer is an analyzer that coordinates with SwitchPointer switches and end-hosts. The analyzer can either be colocated with the end-host component, or on a separate controller. A network operator, upon observing a trigger regarding a spurious network event, uses the analyzer to debug the problem. We describe the design and implementation of the SwitchPointer analyzer in §6.4.3.

An example for using SwitchPointer:

We now describe how a network operator can use SwitchPointer to monitor and debug the too many red lights problem from Figure 6.1 and §6.2.2. The destination end-host of the victim TCP flow A-F detects a large throughput drop and triggers the event. The operator, upon observing the trigger, uses the analyzer module to extract the end-hosts that store the telemetry data relevant to the problem — the analyzer module internally queries the destination end-host for flow A-F to extract the trajectory of its packets (switches S_1, S_2 and S_3 in this example) and the corresponding epochIDs, uses this information to extract the pointers from the three switches (for corresponding epochs), and returns the relevant pointers corresponding to the end-hosts that store the relevant headers for flows that contended with the victim TCP flow (D and E in this example).

The operator then filters the relevant headers from the end-hosts to learn that flow A-F contended with flow B-D and C-E, and can interactively debug the problem using these headers. SwitchPointer debugs other problems in a similar way (more details in §6.5).

6.4 SwitchPointer

In this section, we discuss design and implementation details for various SwitchPointer components.

6.4.1 Switches

SwitchPointer provides the network visibility necessary for debugging network problems by using the memory at network switches as a distributed directory service, and by embedding telemetry information in the packet headers. We now describe the data structure stored at and packet processing pipeline of SwitchPointer switches.

6.4.1.1 Hierarchical data structure for pointers

SwitchPointer switches divide their local view of time into epochs and enable tracking of contending packets and flows at switches by storing pointers to destination end-hosts for packets processed in different epochs. SwitchPointer stores these pointers using a hierarchical data structure, where each subsequent level of the hierarchy stores pointers over exponentially larger time scales. We describe this data structure and discuss how it achieves a favorable tradeoff between switch memory (to store pointers) and bandwidth between data plane and control plane (to periodically transfer pointers from switch memory to persistent storage).

Figure 6.6 shows SwitchPointer's hierarchical data structure with k levels in the hierarchy. Suppose the epoch size is α ms. At the lowermost level, the data structure stores α set of pointers, each corresponding to destinations for packets processed in one epoch; thus the set of pointers at the lowermost level provide a per-epoch information on end-hosts storing headers to all contending packets and flows over an α^2 ms period. In general, at level h ($1 \leq h \leq k-1$), the data structure stores α set of pointers corresponding to packets processed in consecutive α^h ms intervals. The top level stores only one set of pointers corresponding to packets processed in last α^k ms of time period.

The hierarchical data structure, by design, maintains some redundant information. For instance, the first set of pointers in level $h+1$ correspond to packets processed in last α^{h+1} ms of time period, collectively similar to all the set of pointers in level h.

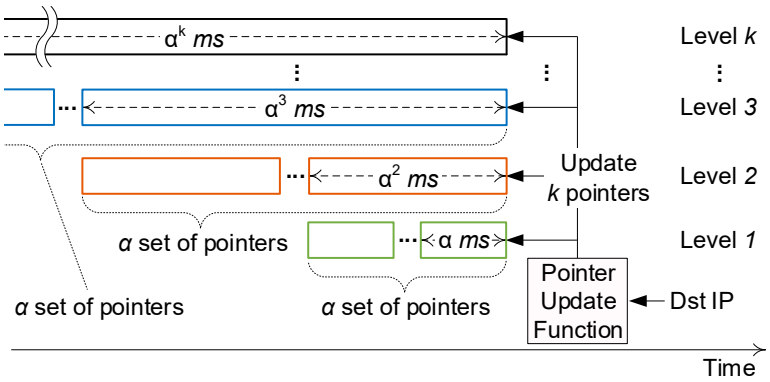


Figure 6.6: SwitchPointer's hierarchical data structure for storing pointers. For each packet that a switch forwards, SwitchPointer stores a pointer to the packet's destination end-host along a hierarchy of k levels. For epoch size α ms, level h $(1 \leq h \leq k-1)$ stores pointers to destination end-hosts for packets processed in last consecutive α^h epochs (that is, α^{h+1} ms) across α set of pointers. The topmost level stores only one set of pointers corresponding to packets processed in last α^k ms.

It is precisely this redundancy that allows SwitchPointer to achieve a favorable tradeoff between switch memory and bandwidth. We return to characterizing this tradeoff below, but note that pointers at the lower level of the hierarchy provide a more fine-grained view of packets and flows contending at a switch and are useful for real-time diagnosis; the set of pointers on the upper levels, on the other hand, provide a more coarse-grained view and are useful for offline diagnosis.

SwitchPointer allows pointers at all levels to be accessed by the analyzer under a *pull model*. For instance, suppose the epoch size is $\alpha = 10$ and the data structure has $k = 3$ levels. Then, each set of pointers at level 1 correspond to 10 ms of time period while those at level 2 correspond to 100 ms of time period. If a network operator wishes to obtain the headers corresponding to packets and flows processed by the switch for last 50 ms (*i.e.*, 5 epochs), it can pull the five most recent set of pointers from level 1; for last 150 ms period, the operator can pull the two most recent pointers from level 2 (which, in fact, correspond to 200 ms time period). In addition to supporting access to the hierarchical data structure using a pull model, SwitchPointer also *pushes* the topmost level of pointers to the control plane for persistent storage every α^k ms which can then be used for offline diagnosis of network events. The toplevel pointers provide coarse-grained view of contending packets and flows at switches which may be sufficient for offline diagnosis but using a push model only for the topmost level pointers significantly reduces the requirements on bandwidth between the data plane and the control plane.

Tradeoff. The hierarchical data structure, as described above, exposes a tradeoff between switch memory and the bandwidth between the data plane and the control plane via two parameters — k and α. Specifically, let the storage needed by a set of pointers to be S bits (this storage requirement depends on the maximum number of end-hosts in the network, and is characterized in next subsection); Then, the overall storage needed by the hierarchical data structure is $\alpha \cdot (k-1) \cdot S + S$ bits. Moreover, since only the topmost pointer is pushed from the data plane to the control plane (once every α^k ms), the bandwidth overhead of SwitchPointer is bounded by $S \times (10^3/\alpha^k)$ bps. For a fixed network size (and hence, fixed S), as k and α are increased, the memory requirements increase and the bandwidth requirements decrease. We evaluate this tradeoff in §6.6 for varying values of k and α; however, we note that misconfiguration of k and α values may result in longer diagnosis time (the analyzer may touch more end-hosts to filter relevant headers) but does not result in correctness violation.

6.4.1.2 Maintaining updated pointers at line rate

We now describe the technique used in SwitchPointer to minimize the switch memory requirements for storing the hierarchical data structure and to minimize the number of operations performed for updating all the levels in the hierarchy upon processing each packet.

Strawman: a simple hash table. A plausible solution for storing each set of pointers in the hierarchical data structure is to use a hash table. However, since SwitchPointer requires updating k set of pointers upon processing each packet (one at each level of hierarchy), using a standard hash table would require k operations per packet in the worst case. This may be too high a overhead for high-speed networks (*e.g.*, with 10Gbps links). One way to avoid such overhead is to use hash tables with large number of buckets so as to have a negligible collision probability. Using such a hash table would reduce the number of operations per packet to just one (independent of number of levels in the hierarchy); however, such a hash table would significantly increase the storage requirements. For instance, consider a network with m destinations; given a hash table with n buckets, the expected number of collisions under simple uniform hashing is $m-(n-n(1-1/n)^m)$. Suppose that $m = 100$K and the target number of collisions is $0.001m$ (*i.e.*, 0.1% of 100K keys). To achieve this target, the number of buckets in the hash table should be close to 50 million, 500× larger than the number of keys. Thus, this strawman approach becomes quickly infeasible for our hierarchical

data structure — it would either require multiple operations per packet to update the data structure or would require very large switch memory.

Our solution: Minimal perfect hash function. Our key observation is that the maximum number of end-hosts in a typical datacenter is known *a priori* and that it changes at coarse time scales (hours or longer). Therefore, we can construct a minimal perfect hash function to plan ahead on the best way to map destinations to buckets to avoid hash collisions completely. In fact, since each level in the hierarchy uses the same perfect hash function, SwitchPointer needs to perform just one operation per packet to find the index in a bit array of size equal to the maximum number of destinations; the same index needs to be updated across all levels in the hierarchy. Upon processing a packet, the bit at the same index across the bit array is set in parallel. Lookups are also easy — to check if a packet to a particular destination end-host was processed in an epoch, one simply needs to check the corresponding bit (given by the perfect hash function) in the bit array.

The minimal perfect hash function provides $O(1)$ update operation and expresses a 4-byte IP address with 1 bit (*e.g.*, 100Kbits for 100K end-hosts). While an additional space is required to construct a minimal perfect hash function, it is typically small (70 KB and 700 KB for for 100K and 1M end-hosts respectively; see §6.6.1). Moreover, while constructing a perfect hash function is a computationally expensive task, small storage requirement of perfect hash tables allow us to recompute the hash function only at coarse-grained time intervals — temporary failures of end-hosts do not impact the correctness since the bits corresponding to those end-hosts will simply remain unused. For resetting pointers at level h, an agent at the switch control plane updates a register with the memory address of next pointer every α^h ms and resets its content. The agent conducts this process for pointers at all levels.

6.4.1.3 Embedding telemetry data

SwitchPointer requires two pieces of information to be embedded in packet headers. The first is the trajectory of a packet, that is, the set of switches (*i.e.*, switchIDs) traversed by the packet between the source and the destination hosts. The second is epoch information (*i.e.*, epochID) on when a packet traverses those switches.

SwitchPointer extends the link sampling idea from CherryPick and PathDump, to efficiently enable packet trajectory tracing and epoch embedding for commonly used datacenter network topologies (*e.g.*, clos networks like fat-tree, leaf-spine and VL2)

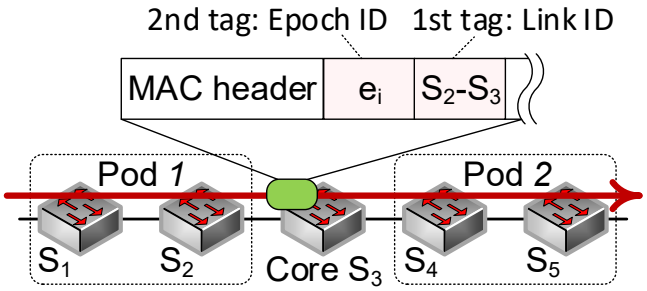


Figure 6.7: Telemetry data embedding using two VLAN tags using a modified version of the technique used in CherryPick. See §6.4.1.3 and §6.4.2.1 for discussion.

without any hardware modifications. Specifically, it is shown in Chapter 4 that an end-to-end path in typical datacenter network topologies can be represented by selectively picking a small number of key links. For instance, in a fat-tree topology the technique reconstructs a 5-hop end-to-end path by selecting only one aggregate-core link and embedding its linkID into the packet header. For embedding epochIDs in addition to the linkID, we extend the technique that relies on IEEE 802.1ad double tagging. When a linkID is added to the packet header using a VLAN tag, we add an epochID using another tag (see Figure 6.7).

The number of rules for embedding linkID increases linearly with respect to the number of switch ports whereas only one flow rule is for epochID embedding. However, the switch needs a rule update once every epoch — as the epoch changes, the switch should be able to increment epochID and add a new epochID for incoming packets. A commodity OpenFlow switch that we use is capable of updating flow rules every 15 ms, giving us a lower bound on α granularity for commodity switches.

We note that the limitations on supported topologies and α granularity in our implementation over commodity switches are merely an artifact of today's switch hardware — it is possible to use SwitchPointer with clean-slate solutions such as INT [6] to support trajectory tracing and epoch embedding over arbitrary topologies.

6.4.2 End-hosts

SwitchPointer uses PathDump to collect and monitor telemetry data carried in packet headers, and to trigger spurious network events. In this subsection, we discuss the extensions needed in PathDump to capture additional pieces of information (*e.g.*, epochIDs) carried in SwitchPointer's packet headers and to query headers.

6.4.2.1 Decoding telemetry data

When a packet arrives at its destination, the destination host extracts the telemetry data from the packet header. If the network supports clean-slate approaches like INT [6], this is fairly straight forward. For implementation using commodity switches (using techniques discussed in §6.4.1.3), the host extracts two VLAN tags containing the switchID and the epochID associated with the switchID. Using the switchID, the end-to-end path can be constructed using techniques in CherryPick and PathDump, giving us a list of switches visited by the packet. Next, we decide a list of epochIDs for each of those switches. However, since only one epochID is available at the end-host, it is hard to determine the missing epochIDs for those switches correctly. Thus, we set a range of epochs that the switches should examine. Specifically, we may need to examine max_delay/α number of pointers at each switch due to uncertainty in epoch identification.

Let Δ denote the a maximum one hop delay and ϵ be a maximum time drift among all switches. Given epochID e_i of switch S and an end-to-end path, the epochIDs for switches along the path are identified as follows.

