Simplifying Datacenter Network Debugging with PathDump

Praveen Tammana
University of Edinburgh

Rachit Agarwal
Cornell University

Myungjin Lee
University of Edinburgh

Abstract

Datacenter networks continue to grow complex due to larger scales, higher speeds and higher link utilization. Existing tools to manage and debug these networks are even more complex, requiring *in-network techniques* like collecting per-packet per-switch logs, dynamic switch rule updates, periodically collecting data plane snapshots, packet mirroring, packet sampling, traffic replay, etc.

This paper calls for a radically different approach to network management and debugging: in contrast to implementing the functionality entirely in-network, we should carefully partition the debugging tasks between the edge devices and the network elements. We present the design, implementation and evaluation of PathDump, a minimalistic tool that utilizes resources at edge devices for network debugging. PathDump currently runs over a real network comprising only of commodity hardware, and yet, can support a surprisingly large class of network debugging problems. Evaluation results show that PathDump requires minimal switch and edge resources, while enabling network debugging at fine-grained time scales.

1 Introduction

Datacenter networks are essential to online services including web search, social media, online commerce, etc. Network outages and performance degradation, even if short-lived, can severely impact these services [1, 2, 3, 6]. Unsurprisingly, there has been a tremendous effort in building tools that allow network operators to efficiently manage networks and debug (the inevitable) network problems [17, 22, 24, 26, 30, 36, 39, 41].

As datacenter networks evolve to larger scales, higher speeds and higher link utilization, new classes of network problems emerge. Accordingly, over the years, network debugging tools have incorporated increasingly complex *in-network techniques* — capturing per-packet per-switch logs [17], collecting snapshots of entire data plane state [20, 21, 22, 26], dynamically updating switch rules [30], selective packet mirroring [33, 41], packet sampling [8, 16, 39, 41], active packet probes [9, 40, 41], traffic replay [37], a potpourri [41] — and this list barely scratches the surface of all the sophisticated techniques that have been proposed to be implemented on network switches for debugging purposes.

In this paper, we do not add to this impressive collection of techniques. Instead, we ask whether there are a non-trivial number of network debugging problems that obviate the need of sophisticated in-network techniques. To explore this question, we argue for a radically different approach: in contrast to implementing the debugging functionality *entirely* in-network, we should carefully partition the functionality between the edge devices and the network switches. Thus, our goal is *not* to beat existing tools, but to help them focus on a smaller set of nails (debugging problems) that we need a hammer (debugging techniques) for. The hope is that by focusing on a smaller set of problems, the already complex networks[1] and the tools for managing and debugging these networks can be kept as simple as possible.

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

[1] as eloquently argued in [41]; in fact, our question about simpler network management and debugging tools was initially motivated by the arguments about network complexity in [41].

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 [14, 28] 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 [17, 33, 41].

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 papers [17, 18, 30, 41] (see Table 2 in the appendix). 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 §2.3 and §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 [15]); 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 tags in hardware and processing more than two tags involves switch CPU (§4.5).

[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 [41]) have to resort to sampling to overcome scalability issues and thus, fail to achieve per-packet granularity.

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 [36], 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.

Overall, this paper 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.

2 PathDump Overview

We start with an overview of PathDump interface (§2.1), and PathDump design (§2.2). We then provide several examples on using PathDump interface for debugging network problems (§2.3, §2.4).

2.1 PathDump Interface

PathDump exposes a simple interface for network debugging; see Table 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` ($\langle t_i, t_j \rangle$);

[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 1: PathDump Interface. See §2.1 for definitions and discussion.

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 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.

2.2 PathDump 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) [36]. PathDump uses the link sampling idea from CherryPick [36] 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 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.

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 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 §4.2.

Silent random packet drops. This network problem occurs when some faulty interface at switch drops packets at random without updating the discarded packet counters at respective interfaces. It is a critical network problem [41] and is often very challenging to localize.

PathDump allows a network operator to implement a localization algorithm such as MAX-COVERAGE [23]. 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.