For the upstream switches of switch S, the epoch range is $[e_i-(\epsilon+j\cdot\Delta)/\alpha, e_i+\epsilon/\alpha]$ and for the downstream switches of S, it is $[e_i-\epsilon/\alpha, e_i+(\epsilon+j\cdot\Delta)/\alpha]$, where j is hop difference between an upstream (or downstream) switch and switch S. Suppose $\alpha = 10$ ms, $\epsilon = \alpha$ and $\Delta = 2\cdot\alpha$. For instance, in the example of Figure 6.7, we set $[e_i-3, e_i+1]$ for switch S_2, $[e_i-1, e_i+3]$ for S_4, and so forth. This provides a reasonable bound due to two reasons. First, a maximum queuing delay is within tens of milliseconds in the datacenter network (*e.g.*, 14 ms in [20]). Second, millisecond-level precision is sufficient as SwitchPointer epochs are of similar granularity.

6.4.2.2 Event trigger and query execution

The end-host also has an agent that communicates with and executes queries on behalf of the analyzer. The agent is implemented using a microframework called flask [4], and implements a variety of techniques (similar to those in existing end-host based systems, PathDump and Trumpet [71]) to monitor spurious network events.

6.4.3 Analyzer

The analyzer is also implemented using flask microframework. It communicates with both switch and end-host agents. From the switch agent, the analyzer obtains pointers

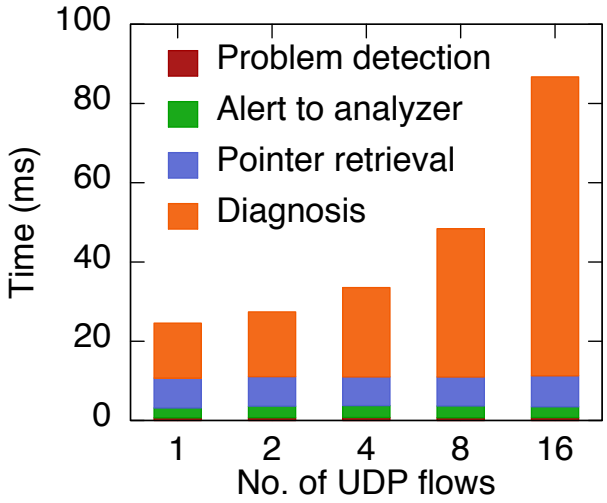


Figure 6.8: Debugging time of the priority-based flow contention problem depicted in Figure 6.2(a). SwitchPointer is able to monitor and debug the problem in less than 100ms. We provide a break down of the diagnosis latency later in Figure 6.13.

to end-hosts for epoch(s). From the end-host agent, it receives alert messages, and exchanges queries and responses. Another responsibility is that it constructs a minimal perfect hash function whenever there are permanent changes in the number of end-hosts in the network, especially when end-hosts are newly added. It then distributes the minimal perfect hash function to all the switches in the network. The analyzer also does pre-processing of pointers by leveraging network topology, flow rules deployed in the network, etc. For example, to diagnose the network problem experienced by a flow, the analyzer filters out irrelevant end-hosts in the pointer if the paths between the flow's source and those end-hosts do not share any path segment of the flow. This way, the analyzer reduces search radius, *i.e.*, number of end-hosts that it has to contact.

6.5 Applications

In this section, we demonstrate some key monitoring applications SwitchPointer supports.

6.5.1 Too much traffic

We debug the problem discussed in §6.2 using SwitchPointer. This problem include two different cases: (i) priority-based flow contention and (ii) microburst-based flow contention. The debugging processes of both cases are similar; the only difference is the former case requires the analyzer to examine flow's priority value. Thus, we only

discuss the former case.

Figure 6.8 shows the breakdown of times it took to diagnose the priority-based flow contention case. First, we instrument hosts with a simple trigger that detects drastic throughput changes. The trigger measures throughput every 1 ms interval and generates an alert to the analyzer if throughput drop is more than 50%. The problem detection takes less than 1 ms, thus almost invisible from the figure (3-4 ms for the microburst-based contention case). Then, it takes 2-3 ms to send the analyzer an alert and to receive an acknowledgment. The alert contains a series of <switchID, a list of epochIDs, a list of byte counts per epoch> tuples that tell the analyzer when and where packets of the TCP flow visit. The analyzer uses the switchIDs and epochIDs, and obtains relevant pointers from switches. In this scenario, it only takes about 7-8 ms to retrieve a pointer from one switch.

Next, the analyzer learns hosts encoded in the pointer, and diagnoses the problem by consulting them; it collects telemetry data such as UDP flow's priority, the number of bytes in UDP flow during the epoch when the TCP flow experiences high delay. The analyzer finally draws a conclusion that the presence of high-priority UDP flows aggravated the performance of the low-priority TCP flow. As shown in Figure 6.8, the time for the diagnosis increases as the number of consulted hosts (*i.e.*, each UDP flow is destined to a different host) increases. Although not too large, the diagnosis overhead inflation pertains to the implementation of connection initiation; we discuss this matter and its optimization in §6.6.2.

6.5.2 Too many red lights

This problem illustrated in Figure 6.1(b) (for its behavior, see Figure 6.3) requires spatial correlation of telemetry data across multiple switches for diagnosis. While this problem is challenging to existing tools, SwitchPointer easily diagnoses it as follows.

First, destination F triggers an alert to the analyzer in no time ($\sim$1 ms) by using our throughput drop detection heuristic introduced in §6.5.1. The alert contains IDs for switches S_1,S_2 and S_3 and their corresponding epochID ranges. The analyzer contacts all of the switches and retrieves pointers that match the epoch IDs for each switch in 10 ms, and then conducts diagnosis (another 20 ms) by obtaining telemetry data for UDP flows B-D and C-E from hosts D and E, respectively. The analyzer finds out that low (A-F) and high prority (B-D and C-E) flows have at least one common epochID, and finally concludes (in about 30 ms) that both flows B-D and C-E contributed to the actual impact on the TCP flow.

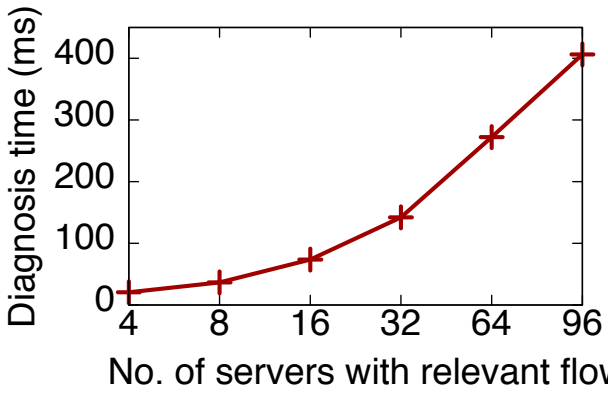


Figure 6.9: Latency for diagnosing load imbalance problem.

6.5.3 Traffic cascades

This problem is a more challenging problem to existing tools because debugging it requires spatial and temporal correlation of telemetry data (see Figure 6.1(c) for the problem illustration and Figure 6.4(b) for its behavior). SwitchPointer diagnoses the problem as follows.

First, the low-priority TCP flow C-E observes a large throughput drop at around 26 ms (see Figure 6.4(b)) and triggers an alert along with switchIDs and corresponding epoch details. Then, the analyzer retrieves pointers that match with epochIDs from S_2 and S_3, contacts F and finds out the presence of middle-priority flow A-F on S_2 caused the contention in ∼25 ms. Since flow A-F has middle-priority, the analyzer subsequently examines pointers from switches (*i.e.*, S_1 and S_2) along the path of flow A-F in order to see whether or not the flow was affected by some other flows. From a pointer from switch S_1, the analyzer comes to know that flow B-C made flow A-F delayed, which in turn had flows A-F and C-E collide. This part of debugging takes another 25 ms. Hence, the whole process takes about 50 ms in total.

Of course, in a large datacenter network, debugging this kind of problem can be more complex than the example we studied here. Therefore, in practice the debugging process may be an off-line task (with a pointer at a higher level that covers many epochs) rather than an online task. However, independent of whether it is an off-line or online task, SwitchPointer showcases, with this example, that it is feasible to diagnose network problems that need both spatial and temporal correlation.

6.5.4 Load imbalance diagnosis

To demonstrate the way SwitchPointer works for diagnosing load imbalance, we create the same problematic setup used in Section 5.4.2. In that setup, a switch that is con-

figured to malfunction, forwards traffic unevenly to two egress interfaces; specifically, packets from flows whose size is less than 1 MB are output on one interface; otherwise, packets are forwarded to the other interface. We vary the number of flows from 4 to 96. Each flow is destined to a different end-host. Using this setup, we can understand how the number of end-hosts contacted by the analyzer impacts SwitchPointer's performance.

The debugging procedure is similar to that of other problems we already studied. This problem is detected by monitoring interface byte counts per second. The analyzer fetches the pointers corresponding to the most recent 1 sec. It then obtains the end-hosts in the pointers, and sends them a query that computes a flow size distribution for each of the egress interfaces of the switch. Finally, the analyzer finds out that there is a clean separation in flow size between two distributions. Figure 6.9 shows the diagnosis time of running a query as a function of the number of end-hosts consulted by the analyzer. The diagnosis time increases almost linearly as the analyzer consults more end-hosts. Since this trend comes from the same cause, we refer to §6.6.2 for understanding individual factors that contribute to the diagnosis time.

6.6 Evaluation

We prototype SwitchPointer on top of Open vSwitch [8] over Intel DPDK [3]. To build a minimal perfect hash function, we use the FCH algorithm [42] among others in CMPH library [2]. We also implement the telemetry data extraction and epoch extrapolation module (§6.4.2.1) on OVS. The module maintains a list of flow records; one record consists of the usual 5-tuple as flowID, a list of switchIDs, a series of epoch ranges that correspond to each switchID, byte/packet counts and a DSCP value as flow priority. This flow record is initially maintained in memory and flushed to a local storage, implemented using MongoDB [7]. We now evaluate SwitchPointer in terms of switch overheads and query performance under real testbeds that consist of 96 servers.

6.6.1 Switch overheads

To quantify switch overheads, we vary epoch duration (α ms), the number of levels in a pointer (k), the number of IP addresses (n) and packet size (p). We set up two servers connected via a 10GE link. From one server, we generate 100K packets, each of which has a unique destination IP (hence, 100K flows); we play those 100K packets repeatedly to the other server where SwitchPointer is running using one 3.1 GHz CPU core. Under the setup, we measure i) throughput, ii) the amount of memory to keep

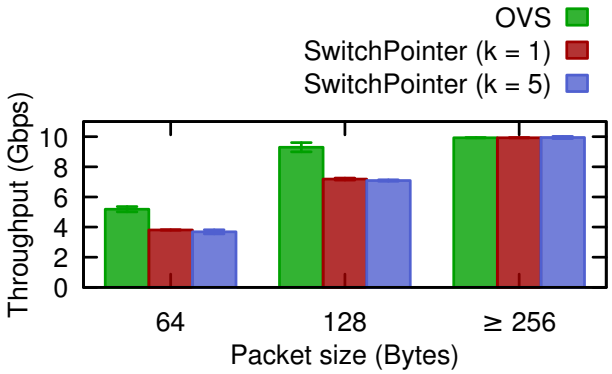


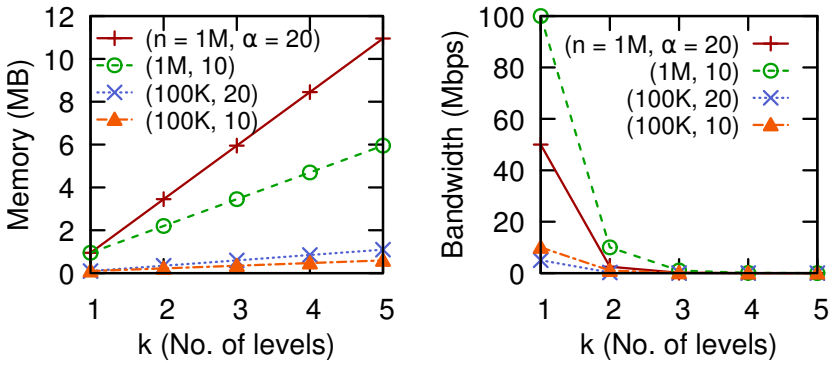
Figure 6.10: For smaller packet sizes, SwitchPointer is unable to sustain line rate due to overheads of perfect hash function. SwitchPointer is able to achieve line rate for a 10GE interface for packets of size 256bytes and more.

pointers on data plane, iii) bandwidth to offload pointers from SRAM (data plane) to off-chip storage (control plane), and iv) pointer recycling period.

Throughput. We compare SwitchPointer's throughput with that of vanilla OVS (baseline) over Intel DPDK. We set $k = 1$ and 5. Here one pointer of SwitchPointer is configured to record 100K unique end-hosts. We then measure the throughput of SwitchPointer while varying p. Our current implementation in OVS processes about 7 million packets per second. From Figure 6.10, we observe that OVS and both configurations of SwitchPointer achieve a full line rate ($\sim$9.99 Gbps) when $p \geq 256$ bytes. In contrast, when $p < 256$ bytes, both OVS and SwitchPointer face throughput degradation. For example, when p is 128 bytes, OVS achieves about 9.29 Gbps whereas SwitchPointer's throughput is about 22% less than that of OVS. However, since an average packet size in data centers is in general larger than 256 bytes (*e.g.*, 850 bytes [24], median value of 250 bytes for hadoop traffic [87]), the throughput of SwitchPointer can be acceptable. We also envision that a hardware implementation atop programmable switch [25, 51] would eliminate the limitation of a software version.

Memory. Perfect hash functions account for about 70 KB ($n = 100$K) and 700 KB ($n = 1$M). In addition, n also governs the pointer's size: 12.5 KB ($n = 100$K) and 125 KB ($n = 1$M). Together SwitchPointer requires to have 82.5 KB and 825 KB, respectively. These are the minimum amount of memory requirement for SwitchPointer. Figure 6.11(a) shows the memory overhead; the memory requirement increases in proportion to each of k and α. When $n = 1$M, $\alpha = 10$ and $k = 3$, SwitchPointer consumes 3.45 MB; for $n = 100$K, it is only 345 KB.

Bandwidth. In contrast to memory overhead, the bandwidth requirement of system



(a) Memory overhead (b) Bandwidth overhead

Figure 6.11: Overheads of SwitchPointer. At (n, α) in the legend, n denotes the maximum number of IP addresses traced by SwitchPointer, and α is an epoch duration in ms.

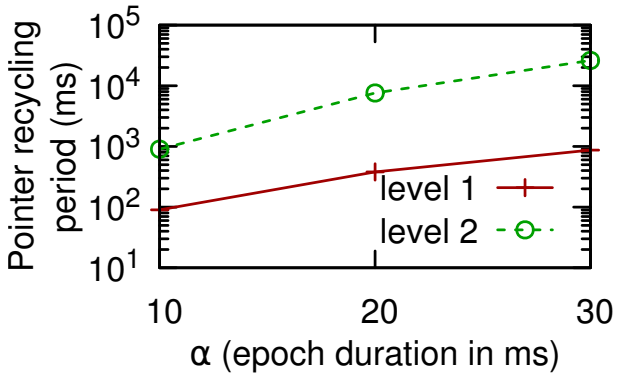
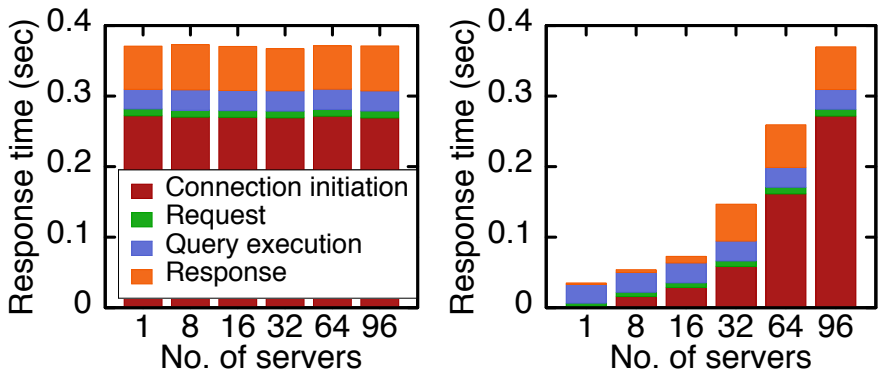


Figure 6.12: Recycling period of a pointer when $k = 3$.

bus between SRAM and off-chip storage decreases as we increase k and α because larger values of those parameters make the pointer flush less frequent. In particular, k has a significant impact in controlling the bandwidth requirement; increasing it drops the requirement exponentially. For $n = 1$M and $\alpha = 10$ (the most demanding setting in Figure 6.11(b)), the bandwidth requirement reduces from 100 Mbps ($k = 1$) to 10 Mbps ($k = 2$).

The results in Figures 6.11(a) and 6.11(b) present a clear tradeoff between memory and bandwidth. Depending on the amount of available resources and user's requirements, SwitchPointer provides a great flexibility in adjusting its parameters. For instance, if memory is a scarce resource, it may be better to keep $k \leq 3$ and $\alpha \leq 10$.

Pointer recycling period. Except for top level pointers, pointers are recycled after all the pointers on the same layer are used. The pointer recycling period at level h is expressed as $\alpha(\alpha^h - 1)$ ms where $1 \leq h < k$. Figure 6.12 shows a tradeoff between α



(a) PathDump (b) SwitchPointer

Figure 6.13: Top-100 query response time. Most of SwitchPointer latency overheads are due to connection initiation requests from the analyzer to the end-hosts and can be improved with a more optimized RPC implementation.

and k. As expected, the recycling period exponentially increases as the level increases (when $\alpha = 10$, the recycling period of a pointer at level 1 is 90 ms and it is 900 ms at level 2). Because too small α may always let SwitchPointer end up accessing a higher-level pointer, α should be chosen carefully.

6.6.2 Query performance

We now evaluate the query performance of SwitchPointer, which we compare with that of PathDump (baseline). We run a query that seeks top-k flows in a switch in our testbed where there are 96 servers. The key difference between SwitchPointer and PathDump is that SwitchPointer knows which end-hosts it needs to contact but PathDump does not. Thus, PathDump executes the query from all the servers in the network. To see the impact of the difference, we vary the number of servers that contain telemetry data of flows that traverse the switch.

From Figure 6.13 we observe that the response time of SwitchPointer gradually increases as the number of servers increases. On the other hand, PathDump always has the longest response time as it has to contact all 96 servers anyway. Both of them only have a similar response time when all the servers have relevant flow records and thus SwitchPointer has to contact all of them.

A closer look reveals that most of the response time is because of connection initiation for both SwitchPointer and PathDump. In our current implementation, the analyzer creates one thread per server to initiate connection when a query should be executed. This on-demand thread creation delays the execution of query at servers. This is an im-

plementation issue, not a fundamental flaw in design. Thus, it can be addressed with proper technique such as thread pull management. However, since PathDump must contact all the servers regardless of whether or not the servers have useful telemetry data, it wastes servers' resources. On the contrary, SwitchPointer only spends right amounts of server resources, thus offering a scalable way of query execution.

6.7 Limitations

Supporting arbitrary range queries. SwitchPointer switch divides time into epochs and maintains pointers to end-hosts that the switch has seen in that epoch. Section 6.4 provided discussion on the right epoch duration (too small vs, too large), and the need for hierarchical data structure to store pointers. Such a data structure supports both real-time and off-line queries. In brief, for a given time range query, SwitchPointer responds with a minimum number of pointers that covers the range. However, range queries sometimes may not sufficient to accurately identify contending flows and packets. So, we envision support for arbitrary time range queries (not multiple of epoch duration (α) set initially) would improve usability of SwitchPointer. More work on the design is required to close the gap between epoch duration and arbitrary time range queries.

Inaccuracies in determining epochs. There are many ways to embed telemetry data (switch ID, epoch ID) into packet header. One idea is to use a clean state approach like INT [51]. But INT has packet header space limitation in current deployments. In this work, SwitchPointer extends the link sampling technique presented in CherryPick. It works for popular data center topologies using commodity OpenFlow-compatible switches. But, SwitchPointer does not provide the exact epoch at which a packet is processed by pod switches (not present in core layer). Instead, it exploits the fact that generally we can set bounds on clock difference and maximum queuing delays between any pair of devices in a data center. So, using CherryPick, SwitchPointer can only provides a range of switch epoch IDs for pod switches. It requires more evaluation in a controlled environment to understand the impact of time drift or queuing delay variations on determining the epochs. Of course, embedding techniques like INT would overcome inaccuracies and also simplifies determining the switch epoch IDs.

Scalability. Due to limited resources at servers in our testbed, we could only evaluate SwitchPointer's throughput with 100K flows on a 10GE link. Our testbed has quad-core servers equipped with modest memory (4MB) and 10GE interface cards. For

scalability, throughput test could have evaluated up to one million flows (requires more memory to generate and consume traffic at line rate) on a 40GE link.

6.8 Summary

This chapter presents SwitchPointer, a system that integrates the benefits of end-host based approaches and in-network approaches to network monitoring and debugging. SwitchPointer uses end-host resources and programmability to collect and monitor telemetry data, and to trigger spurious network events. The key technical contribution of SwitchPointer is to enable network visibility by using switch memory as a "directory service" — SwitchPointer switches use a hierarchical data structure to efficiently store *pointers* to end-hosts that store relevant telemetry data. Using experiments on real-world testbeds, we have shown that SwitchPointer efficiently monitors and debugs a large class of network problems, many of which were either hard or even infeasible with existing designs.

Chapter 7

Fault Localization in Large-Scale Network Policy Deployment

7.1 Introduction

Software-defined Networking (SDN) enables flexible and intent-based policy management [82, 92, 15, 41, 69, 102, 9]. As programmability offered by SDN makes network management easier, troubleshooting network problems become increasingly challenging. An ideal troubleshooting tool for admins should allows to quickly detect, localize, inspec, fix the network problem. In specific, the tool should quickly nail downs to the part of the policy that the admin should further look at to fix observed failures. Towards this direction, we built Scout, an end-to-end system that automatically pinpoints not only faulty policy objects, but also physical-level failures, the root cause for policy objects to become faulty.

In the existing policy management frameworks [82, 15], low-level rules are built from policy objects (in short, objects) such as marketing group, DB tier, filter, and so on. Our study on a production cluster reveals that even one object can be used to create TCAM rules for over thousands of endpoints. This implies that a fault of that single object can lead to a communication outage for those numerous endpoints. In order to find which particular part of the policy failure is the main cause, examining all of the TCAM rules associated with the endpoints is a needle-in-a-haystack problem and would be tedious.

Scout design. We call the problem of finding out the impaired parts of the policy as a *network policy fault localization problem*, which we tackle via *risk modeling* [62]. We model risks as simple bipartite graphs that capture dependencies between risks

(*i.e.*, objects) and nodes (*e.g.*, endpoints or end user applications) that rely on those risks. We then annotate the risk models for those risks and nodes that are associated with the observed failures. Using those models, we devise a greedy fault localization algorithm that outputs a hypothesis, a minimum set of most-likely faulty policy objects (*i.e.*, risks) that explains most of the observed failures.

Scout challenges. At first glance, solving this policy fault localization problem looks straightforward as a similar problem has been studied for IP networks [62]. However, there are two key challenges. First, it is difficult to represent risks in the network policy as a single model. Solving many risk models can be computationally expensive. In our modeling, we fortunately require two risk models only: switch risk model and controller risk model (§7.3.2). We make the two models based on our observation that faults of policy objects occur at two broad layers (controller and switch). If the controller malfunctions, unsuccessful policy deployment can potentially affect all the switches in the network (thus, controller risk model). On the contrary, a policy deployment failure can be limited to a switch [112, 44] if that switch only becomes faulty (thus, switch risk model).

Our second challenge stems from the fact that the degree of impact on endpoints caused by a faulty object varies substantially. When a fault event occurs, some objects are responsible for all of the impacted endpoints. On the contrary, some other objects cause trouble to a small fraction of total number of endpoints that rely on them. This variety makes accurate fault localization difficult. An existing algorithm [62] tends to choose policy objects in the former case while it treats objects in the latter case as input noise. However, in our problem, some objects do belong to the latter case. To handle this issue, Scout employs a 2-stage approach; it first picks objects only if all of their dependent endpoints are in failure; next, for (typically a small number of) objects left unexplained in the risk model, it looks up the change logs (maintained at a controller) and selects the objects to which some actions are recently applied (§7.4). Despite its simplicity, this heuristic effectively localizes faults (§7.6).

Scout contributions. Overall, the main contributions addressing fault localization problem in this thesis are:

- We introduce and study a network policy fault localization problem (§7.2). This is a new problem that gained little attention but is of utmost importance in operating a policy management framework safely.
- We introduce two risk models (switch and controller risk models) that precisely

capture the characteristics of the problem and help its formulation.

- We devise a policy fault localization algorithm that quickly narrows down a small number of suspicious faulty objects (§7.4). We then design and implement Scout (§7.5), a system that conducts an end-to-end automatic fault localization from failures on policy objects to physical-level failures that made the objects faulty.

- We evaluate Scout using a real production cluster and extensive simulations (§7.6). Our evaluations show that Scout achieves 20-50% higher accuracy than an existing solution and is scalable. Scout runs a large-scale controller risk model of a network with 500 leaf switches, under 130 seconds in a commodity machine.

This chapter is organized as follows: Section 7.2 presents the background to understand network policy deployment procedure, Section 7.3 formulates the fault localization problem with shared risk models, Section 7.4 describes the proposed algorithm that localize faulty policy objects, and Section 7.5 presents an end-to-end system, Scout that conducts analysis from faulty policy object localization to physical-level root cause diagnosis.