2.4 Reducing debugging space

As discussed in §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 §2.3 and suppose we want to ensure that packets do not traverse a switch s_1 (that

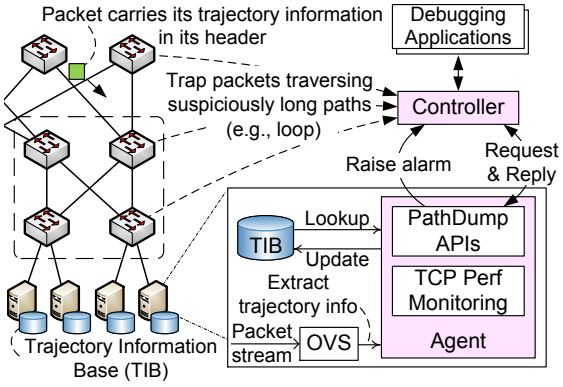


Figure 1: System overview.

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.

3 PathDump Implementation

PathDump implementation comprises of three main components (Figure 1):

- In-network implementation for tracing packet trajectories using packet headers and static network switch rules (§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 (§3.2); we use C/C++ and Python for implementing the stack.
- A controller running network debugging applications in conjunction with the server stack (§3.3). The current controller implementation uses Flask [5] — 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.

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 [36]. 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 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 [34]) and/or longer path lengths without any modification in its design and implementation.

[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.

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).

To keep the paper self-contained, we briefly review the ideas from CherryPick [36] below; we refer the readers to [36] 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: CherryPick [36]. 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 [36]. 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 [13]. 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 [36]). The number of rules at switch grows linearly over switch port density.

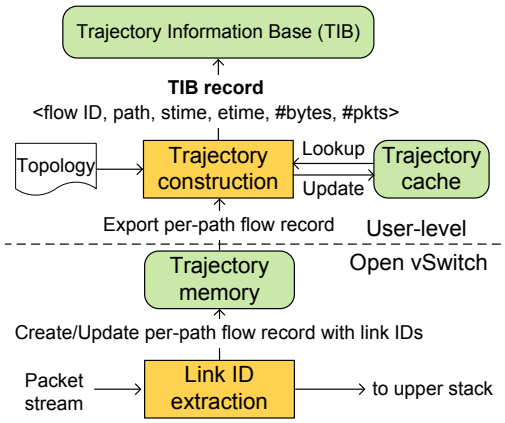


Figure 2: Trajectory information update procedure.

• *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 §4.5).

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 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 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 [4] 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 (§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.

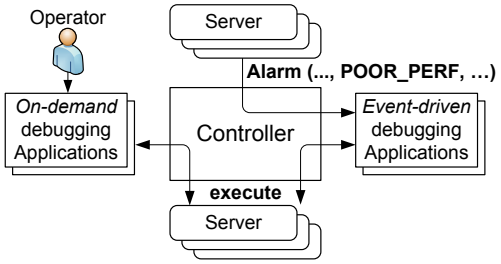


Figure 3: Workflow of PathDump.

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 §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 [27] and iMR [25], 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. The query results are aggregated from the bottom of the tree. However, the current implementation is not fully optimized yet; and improving its efficacy is left as part of our future work.

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 §5, 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.

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.

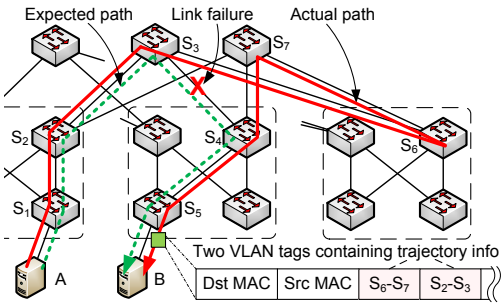


Figure 4: An example of path conformance check.

We thus implement a monitoring module at server that checks TCP connection performance, 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 (§4.3), blackhole diagnosis (§4.4) and TCP performance anomaly diagnosis (§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 §2.3.