7.2 Background

7.2.1 Network policy

In general network policies dictate the way traffic should be treated in a network. In managing network policies, tenant/admins should be able to express their intent on traffic via a model and to enforce the policies at individual network devices. To enable more flexible composition and management of network policies, several frameworks [82, 15, 9] present the network policies in an abstracted model (*e.g.*, a graph) that describes communication relationships among phyiscal/logical entities such as servers, switches, middleboxes, VMs, etc.

Intent illustration. As an example, consider a canonical 3-tier web service that consists of Web, App and DB servers (or VMs) as shown in Figure 7.1(a). Here the tenant intent is to allow communication on specific ports between the application tiers, *i.e.*, port 80 between Web and App, ports 80 and 700 between App and DB. A network policy framework transforms intent of users (tenant, network admins, etc.) into an abstracted policy as illustrated in Figure 7.1(b).

Network policy presentation. For driving our discussion, we here apply a network policy abstraction model in [15], which is quite similar to other models (*e.g.*, GBP [9],

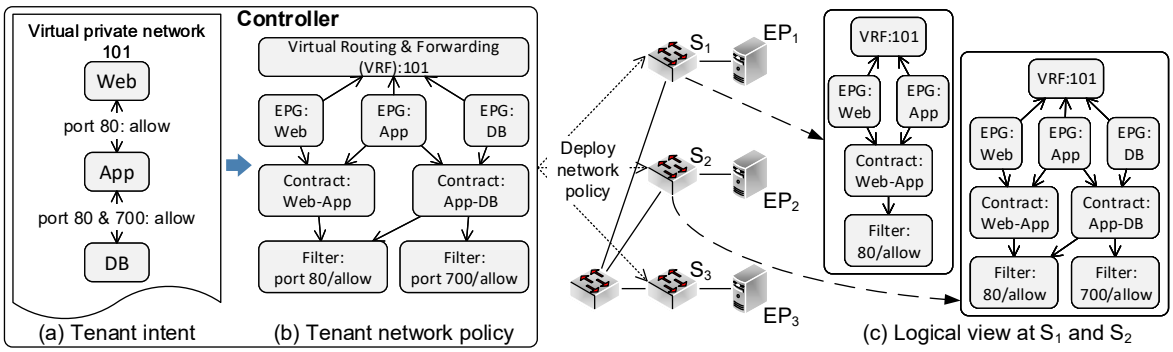


Figure 7.1: An example of network policy management framework. EP stands for endpoint, and EPG denotes endpoint group.

PGA [82]); and our work for localizing faults in network policy management is agnostic to policy abstraction model itself. Figure 7.1(b) illustrates a network policy (as a graph represented with policy objects) transformed from the tenant intent shown in Figure 7.1(a). We discuss each of those policy objects next.

An *endpoint group (EPG)* represents a set of *endpoints* (EPs), *e.g.*, servers, VMs, and middleboxes, that belong to the same application tier. A *filter* governs access control between EPGs. This policy entity takes a whitelisting apporach, which by default blocks all traffic in the absence of filters.

A mapping between EPGs and filters is indirectly managed by an object called *contract*, which serves as a glue between EPGs and filters. A contract defines what filters need to be applied to which EPGs. Thus, a contract enables easy modification of filters. For example, in Figure 7.1(b), suppose EPG:App and EPG:DB no longer need to talk to each other on port 700. One can simply remove "Filter: port 700/allow" from the Contract:App-DB without need to modify the contract.

Finally, the scope of all EPGs in a tenant policy is defined using a layer-3 virtual private network, realized with a virtual routing and forwarding (VRF) object.

Network policy deployment. A network policy should be realized through deployment. A centralized controller maintains the network policy and makes changes on it. When updates (add/delete/modify) on a network policy are made, the controller compiles the new policy and produces instructions that consist of policy objects and the update operations associated with the objects. The controller then distributes the instructions to respective switch agents. The switch agents also keep a local view on the network policy to which the instructions from the controller are applied. The switch agents transform any changes on the logical view into low-level TCAM rules.

Consider a network topology (Figure 7.1) where EP_1 is attached to switch S_1,

No.	Rule	Action
1	VRF:101,Web,App,Port80	Allow
2	VRF:101,App,Web,Port80	Allow
3	VRF:101,App,DB,Port80	Allow
4	VRF:101,DB,App,Port80	Allow
5	VRF:101,App,DB,Port700	Allow
6	VRF:101,DB,App,Port700	Allow
7	*,*,*,*	Deny

Figure 7.2: TCAM rules in switch S_2. Note that here a rule is annotated with object types in it for ease of exposition.

EP_2 to S_2 and EP_3 to S_3. Let us assume that $EP_1 \in EPG{:}Web$, $EP_2 \in EPG{:}App$ and $EP_3 \in EPG{:}DB$. Putting it altogether, the controller sends out the instructions about $EPG{:}Web$ to switch S_1 (as EP_1 is connected to S_1), those about $EPG{:}App$ to switch S_2, and so forth. As the three switches receive the instructions on those EPGs for the first time, they build a logical view from scratch (see Figure 7.1(c) for example). Hence, a series of add operations invoke TCAM rule installations in each switch. Figure 7.2 shows access control list (ACL) rules rendered in TCAM of S_2.

7.2.2 Network state inconsistency

Network policy enforcement is by nature a distributed process and involves the management of three key elements: (i) a global network policy at controller, (ii) a local network policy at switch agent, and (iii) TCAM rules generated from the local policy. Ideally, the states among these three elements should be *equivalent* in order for the network to function as intended by admins.

In reality, these elements may not be in an equivalent state due to a number of reasons. A switch agent may crash in the middle of TCAM rule updates. A temporal disconnection between the controller and switch agent during the instruction push. TCAM has insufficient space to add new ACL rules, which renders the rule installation incomplete. The agent may run a local rule eviction mechanism, which even worsens the situation because the controller may be unaware of the rules deleted from TCAM. Even TCAM is simply corrupted due to hardware failure. All of these cases can create a state mismatch among controller, switch agent and TCAM level, which compromises the integrity of the network.

One approach to this issue is to make network policy management frameworks

more resilient against failures. However, failures are inevitable, so is the network state inconsistency.

7.3 Shared Risks in Network Policy

We exploit shared risk models for our network policy fault localization problem. The shared risk model has been well studied in IP networks [62]. For instance, when a fibre optic cable carries multiple logical IP links, the cable is recognized as a *shared risk* for those IP links because the optical cable failure would make the IP links fail or perform poorly.

Deploying a network policy also presents shared risks. A network policy comprises policy objects (such as VRF, EPGs, contract, filter, etc). The relationship among those objects dictates how a network policy must be realized. If an object is absent or ill-represented in any of controller, switch agent and TCAM layers, all of EPG pairs that rely on that object would be negatively impacted. Thus, these policy objects on which a set of EPG pairs rely are shared risks in the network policy deployment.

Figure 7.2 depicts that a TCAM rule is expressed as a combination of objects presented in a logical model at switch S_2. If the 5th and 6th TCAM rules in the figure are absent from TCAM, all the traffic between EPG:App and EPG:DB via port 700 would be dropped. The absence of correct rules boils down to a case where one or more objects are not rendered correctly in TCAM; a corrupted TCAM may write a wrong VRF identifier (ID) or EPG ID for those rules; S_2 may drop the filter 'port 700/allow' from its logical view due to software bug. Such absence or mispresentation of objects directly affect the EPG pairs that share the objects. Thus, shared risk objects for App-DB EPG pair are VRF:101, EPG:App, EPG:DB, Contract:App-DB, Filter:80/allow, and Filter:700/allow.

A key aspect of shared risks is that they can create different degree of damages to EPG pairs. If an incorrect VRF ID is distributed from the controller to switch agents, all pairs of EPGs belonging to the VRF would be unable to communicate. In contrast, if one filter is incorrectly deployed in one switch, the impact would be limited to the endpoints in the EPG pairs that are directly connected to the switch (and to other endpoints that might attempt to talk to those endpoints).

In a network policy, a large number of EPG pairs may depend on a shared risk (object) and/or a single EPG pair may rely on multiple shared risk objects. These not only signify the criticality of a shared risk but also the vulnerability of EPG pairs. More importantly, a dense correlation between shared risks and EPG pairs makes it

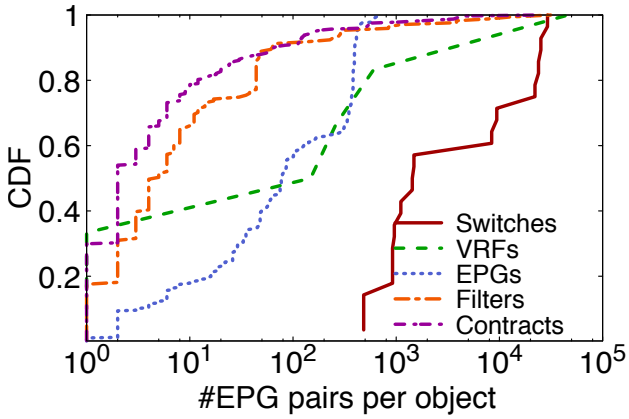


Figure 7.3: Number of EPG pairs per object.

promising to apply risk modeling techniques to fault localization of network policy deployment.

To understand the degree of sharing between EPG pairs and policy objects, we analyze policy configurations from a real production cluster that comprises about 30 switches and hundreds of servers. Figure 7.3 shows the cumulative distribution function on the number of EPG pairs sharing a policy object, from which we make the following observations:

7.3.1 A case study in a production cluster

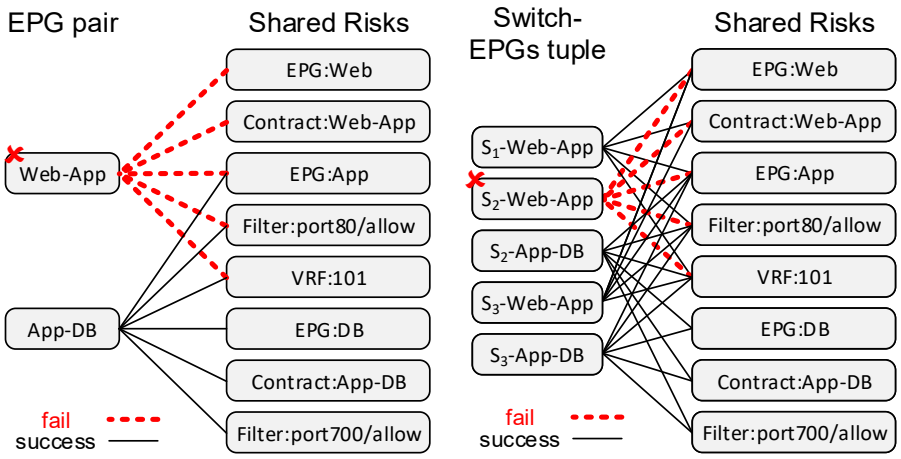
- *A failure in deploying VRF would lead to a breakdown of a number of EPG pairs.* A majority of VRF objects has more than 100 EPG pairs. 10% VRFs are shared by over 1,000 EPG pairs and 2-3% VRFs by over 10,000 EPG pairs.

- *EPGs are configured to talk to many EPGs.* About 50% of EPGs belong to more than 100 EPG pairs, which implies that the failure of an EPG is communication outage with a significant number of EPGs.

- *The failure of a physical object such as switch would create the biggest impact on EPG pairs.* About 80% of switches maintain at least 1,000s of EPG pairs.

- *Contract and filter are mostly shared by a small number of EPG pairs.* 70% of the filters and 80% of the contracts are used by less than 10 EPG pairs.

Overall, it is evident that failures in a shared risk affect a great number of EPG pairs. Thus, spatial correlation holds promise in localizing problematic shared risks among a huge number of shared risks in large-scale networks.



(a) Switch risk model for switch S_2 (b) Controller risk model

Figure 7.4: Risk models for switch S_2. When the 1st rule is missing from the TCAM in S_2 in Figure 7.2, the edges associated with the Web-App EPG pair are marked as fail (details on §7.3.3).

7.3.2 Risk models

We adopt a bipartite graph model that has been actively used to model risks in the traditional IP network [62]. A bipartite graph demonstrates associations between policy objects and the elements that would be affected by those objects. At one side of the graph are policy objects (*e.g.*, VRF, EPG, filter, etc.); and the affected elements (*e.g.*, EPG pairs) are located at the other side. An edge between a pair of nodes in the two parties is created if an affected element relies on a policy object under consideration.

In modeling risks for network policy, one design question is how to represent risks in the 3-tier deployment hierarchy that involves controller, switch agent and TCAM. During rule deployment, there are two major places that eventually cause the failure of TCAM rule update—one from controller to switch agent and the other from switch agent to TCAM. The former may cause global faults whereas the latter does local faults. For instance, if the controller cannot reach out to a large number of switches for some reason, the policy objects across those unreachable switches are not updated. On the other hand, when one switch is unreachable, a switch agent misbehaves or TCAM has hardware glitches, the scope of risk model should be restricted to a particular switch level. Thus, in order to capture global- and local-level risks properly, we propose two risk models: (i) switch and (ii) controller risk model.

Switch risk model. A switch risk model consists of shared risks (*i.e.*, policy objects) and the elements (*i.e.*, EPG pair) that can be impacted by the shared risks on a per-

switch basis. The model is built from a network policy and the physical locations of endpoints belonging to EPGs in the network. Figure 7.4(a) shows an example of switch risk model for switch S_2 given the local view on network policy in Figure 7.1(c). The left-hand side in the model shows all EPG pairs deployed in switch S_2. Each EPG pair has an edge to those policy objects (on the right-hand side in the model) that it relies on in order to allow traffic between endpoints in the EPG pair. For instance, the Web-App EPG pair has outgoing edges to EPG:Web, EPG:App, VRF:101, Filter:port80/Allow, and Contract:Web-App. An edge is flagged as either success or fail, soon discussed in §7.3.3.

Controller risk model. A controller risk model captures shared risks and their relationships with vulnerable elements across all switches in the network. A controller risk model is constructed in a similar manner of a switch risk model. In the controller risk model, a switch ID and an EPG pair form a triplet (on the left-hand side in Figure 7.4(b)). A triplet has edges to policy objects that the EPG pair relies on in that specific switch. Since the same policy object can be present in more than one switch, an EPG pair in multiple switches can have an edge to the object.

7.3.3 Augmenting risk models

In a conventional risk model, when an element affected by shared risks experiences a failure, it is referred to as an *observation*. In case of switch risk model, an EPG pair is an observation when endpoints in the EPG pair are allowed to communicate but fail to do so.

In our work, an observation is made by collecting the TCAM rules (T-type rules) deployed across all switches periodically and/or in an event-driven fashion, and by conducting an equivalence check between logical TCAM rules (L-type rules) converted from the network policy at the controller and the collected T-type rules. For this, we use an in-house equivalence checker. The equivalence check is to compare two reduced ordered binary decision diagrams (ROBDDs); one from L-type rules, and the other from T-type rules. If both ROBDDs are equivalent, there is no inconsistency between the desired state (*i.e.*, the network policy) and actual state (*i.e.*, the collected TCAM rules). If not, the tool generates a set of *missing* TCAM rules that explains the difference and that should have been deployed in the TCAM but absent from the TCAM). Those missing rules allow to annotate edges in the risk models as failure, thereby providing more details on potentially problematic shared risks. Note that simply reinstalling those missing rules is a stopgap, not a fundamental solution to address

the real problem that creates state inconsistency.

Potentially, the L-T equivalence checker can produce a large number of missing rules. As demonstrated by our study on dependencies between objects (§7.3.1 and Figure 7.3), one ill-presented object at controller and/or switch agent can cause policy violations for over thousands of EPG pairs and make thousands of rules missing from the network. Unfortunately, it is expensive to do object-by-object checking present in the observed violations. Thus, we treat all objects in the observed violations as a potential culprit. We then mark (augment) the edges between the *malfunctioning* EPG pair (due to the missing rule) and its associated objects in the violation as fail.

Figure 7.4(a) illustrates how the switch risk model is augmented with suspect objects if the 1st rule is missing from the TCAM in S_2 in Figure 7.2. To pinpoint culprit object(s), one practical technique is to pick object(s) that explains the observation best (*i.e.*, the famous Occam's Razor principle); in this example, EPG:Web and Contract:Web-App would explain the problem best as they are solely used by the Web-App EPG pair. The lack of the augmented data would make it hard to localize fault policy objects as it suggests that all objects appear equally plausible. Note that the example is deliberately made simple to ease discussion. In reality many edges between EPG pairs and shared risks can be marked as fail (again, see the high degree of dependencies between objects from Figure 7.3).

7.4 Fault Localization

We now build a fault localization algorithm that exploits the risk models discussed in §7.3. We first present a general idea, explain why the existing approach falls short in handling the problem at hand and lastly describe our proposed algorithm.

7.4.1 General idea

In the switch risk model, for instance, an EPG pair is marked as fail, if it has at least one failed edge between the pair and a policy object (see Figure 7.4(a)). Otherwise, the EPG pair is success. Each EPG pair node marked as fail is an *observation*. A set of observations is called a *failure signature*. Any policy object shared across multiple EPG pairs becomes a shared risk.

If all edges to an object are marked as fail, it is highly likely that the failure of deploying that object explains the observations present in the failure signature, and such an object is added to a set called *hypothesis*. Recall in Figure 7.4(a) that the EPG:Web and Contract:Web-App objects best explain the problem of Web-App EPG

pair. On the other hand, other objects such as VRF:101 and EPG:App are less likely to be the culprit because they are also shared by App-DB EPG pair which has no problem. An ideal algorithm should be able to pick all the responsible policy objects as a hypothesis.

In many cases, localizing problematic objects is not as simple as shown in Figure 7.4(a). Multiple object failures can take place simultaneously. In such a case, it is prohibitive to explore all combinations of multiple objects that are likely to explain all of the observations in a failure signature. Therefore, the key objective is to identify a minimal hypothesis (in other words, a minimum number of failed objects) that explains most of the observations in the failure signature. An obvious algorithmic approach would be finding a minimal set of policy objects that covers risk models presented as a bipartite graph. This general set cover problem is known to be NP-complete [52].

7.4.2 Existing algorithm: SCORE

We first take into account a greedy approximation algorithm used by SCORE [62] system that attempts to solve the min set coverage problem and that offers $O(\log n)$-approximation to the optimal solution [62], where n is the number of affected elements (*e.g.*, EPG pairs in our problem). We first explain the SCORE algorithm and further discuss its limitation.

Algorithm. The greedy algorithm in the SCORE system picks policy objects to maximize two utility values—*(i) hit ratio* and *(ii) coverage ratio*—computed for each shared risk. We first introduce a few concepts in order to define them precisely under our switch risk model. The same logic can be applied to the controller risk model.

Let G_i be a set of EPG pairs that depend on a shared risk i, O_i be a subset of G_i in which EPG pairs are marked as fail (*observations*) due to failed edges between the EPG pairs and the shared risk i, and F be the failure signature, a set of all observations, *i.e.*, $F = \bigcup O_i$ for all i. For shared risk i, a hit ratio, h_i is then defined as:

$$h_i = |G_i \cap O_i| / |G_i| = |O_i| / |G_i|$$

In other words, a hit ratio is a fraction of EPG pairs that are observations out of all EPG pairs that depend on a shared risk. A hit ratio is 1 when all EPG pairs that depend on a shared risk are marked as fail. And a coverage ratio, c_i is defined as:

$$c_i = |G_i \cap O_i| / |F| = |O_i| / |F|$$

A coverage ratio denotes a fraction of failed EPG pairs associated with a shared risk from the failure signature.

The algorithm chooses shared risks whose hit ratio is above some fixed threshold value. Next, given the set of selected shared risks, the algorithm outputs those shared risks that have the highest coverage ratio values and that maximize the number of explained observations.

Limitation. The algorithm treats a shared risk with a small hit ratio as noise and simply ignores it. However, in our network policy fault localization problem, we observe that while some policy objects such as filter have a small hit ratio (≈ 0.01), they are indeed responsible for the outage of some EPG pairs. The algorithm excludes those objects, which results in a huge accuracy loss (results in §7.6.2).

It turns out that not all EPG pairs that depend on the object are present in the failure signature. For instance, suppose that 100 EPG pairs depend on a filter, which needs 100 TCAM rules. In this case, if one TCAM rule is missing, a hit ratio of the filter is 0.01. This can happen if installing rules for those EPG pairs is conducted with a time gap. For instance, 99 EPG pairs are configured first, and the 100th EPG pair is a newly-added service, hence configured later.

To make it worse, in reality the hit ratio can vary significantly too. In the previous example, if 95 TCAM rules are impacted, the hit ratio is 0.95. The wide variation of hit ratio values can occur due to (1) switch TCAM overflow; (2) TCAM corruption [112] that causes bit errors on a specific field in a TCAM rule or across TCAM rules; and (3) software bugs [105] that modify object's value wrong at controller or switch agent. While the SCORE algorithm allows change of a threshold value to handle noisy input data, such a static mechanism helps little in solving the problem at hand, confirmed by our evaluation results in §7.6.

7.4.3 Proposed algoirthm: Scout

We propose Scout algorithm that actively takes into account policy objects whose hit ratio percentage is less than 100% and thus overcomes the limitation of the SCORE algorithm. Basically, our algorithm also greedily picks the faulty objects and outputs hypothesis that has a minimal set of objects most likely explains all the observations in a failure signature.

Algorithm 1 shows the core part of our fault localization algorithm. The algorithm takes failure signature F and risk model R as input. F has a set of observations, *e.g.*, EPG pairs marked as fail in the switch risk model. For each observation in F, the algorithm obtains a list of policy objects with fail edges to that observation and computes the utility values (*i.e.*, hit and coverage ratios) for all those objects (lines 6-10). Then,

Algorithm 1 Scout (F, R, C)

```
1: ▷ F: failure signature, R: risk model, C: change logs
2: ▷ P: unexplained set, Q: explained set, H: hypothesis
3: P ← F; Q ← ∅; H ← ∅
4: while P ≠ ∅ do
5:     K ← ∅                                   ▷ K: a set of shared risks
6:     for observation o ∈ P do
7:         objs ← getFailedObjects(o,R)
8:         updateHitCovRatio(objs,R)
9:         K ← K ⋃ objs
10:    end for
11:    faultySet ← pickCandidates(K)
12:    if faultySet = ∅ then
13:        break
14:    end if
15:    affected ← GetNodes(faultySet,R)
16:    R ← Prune(affected,R)
17:    P ← P \ affected; Q ← Q ⋃ affected
18:    H ← H ⋃ faultySet
19: end while
20: if P ≠ ∅ then
21:    for observation o ∈ P do
22:        objs ← lookupChangeLog(o,R,C)
23:        H ← H ⋃ objs
24:    end for
25: end if
26: return H
```

based on the utility values of shared risks in the model, the algorithm picks a subset of the shared risks and treats them as faulty (line 11 and Algorithm 2). In Algorithm 2, if the hit ratio of a shared risk is 1, the risk is included in a candidate risk set (lines 3-7); and then from the set, the shared risks that have the highest coverage ratio values are finally chosen; *i.e.*, a set of shared risks that covers a maximum number of unexplained observations (line 8).

If $faultySet$ is not empty, all EPG pairs that have an edge to any shared risk in the $faultySet$ are pruned from the model (lines 15-16), and failed EPG pairs (observa-

Algorithm 2 pickCandidates(riskVector)

1: $hitSet \leftarrow \emptyset$
2: $maxCovSet \leftarrow \emptyset$
3: **for** $risk\ r \in riskVector$ **do**
4: **if** $\texttt{hitRatio}(r) = 1$ **then**
5: $hitSet \leftarrow hitSet \bigcup \{r\}$
6: **end if**
7: **end for**
8: $maxCovSet \leftarrow \texttt{getMaxCovSet}(hitSet)$
9: **return** $maxCovSet$

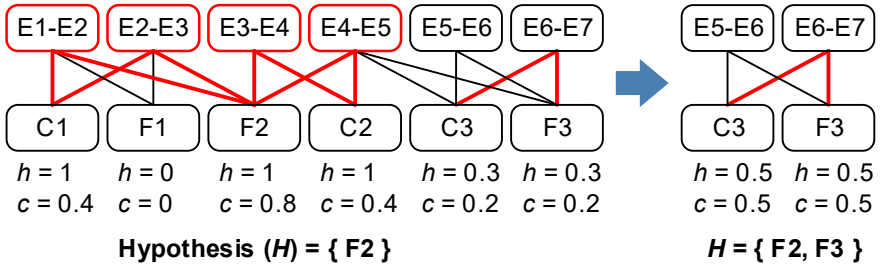


Figure 7.5: An illustration of SCOUT algorithm using a switch risk model. Edges and nodes in red color are fail and those in black are success. Note that h refers to hit ratio and c to coverage ratio.

tions) are moved from unexplained to explained (line 17). Finally, all the shared risks in the $faultySet$ are added to the hypothesis set, H. This process repeats until either there are no more observations left unexplained or when $faultySet$ is empty.

Some observations may remain unexplained because the shared risks associated with those observations have a hit ratio less than 1 and thus are not selected during the above candidate selection procedure. To handle the remaining unexplained observations, the Scout algorithm searches logs about changes made to objects (which are obtained from the controller), and selects the objects to which some actions are recently applied (lines 21-24 in Algorithm 1). Despite its simplicity, this heuristic makes huge improvement in accuracy (§7.6.2).

Example. Figure 7.5 shows an example of how the Scout algorithm works. The lines 4-19 in Algorithm 1 cover the following: (i) filter F2 is identified as a candidate because it has the highest coverage ratio among the shared risks with a hit ratio of 1; (ii) all the EPG pairs that depend on F2 are pruned from the model; (iii) and F2 is added to hypothesis. The lines 21-24 ensure that the algorithm adds filter F3 (assuming

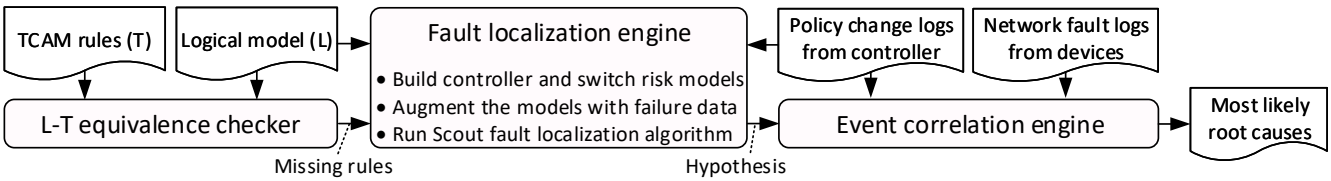


Figure 7.6: Overview of Scout system.