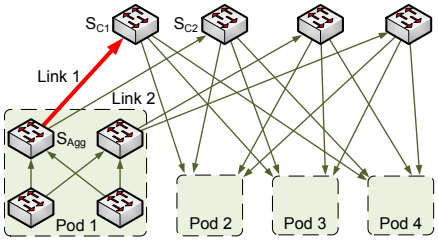
3.3 PathDump controller

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

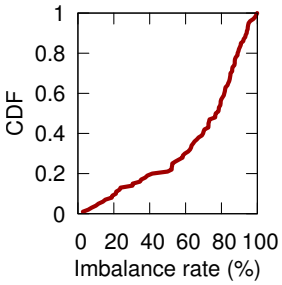
It installs flow rules in switches that append link IDs in the packet header (using `push_vlan` output action) in order to enable packet trajectory tracing. This is one-time task when the controller is initialized, and the rules are not modified once they are installed. We use switches that support a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

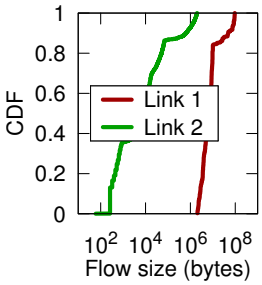
[5] https://github.com/brendangregg/perf-tools



(a) S_{Agg} poorly load-balances traffic



(b) Load imbalance rate



(c) Flow size distribution

Figure 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.

4 Applications

PathDump can support various debugging applications for datacenter network problems including both persistent and transient ones (see Table 2 in the appendix for a comprehensive list of debugging applications). In this section, we highlight a subset of those applications.

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 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 §2.3), that a 6-hop or longer path is a violation of the path conformance policy. The agent detects such packets in real time and alerts the controller to the violation along with the flow key and trajectory.

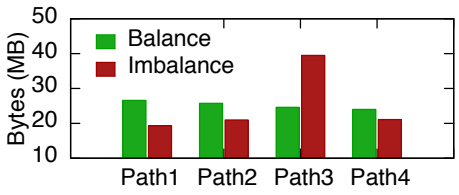


Figure 6: Traffic distribution of a flow along four different paths under balanced and imbalanced cases.

4.2 Load imbalance diagnosis

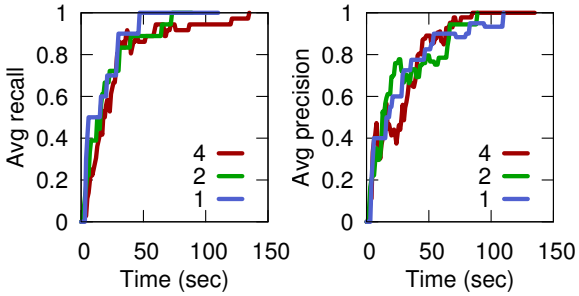
Datacenter networks employ load-balancing mechanisms such as ECMP and packet spraying [15] to exploit numerous equal-cost paths. However, when these mechanisms work poorly, uneven load splits can hurt throughput and flow completion time. PathDump can help narrow down the root causes of load imbalance problems, which we demonstrate using two load-balancing mechanisms: (i) ECMP and (ii) packet spraying.

ECMP load-balancing. This scenario (Figure 5(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 [10], 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 [31].

Figure 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 §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(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.

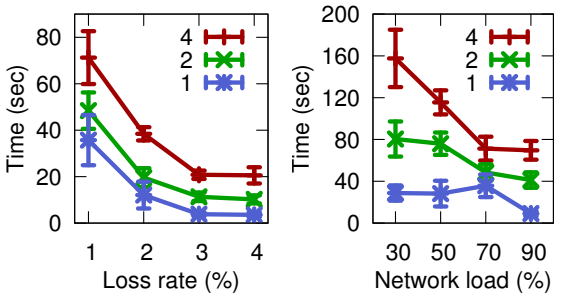
This scenario illustrates how PathDump handles a persistent problem. The application can be easily extended for tackling transient ECMP hash collisions among long flows by exploiting the TCP performance alert function.

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



(a) Recall (b) Precision

Figure 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.