F3 is lately modified) to the hypothesis since there are no shared risks with a hit ratio of 1.

7.5 Scout System

We present Scout system that can conduct an end-to-end analysis from fault localization of policy objects to physical-level root cause diagnosis. The system mainly consists of (i) fault localization engine and (ii) event correlation engine. The former runs the proposed algorithm in §7.4.3 and produces policy objects (*i.e.*, hypothesis) that are likely to be responsible for policy violation of EPG pairs. The latter correlates the hypothesis and two system-level logs from the controller and network devices, and produces the most-likely root causes at physical level that caused object failures. Our prototype is written in about 1,000 lines of Python code. We collect the logical network policy model and its change logs from Cisco's application centric controller, and switch TCAM rules and the device fault logs from Nexus 9000 series switches. Figure 7.6 illustrates the overall architecture of Scout system.

7.5.1 Physical-level root cause diagnosis

Knowing root causes at a physical level such as control channel disruption, TCAM overflow, bugs, system crashes, etc. is as equally important as fixing failed objects in the network policy. In general, when a trouble ticket is raised, the current practice is to narrow down possible root causes by analyzing system logs such as fault logs from network devices. However, a majority in a myriad of log data is often irrelevant to the caused failure. Filtering out such noises can be done to some extent by correlating the logs with the generation time of the trouble ticket, but not effective enough to reduce search space.

The event correlation engine shown in Figure 7.6 is a systematic and automated approach to the above problem. The engine correlates the fault logs from network devices, the change logs from the network policy controller and the hypothesis generated from the fault localization engine. It then infers the most likely physical-level root causes through the correlation.

The engine works in three simple steps: (i) Using the hypothesis, it first identifies a set of change logs that it has to examine; (ii) with the timestamps of those change logs, it then narrows down the relevant faulty logs (those that are logged before the policy changes and keep alive); and (iii) it finally associates impacted policy objects with the fault(s) found in the relevant fault logs and outputs them.

The engine is pre-configured with signatures for known faults (*e.g.*, disconnected switch, TCAM overflow), composed by network admins with their domain knowledge and prior experience. When fault logs match a signature, faults are identified and associated with the impacted policy objects. Otherwise, the objects are tagged with 'unknown'. Note that signatures can be flexibly added to the engine, and the system's ability would be naturally enhanced with more signatures.

7.5.2 Example usecases

We explain three realistic use cases in a testbed and demonstrate the workflow of our system and its efficacy on fault localization. For this purpose, we use the network policy for the 3-tier web service shown in Figure 7.1(a). We first test a TCAM overflow case by continously adding one new filter after another to the Contract:App-DB object. For the other two cases, we make a switch not respond to the controller in the middle of updates, by silently dropping packets to the switch. Note that across all cases, we let the switch generate 1000s of noisy logs in addition to the actual fault log.

TCAM overflow. Due to TCAM overflow, several filters were not deployed at TCAM. The switch under test generated fault logs that indicate TCAM overflow when its TCAM utilization was beyond a certain level. Our system first localized the faulty filter objects with risk models, correlated them with the change logs for 'add filter' instruction, and subsequently the change logs with the fault logs. Our system had the fault signature of TCAM overflow, so it was able to match the fault logs with that signature and tag those failed filters accordingly.

Unresponsive switch. In this use case, the switch under test became unresponsive while the controller was sending the 'add filter' instructions to the switch. The equivalence checker reported that the rules associated with some filters are missing. Then, the Scout algorithm localized those filters as faulty objects. Using filter creation times from the change logs and the fault logs that indicate the switch was inactive (both maintained at the controller), the correlation engine was able to detect that filters were created when the switch was inactive.

Too many missing rules. As a variant of the above scenario, we pushed a policy with a large number of policy objects onto the unresponsive switch. We found out that more than 300K missing rules were reported by the equivalence checker. Without fault localization, it is extremely challenging for network admins to correlate and identify the set of underlying objects that are fundamentally responsible for the problem. Scout narrowed it down and reported the unresponsive switch as the root cause behind these huge number of rule misses.

7.6 Evaluation

We evaluate Scout in terms of (i) suspect set reduction, (ii) accuracy, and (iii) scalability. We mean by suspect set reduction a ratio, γ between the size of hypothesis (a set of objects reported by Scout) and the number of all objects that failed EPG pairs rely on; the smaller the ratio is, the less objects network admins should examine. As for accuracy, we use precision ($|G \cap H|/|H|$) and recall ($|G \cap H|/|G|$) where H is hypothesis and G is a set for ground truth. A higher precision means fewer false positives and a higher recall means fewer false negatives. Finally, we evaluate scalability via measuring running times across different network sizes.

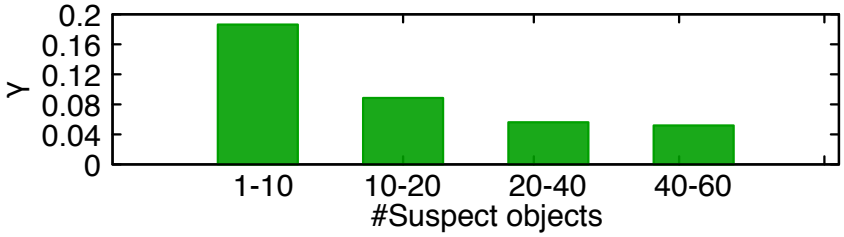
7.6.1 Evaluation environment

Setup. We conduct our evaluation under two settings.

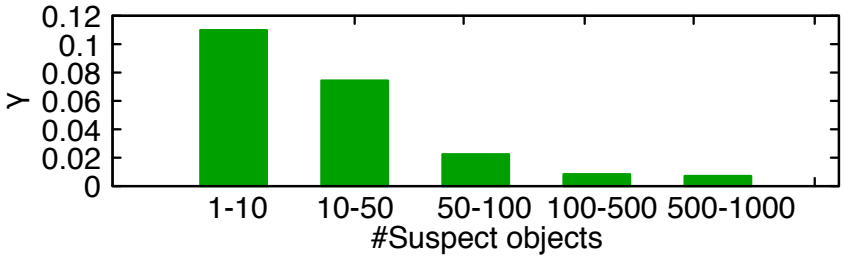
Simulation: We build our simulation setup with network policies used in our production cluster that comprises about 30 switches and 100s of servers. The cluster dataset contains 6 VRFs, 615 EPGs, 386 contracts, and 160 filters.

Testbed: We build a network policy that consists of 36 EPGs, 24 contracts, 9 filters, and 100 EPG pairs, based on the statistics of the number of EPGs and their dependency on other policy objects obtained from the above cluster dataset.

Fault injection. We define two types of faults that cause inconsistency between network policy and switch TCAM rules. *(i) Full object fault* means that all TCAM rules associated with an object are missing. *(ii) Partial object fault* is a fault that makes some of the EPG pairs that depend on an object fail to communicate. That is, some TCAM rules associated with the object are missing. For both simulation and experiment, we randomly generate the two types of faults with equal weight.



(a) Faults in testbed



(b) Simulated faults

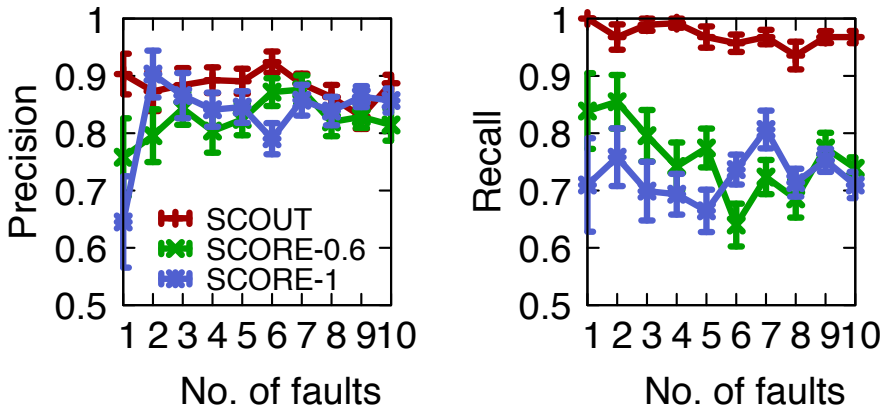
Figure 7.7: Suspect set reduction.

7.6.2 Results

Suspect set reduction. We first compare the size of hypothesis with the number of policy objects (a suspect set) that EPG pairs in failure depend on. We use the metric γ defined earlier for this comparison. Figure 7.7 shows the suspect set reduction ratio in the simulation and testbed. We generate 1,500 faults of object in the simulation and 200 faults of object in the testbed; for each object fault, we compute the total number of objects, that the EPG pairs impacted by the faulty object depend on. From the figure, we see γ is less than 0.08 in most cases. Scout reports at maximum 10 policy objects in the hypothesis whereas without fault localization network admin should suspect as many as a thousand policy objects. This smaller γ value means that network admins need to examine a relatively small number of objects to fix inconsistencies between a network policy and deployed TCAM rules. Therefore, Scout can greatly help reduce repair time and necessary human resources.

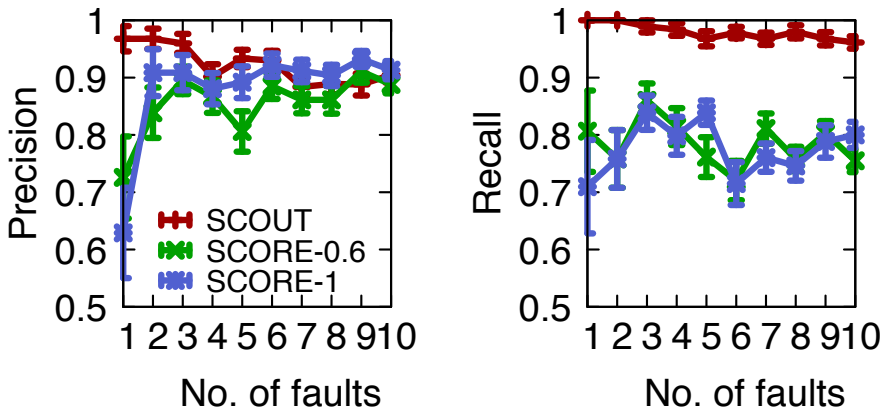
Accuracy. While it is great that Scout produces a handful of objects that require investigation, a more important aspect is that the hypothesis should contain more number of truly faulty objects and less number of non-faulty objects. We study this using precision and recall. In addition, we compare Scout's accuracy with SCORE's. We use two different error threshold values for SCORE to see if changing parameters would help improve its accuracy.

Figures 7.8(a) and 7.8(b) show recall and precision of fault localization with multiple number of simultaneous faulty objects (x-axis) in the switch risk model. From the



(a) Precision (b) Recall

Figure 7.8: Fault localization performance on switch risk model. X in SCORE-X is an error threshold set for hit ratio. The results are averaged over 30 runs.

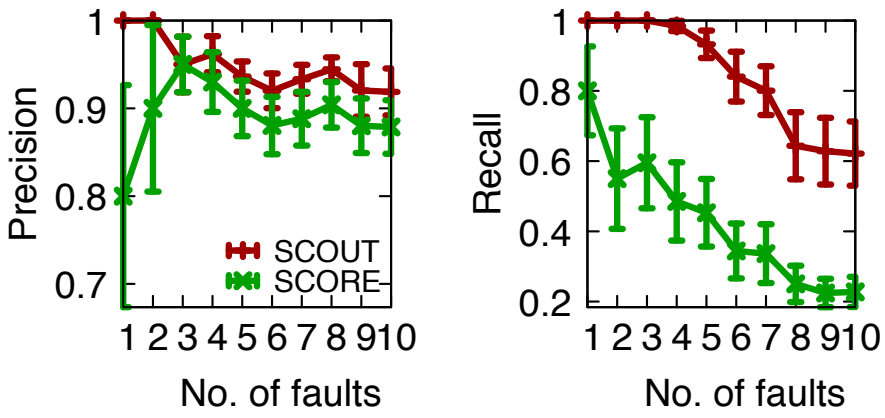


(a) Precision (b) Recall

Figure 7.9: Fault localization performance on controller risk model with faulty policy objects across switches. X in SCORE-X is an error threshold set for hit ratio. Each data point is an average over 30 runs.

figures, we observe Scout's recall is 20-30% better than SCORE's without any compromise on precision. The error threshold values make little change in the performance of SCORE. Also, the high recall of Scout suggests that Scout can always find most faulty objects. Moreover, high precision (close to 0.9) suggests fewer false positives. For instance, with 10 faulty objects in the network policy, Scout reports on average one additional object as faulty. In Figures 7.9(a) and 7.9(b) we observe similar trends for the controller risk model.

Figures 7.10(a) and 7.10(b) compare the accuracy of Scout and SCORE with up to 10 simultaneous faults in the testbed. SCORE's error threshold is set to 1. From the figures, we observe Scout's recall is much better (20-50%) than SCORE's while its precision is comparable to SCORE's. Scout detects all faulty objects when there



(a) Precision (b) Recall

Figure 7.10: Fault localization performance when policy objects fail to be deployed in a switch. Each data point is an average over 10 runs.

are less than four faults, *i.e.*, with 100% recall and about 98% precision. When there are five or more faults, Scout's accuracy (especially, recall) begins to decrease. The difference in accuracy between the simulation and testbed setup is mainly because of a low degree of risk sharing among EPG pairs in the testbed, when compared to the simulation dataset obtained from the production cluster.

Scalability. We measure running time of Scout under a controller risk model from the network policy deployed in 10 switches in the production cluster. We scale the model up to 500 switches by adding new EPG and switch pairs. We observe that Scout takes about 45 and 130 seconds with 200 and 500 switches respectively, on a machine with a 4-core 2.6 GHz CPU and 16GB memory.

7.7 Limitations

Routing policy deployment failures. In general, there are mainly two types of network policies: security and routing policies. Low-level rules like access control list, are installed into edge switches based on security policies, and enforce which traffic to drop, forward, or modify. On the other hand, routing policies enforce which path the traffic should follow between an ingress and an egress edge switch. In this chapter, we present Scout, an end-to-end system that localizes faults while deploying security policies. Scout can be extended to provide support for routing policy deployment failures as well. This is important because, an edge switch processes traffic as defined by both routing and security policy. Perhaps, similar to Scout, we can introduce models that capture routing policy deployment failures, run a fault localization algorithm, and

localize the minimal set of policy objects that explains most of the routing failures.

Maintaining signatures of known faults. The event correlation engine infers the most likely physical-level failures, like a disconnected switch, TCAM overflow, etc., which might be the root cause for policy deployment failures. In this work, we pre-configured the engine with signatures of all known physical-level failures. When a spurious network event is detected, network fault logs are matched against the signatures, and infer the root cause. However, in practice, updating (add, delete, or modify) a signature list requires domain knowledge and prior experience. For now, we manually update the signatures, instead of automatically creating them.

7.8 Summary

Network policy abstraction enables flexible and intuitive policy management. However it also makes network troubleshooting prohibitively hard when network policies are not deployed as expected. In this chapter we introduced and solved a network policy fault localization problem where the goal is to identify faulty policy objects that have low-level rules go missing from network devices and thus are responsible for network outages and policy violations. We formulated the problem with risks models and proposed a greedy algorithm that accurately pinpoints faulty policy objects and built Scout, an end-to-end system that automatically pinpoints not only faulty policy objects but also physical-level failures.

Chapter 8

Conclusion

To conclude, this chapter outlines the areas of future works and summarizes thesis contributions.

8.1 Future work

Per-packet logs support. PathDump and SwitchPointer make a case for shifting debugging functionality from networks to end-hosts. Because storing individual packet header has a high latency and throughput bottleneck at high line rates (*e.g.*, 10Gig, 40Gig), end-host agent aggregate all packets within a flow and store records on a per-flow (5-tuple) basis. However, some debugging applications that run SQL-like queries on various packet header fields require per-packet logs, therefore may not fully benefit with flow-level statistics. For example, queries that execute on custom packet header fields (defined to enable specific functionality), like VXLAN header fields in CONGA [19], TPP fields in [51], etc.

Can we provide support to capture, store, and execute queries on per-packet logs while respecting end-host resources? Some possibilities are integrating advanced data structures advocated in trumpet [71], use hardware-based packet capture [47], or filter packet headers that represents a spurious network event in the hardware. Such a per-packet log capability at end-hosts would enable debugging a class of network problems in addition to those currently supported by PathDump and SwitchPointer.

Universal packet trajectory tracing. In a large-scale data center, there exist hundreds of paths between a pair of end-hosts. While debugging network problems, knowing the packet path greatly reduces debugging space [44, 112, 49, 22, 88]. Some of the techniques to trace packet trajectory are: enforcing path-let routing [110], embedding

each linkID that a packet traverse into the packet header [51], collect per-packet per-switch logs [47], and link sampling technique like CherryPick. In this thesis, based on a link sampling technique, we provide OpenFlow rules for tracing a packet path in two popular topologies, fat-tree and VL2. We also showed this technique requires minimal switch rules and small packet header space (one VLAN tag for 4-hops), so that CherryPick works in a network with commodity SDN switches. However, due to packet header space and switch hardware limitation, CherryPick may not support other topologies like DCell, BCube, HyperX, etc.

We envision this limitations would go away with the emergence of programmable hardware. So, a promising future work would be to come up with a system for tracing packet trajectory irrespective to underlying network topology. The system should exploit topology characteristics, and automatically generate switch flow rules, such that, it uses optimal switch resources and packet header space. One possibility is to leverage switch hardware programmability, and define a packet parser, match-action rules, and packet header bits that work for a specific topology. Such a system abstracts the underlying network topology's details, and ease operators while debugging network problems.

Policy for maximum resource utilization. Typically, multiple tenants share the network infrastructure. Tenants could be application owners in a cloud, business units in an enterprise network, or departments in a campus network. In addition to the usual operations, a SDN controller converts a tenant policy to low-level rules, and also guarantees performance isolation. For example, access control rules derived from a policy are deployed in edge switches, as if the customer is the only one using the switch resources. From the cloud provider's perspective, in addition to optimizing resource allocation for a single tenant, the provider should also satisfy other tenants such that resource utilization is maximized.

Can we dynamically recommend a network policy that best meets both tenant needs (security, isolation) and cloud provider requirements (maximum resource utilization)? Perhaps, one approach is to build a global model similar to bipartite models in Scout, that captures tenant policies and their resources utilization, run optimization algorithms on the model, and recommend a win-win policy to the tenants.

SwitchPointer on-chip support. We envision that a hardware prototype that implements SwitchPointer pointer would be the good next step to realize our idea — distributed switch storage as distributed directory service. It would eliminate the perfor-

mance limitations of the current software versions we have built. One approach is to study the available constructs (hashing, per-packet memory update) in programmable hardware such as P4 and NetFPGA, if necessary also implement new constructs.

8.2 Contributions

Monitoring and debugging data plane problems in large-scale networks is complex. Existing solutions operate at one of the two extremes — systems running at end-hosts (more resources but less visibility into the network) or at network switches (more visibility, but limited resources). This thesis calls for a different approach for network debugging: it carefully partitions the monitoring and debugging functionality between network elements and end-hosts. Towards this direction, this thesis presents CherryPick, PathDump, and SwitchPointer, together integrates in-network visibility and resources and programmability of end-hosts. In specific, we showed that an end-host based network debugger, PathDump gain in-network visibility with SwitchPointer (with its pointers to end-hosts at switches) and CherryPick (with its packet trajectory tracing technique), and allows to debug a large class of network problems which includes those that are hard or even infeasible to debug with existing systems.

While CherryPick, PathDump, and SwitchPointer focus on debugging problems in the network data plane, Scout deals with network policy deployment failures. Scout is an end-to-end system that first localizes faulty policy objects, then conduct analysis to identify physical level failure; a possible root cause for objects become faulty. The impact of Scout remains to be seen. Scout work is tested and evaluated on Cisco's SDN solution for data center networks — Application Centric Infrastructure (ACI) — integration of Scout into Cisco's in-house debugging tool is part of the future plan.

8.3 Towards automated network debugging

A fully automated debugging tool — a key component in self-driving networks — should detect, locate, and find root cause of network problems. In specific, it should tell about *where* and *what* is the problem, instead of using human skills and expertise which might take many man-hours. This thesis does not target "automated" debugging but rather builds a framework to simplify the debugging process. We hope this thesis would guide network admins while designing debugging tools for self-driving networks.

Bibliography

[1] Amazon.com suffers outage: Nearly $5m down the drain? http://tinyurl.com/od7vhm8.

[2] CMPH - C Minimal Perfect Hashing Library. http://cmph.sourceforge.net/.

[3] DPDK: Data Plane Development Kit. http://dpdk.org/.

[4] Flask. http://flask.pocoo.org/.

[5] Hyper-V virtual switch. https://tinyurl.com/y8orbkkp.

[6] In-band Network Telemetry. https://github.com/p4lang/p4factory/tree/master/apps/int.

[7] MongoDB. https://www.mongodb.org/.

[8] Open vSwitch. http://openvswitch.org/.

[9] OpenDaylight Group Policy. https://wiki.opendaylight.org/view/Group_Policy:Main.

[10] Programmable hardware switches. https://www.sdxcentral.com/articles/news/att-picks-barefoot-networks-programmable-switches/2017/04/.

[11] sFlow. http://www.sflow.org/.

[12] Solving the mystery of link imbalance. http://tinyurl.com/m9vv4zj.

[13] Sampled NetFlow. http://www.cisco.com/c/en/us/td/docs/ios/12_0s/feature/guide/12s_sanf.html, 2003.

[14] PathDump. https://github.com/PathDump, 2016.

[15] Cisco Application Centric Infrastructure. https://goo.gl/WQYqnv, 2017.

[16] AGARWAL, K., ROZNER, E., DIXON, C., AND CARTER, J. SDN Traceroute: Tracing SDN Forwarding Without Changing Network Behavior. In *ACM HotSDN* (2014).

[17] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A Scalable, Commodity Data Center Network Architecture. In *ACM SIGCOMM* (2008).

[18] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. Hedera: Dynamic flow scheduling for data center networks. In *USENIX NSDI* (2010).

[19] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., VAIDYANATHAN, R., CHU, K., FINGERHUT, A., LAM, V. T., MATUS, F., PAN, R., YADAV, N., AND VARGHESE, G. Conga: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM* (2014).

[20] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center TCP (DCTCP). In *ACM SIGCOMM* (2010).

[21] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pFabric: minimal near-optimal datacenter transport. In *ACM SIGCOMM* (2013).

[22] ARZANI, B., CIRACI, S., CHAMON, L., ZHU, Y., LIU, H., PADHYE, J., OUTHRED, G., AND LOO, B. T. Closing the network diagnostics gap with vigil. In *ACM SIGCOMM Posters and Demos* (2017).

[23] BAHL, P., CHANDRA, R., GREENBERG, A., KANDULA, S., MALTZ, D. A., AND ZHANG, M. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM* (2007).

[24] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. Understanding Data Center Traffic Characteristics. *ACM SIGCOMM CCR 40*, 1 (Jan. 2010).

[25] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *ACM SIGCOMM* (2013).

[26] CASADO, M., FREEDMAN, M. J., PETTIT, J., LUO, J., MCKEOWN, N., AND SHENKER, S. Ethane: Taking control of the enterprise. ACM SIGCOMM.

[27] CHEN, A., WU, Y., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance. In *ACM SIGCOMM* (2016).

[28] CHEN, H., FOSTER, N., SILVERMAN, J., WHITTAKER, M., ZHANG, B., AND ZHANG, R. Felix: Implementing traffic measurement on end hosts using program analysis. In *ACM SIGCOMM SOSR* (2016).

[29] CHEN, Y., GRIFFITH, R., LIU, J., KATZ, R. H., AND JOSEPH, A. D. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *ACM Workshop on Research on Enterprise Networking* (2009).

[30] CHIESA, M., NIKOLAEVSKIY, I., PANDA, A., GURTOV, A., SCHAPIRA, M., AND SHENKER, S. Exploring the Limits of Static Failover Routing. *CoRR abs/1409.0034* (2014).

[31] COLE, R., OST, K., AND SCHIRRA, S. Edge-Coloring Bipartite Multigraphs in O(E log D) Time. *Combinatorica 21*, 1 (2001).

[32] CURTIS, A. R., KIM, W., AND YALAGANDULA, P. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *IEEE INFOCOM* (2011).

[33] CURTIS, A. R., MOGUL, J. C., TOURRILHES, J., YALAGANDULA, P., SHARMA, P., AND BANERJEE, S. Devoflow: scaling flow management for high-performance networks. In *ACM SIGCOMM* (2011).

[34] DHAMDHERE, A., TEIXEIRA, R., DOVROLIS, C., AND DIOT, C. NetDiagnoser: Troubleshooting Network Unreachabilities Using End-to-end Probes and Routing Data. In *ACM CoNEXT* (2007).

[35] DIXIT, A., PRAKASH, P., HU, Y. C., AND KOMPELLA, R. R. On the Impact of Packet Spraying in Data Center Networks. In *IEEE INFOCOM* (2013).

[36] DUFFIELD, N. G., AND GROSSGLAUSER, M. Trajectory Sampling for Direct Traffic Observation. *IEEE/ACM ToN 9*, 3 (2001).

[37] ESTAN, C., KEYS, K., MOORE, D., AND VARGHESE, G. Building a better netflow. In *ACM SIGCOMM* (2004).

[38] ESTAN, C., AND VARGHESE, G. New directions in traffic measurement and accounting. In *ACM SIGCOMM* (2002).

[39] FERGUSON, A. D., ET AL. Participatory networking: An api for application control of sdns. In *ACM SIGCOMM* (2013).

[40] FOGEL, A., FUNG, S., PEDROSA, L., WALRAED-SULLIVAN, M., GOVINDAN, R., MAHAJAN, R., AND MILLSTEIN, T. A General Approach to Network Configuration Analysis. In *USENIX NSDI* (2015).

[41] FOSTER, N., HARRISON, R., FREEDMAN, M. J., MONSANTO, C., REXFORD, J., STORY, A., AND WALKER, D. Frenetic: A network programming language. In *ACM SIGPLAN ICFP* (2011).

[42] FOX, E. A., CHEN, Q. F., AND HEATH, L. S. A Faster Algorithm for Constructing Minimal Perfect Hash Functions. In *ACM SIGIR* (1992).

[43] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VL2: A Scalable and Flexible Data Center Network. In *ACM SIGCOMM* (2009).

[44] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., LIN, Z.-W., AND KURIEN, V. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *ACM SIGCOMM* (2015).

[45] GUPTA, A., BIRKNER, R., CANINI, M., FEAMSTER, N., MAC-STOKER, C., AND WILLINGER, W. Network monitoring as a streaming analytics problem. In *ACM HotNets* (2016), HotNets '16.

[46] HANDIGOL, N., HELLER, B., JEYAKUMAR, V., MAZIÉRES, D., AND MCKEOWN, N. Where is the Debugger for My Software-defined Network? In *ACM HotSDN* (2012).

[47] HANDIGOL, N., HELLER, B., JEYAKUMAR, V., MAZIÈRES, D., AND MCKEOWN, N. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *USENIX NSDI* (2014).

[48] HONDA, M., HUICI, F., LETTIERI, G., AND RIZZO, L. mswitch: A highly-scalable, modular software switch. In *ACM SIGCOMM SOSR* (2015).

[49] HUANG, P., GUO, C., ZHOU, L., LORCH, J. R., DANG, Y., CHINTALAPATI, M., AND YAO, R. Gray failure: The achilles' heel of cloud-scale systems. In *ACM HotOS* (2017).

[50] HUANG, Q., JIN, X., LEE, P. P. C., LI, R., TANG, L., CHEN, Y.-C., AND ZHANG, G. Sketchvisor: Robust network measurement for software packet processing. In *ACM SIGCOMM* (2017).

[51] JEYAKUMAR, V., ALIZADEH, M., GENG, Y., KIM, C., AND MAZIÈRES, D. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. In *ACM SIGCOMM* (2014).

[52] JOHNSON, D. S. Approximation algorithms for combinatorial problems. *Journal of computer and system sciences 9*, 3 (1974), 256–278.

[53] KANDULA, S., ET AL. Shrink: A tool for failure diagnosis in ip networks. In *ACM SIGCOMM workshop on Mining network data* (2005).

[54] KANDULA, S., MAHAJAN, R., VERKAIK, P., AGARWAL, S., PADHYE, J., AND BAHL, P. Detailed Diagnosis in Enterprise Networks. In *ACM SIGCOMM* (2009).

[55] KANDULA, S., SENGUPTA, S., GREENBERG, A., PATEL, P., AND CHAIKEN, R. The Nature of Data Center Traffic: Measurements & Analysis. In *ACM IMC* (2009).

[56] KANG, N., ET AL. Optimizing the "one big switch" abstraction in software-defined networks. In *ACM CoNEXT* (2013).

[57] KATTA, N., HIRA, M., KIM, C., SIVARAMAN, A., AND REXFORD, J. Hula: Scalable load balancing using programmable data planes. In *ACM SOSR* (2016).

[58] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real Time Network Policy Checking Using Header Space Analysis. In *USENIX NSDI* (2013).

[59] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header Space Analysis: Static Checking for Networks. In *USENIX NSDI* (2012).

[60] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *USENIX NSDI* (2013).

[61] KOMPELLA, R. R., YATES, J., GREENBERG, A., AND SNOEREN, A. C. Detection and localization of network black holes. In *IEEE INFOCOM* (2007).

[62] KOMPELLA, R. R., YATES, J., GREENBERG, A., AND SNOEREN, A. C. Fault localization via risk modeling. *IEEE Transactions on Dependable and Secure Computing 7*, 4 (2010), 396–409.

[63] LI, Y., MIAO, R., KIM, C., AND YU, M. FlowRadar: A Better NetFlow for Data Centers. In *USENIX NSDI* (2016).

[64] LIU, Z., MANOUSIS, A., VORSANGER, G., SEKAR, V., AND BRAVERMAN, V. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *ACM SIGCOMM* (2016).

[65] LOGOTHETIS, D., TREZZO, C., WEBB, K. C., AND YOCUM, K. In-situ MapReduce for Log Processing. In *USENIX ATC* (2011).

[66] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P. B., AND KING, S. T. Debugging the Data Plane with Anteater. In *ACM SIGCOMM* (2011).

[67] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: Interactive analysis of web-scale datasets. In *VLDB* (2010).

[68] MITTAL, R., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. Universal packet scheduling. In *ACM HotNets* (2015).

[69] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., AND WALKER, D. Composing Software-defined Networks. In *USENIX NSDI* (2013).

[70] MOSHREF, M., YU, M., GOVINDAN, R., AND VAHDAT, A. Dream: dynamic resource allocation for software-defined measurement. In *ACM SIGCOMM* (2014).

[71] MOSHREF, M., YU, M., GOVINDAN, R., AND VAHDAT, A. Trumpet: Timely and Precise Triggers in Data Centers. In *ACM SIGCOMM* (2016).

[72] MYSORE, R. N., ET AL. Gestalt: Fast, Unified Fault Localization for Networked Systems. In *USENIX ATC* (2014).

[73] NARAYANA, S., REXFORD, J., AND WALKER, D. Compiling Path Queries in Software-defined Networks. In *ACM HotSDN* (2014).

[74] NARAYANA, S., SIVARAMAN, A., NATHAN, V., GOYAL, P., ARUN, V., ALIZADEH, M., JEYAKUMAR, V., AND KIM, C. Language-Directed Hardware Design for Network Performance Monitoring. In *ACM SIGCOMM* (2017).

[75] NARAYANA, S., TAHMASBI, M., REXFORD, J., AND WALKER, D. Compiling Path Queries. In *USENIX NSDI* (2016).

[76] NELAKUDITI, S., LEE, S., YU, Y., ZHANG, Z.-L., AND CHUAH, C.-N. Fast local rerouting for handling transient link failures. In *IEEE/ACM ToN* (2007).

[77] NELSON, T., YU, D., LI, Y., FONSECA, R., AND KRISHNAMURTHI, S. Simon: Scriptable Interactive Monitoring for SDNs. In *ACM SIGCOMM SOSR* (2015).

[78] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at facebook. In *USENIX NSDI* (2013).

[79] OPEN NETWORKING FOUNDATION. OpenFlow Switch Specification Version 1.4.0. http://tinyurl.com/kh6ef6s, 2013.

[80] PEARCE, O., GAMBLIN, T., DE SUPINSKI, B. R., SCHULZ, M., AND AMATO, N. M. Quantifying the Effectiveness of Load Balance Algorithms. In *ACM ICS* (2012).

[81] PEARL, J. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., 1988.

[82] PRAKASH, C., ET AL. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *ACM SIGCOMM* (2015).

[83] PRAKASH, P., DIXIT, A., HU, Y. C., AND KOMPELLA, R. The TCP Outcast Problem: Exposing Unfairness in Data Center Networks. In *USENIX NSDI* (2012).

[84] RAICIU, C., BARRE, S., PLUNTKE, C., GREENHALGH, A., WISCHIK, D., AND HANDLEY, M. Improving Datacenter Performance and Robustness with Multipath TCP. In *ACM SIGCOMM* (2011).

[85] RASLEY, J., STEPHENS, B., DIXON, C., ROZNER, E., FELTER, W., AGARWAL, K., CARTER, J., AND FONSECA, R. Planck: Millisecond-scale monitoring and control for commodity networks. In *ACM SIGCOMM* (2014).

[86] RASLEY, J., STEPHENS, B., DIXON, C., ROZNER, E., FELTER, W., AGARWAL, K., CARTER, J., AND FONSECA, R. Planck: Millisecond-scale monitoring and control for commodity networks. In *ACM SIGCOMM* (2014).

[87] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the social network's (datacenter) network. In *ACM SIGCOMM* (2015).

[88] ROY, A., ZENG, H., BAGGA, J., AND SNOEREN, A. C. Passive realtime datacenter fault detection and localization. In *USENIX NSDI* (2017).

[89] SEKAR, V., REITER, M. K., WILLINGER, W., ZHANG, H., KOMPELLA, R. R., AND ANDERSEN, D. G. Csamp: A system for network-wide flow monitoring. In *USENIX NSDI* (2008).

[90] SEKAR, V., REITER, M. K., AND ZHANG, H. Revisiting the case for a minimalist approach for network flow monitoring. In *ACM IMC*.

[91] SINGH, A., ET AL. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *ACM SIGCOMM* (2015).

[92] SOULÉ, R., ET AL. Merlin: A Language for Provisioning Network Resources. In *ACM CoNEXT* (2014).

[93] STEINDER, M., AND SETHI, A. S. Increasing robustness of fault localization through analysis of lost, spurious, and positive symptoms. In *IEEE INFOCOM* (2002).

[94] STEINDER, M., AND SETHI, A. S. Probabilistic Fault Localization in Communication Systems Using Belief Networks. *IEEE/ACM ToN 12*, 5 (2004).

[95] STEINDER, M., AND SETHI, A. S. A survey of fault localization techniques in computer networks. *Science of Computer Programming 53*, 2 (2004), 165 – 194.

[96] SUH, J., KWON, T. T., DIXON, C., FELTER, W., AND CARTER, J. B. OpenSample: A Low-Latency, Sampling-Based Measurement Platform for Commodity SDN. In *IEEE ICDCS* (2014).

[97] SUN, P., YU, M., FREEDMAN, M. J., REXFORD, J., AND WALKER, D. Hone: Joint host-network traffic management in software-defined networks. *JNSM 23*, 2 (Apr. 2015).

[98] TAMMANA, P., AGARWAL, R., AND LEE, M. CherryPick: Tracing Packet Trajectory in Software-defined Datacenter Networks. In *ACM SIGCOMM SOSR* (2015).

[99] TAMMANA, P., AGARWAL, R., AND LEE, M. Simplifying Datacenter Network Debugging with PathDump. In *USENIX OSDI* (2016).

[100] TSO, F. P., HAMILTON, G., WEBER, R., PERKINS, C. S., AND PEZAROS, D. P. Longer is better: exploiting path diversity in data center networks. In *IEEE ICDCS* (2013).

[101] VENKATARAMAN, S., SONG, D. X., GIBBONS, P. B., AND BLUM, A. New streaming algorithms for fast detection of superspreaders.

[102] VOELLMY, A., ET AL. MAPLE: Simplifying SDN Programming Using Algorithmic Policies. In *ACM SIGCOMM* (2013).

[103] WUNDSAM, A., LEVIN, D., SEETHARAMAN, S., AND FELDMANN, A. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *USENIX ATC* (2011).

[104] YANG, B., LIU, J., SHENKER, S., LI, J., AND ZHENG, K. Keep forwarding: Towards k-link failure resilient routing. In *IEEE INFOCOM* (2014).

[105] YIN, Z., CAESAR, M., AND ZHOU, Y. Towards understanding bugs in open source router software. *SIGCOMM CCR 40*, 3 (2010).

[106] YU, M., GREENBERG, A., MALTZ, D., REXFORD, J., YUAN, L., KANDULA, S., AND KIM, C. Profiling Network Performance for Multi-tier Data Center Applications. In *USENIX NSDI* (2011).

[107] YU, M., JOSE, L., AND MIAO, R. Software defined traffic measurement with OpenSketch. In *USENIX NSDI* (2013).

[108] ZARIFIS, K., MIAO, R., CALDER, M., KATZ-BASSETT, E., YU, M., AND PADHYE, J. DIBS: Just-in-time Congestion Mitigation for Data Centers. In *ACM EuroSys* (2014).

[109] ZENG, H., KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Automatic Test Packet Generation. *IEEE/ACM ToN 22*, 2 (2014), 554–566.

[110] ZHANG, H., LUMEZANU, C., RHEE, J., ARORA, N., XU, Q., AND JIANG, G. Enabling Layer 2 Pathlet Tracing Through Context Encoding in Software-defined Networking. In *ACM HotSDN* (2014).

[111] ZHOU, W., SHERR, M., TAO, T., LI, X., LOO, B. T., AND MAO, Y. Efficient Querying and Maintenance of Network Provenance at Internet-scale. In *ACM SIGMOD* (2010).

[112] ZHU, Y., KANG, N., CAO, J., GREENBERG, A., LU, G., MAHAJAN, R., MALTZ, D., YUAN, L., ZHANG, M., ZHAO, B. Y., AND ZHENG, H. Packet-Level Telemetry in Large Datacenter Networks. In *ACM SIGCOMM* (2015).