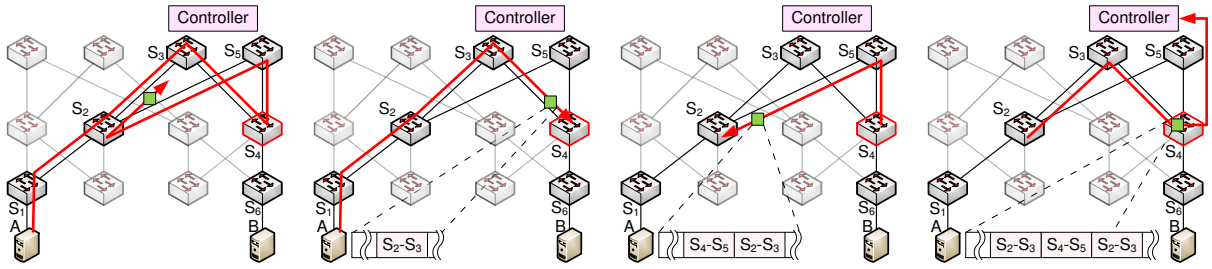
(a) Network load = 70% (b) Loss rate = 1%

Figure 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 ($= 10$).

are deliberately forwarded to one of the paths (*i.e.*, Path 3 in Figure 6). The flow size is set to 100 MB. Figure 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 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.

4.3 Silent random packet drops

We implement the silent packet drop debugging application as described in §2.3 and conduct experiments in a 4-ary fat-tree topology, where each end-host generates traf-



(a) A routing loop case (b) Step 1 (c) Step 2 (d) Step 3

Figure 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.

fic 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 7, we observe the accuracy (both recall and precision) also increases accordingly; the recall increases faster than the precision. It is clear from Figure 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.

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 §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).

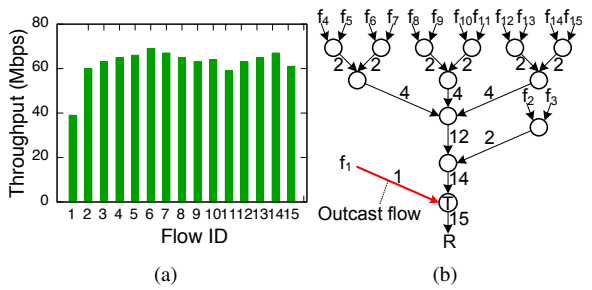
Blackhole at a ToR-aggregate link in the source pod. This blackhole impacts two subflows. The controller identifies two paths that impacted the two subflows using the same way as before. By joining the two paths, the controller can pick four common switches, which should be examined with higher priority.

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

4.5 Routing loop debugging

PathDump debugs routing loop in *real-time* by trapping a suspiciously long path in the network. As discussed in §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 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 9(b)–9(d)) and appears at the controller, two of the tags have the same link ID ($S_2 - S_3$ in Figure 9(d)). Hence, the loop is detected immediately at this stage.



(a) (b)

Figure 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.

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 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.

4.6 TCP performance anomaly diagnosis

PathDump can diagnose incast [12] and outcast [32] 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 §4.4. In addition to that, 15 TCP senders send data to a single receiver for 10 seconds. Thus, as shown in Figure 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 [32] 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 10(a)) and constructs a path tree for all 15 flows (Figure 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 System 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.

5.1 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) [19].

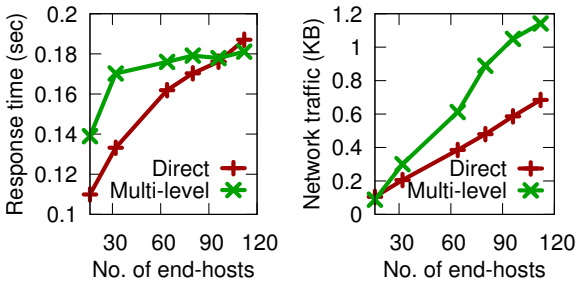
For multi-level query execution, we construct a logical 4-level aggregation tree with 112 end-hosts. Our PathDump controller sits on the top of the tree (level 0). Right beneath the controller are 7 nodes or end-hosts (level 1). Each first-level node has, as its child, four nodes (level 2), each of which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

5.2 Query performance

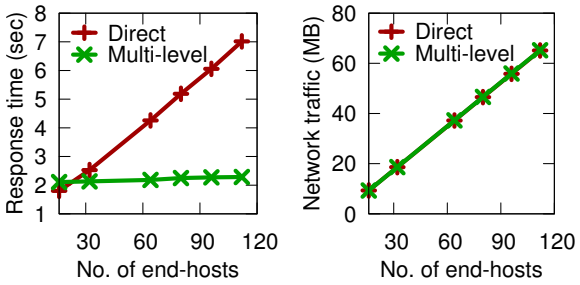
We compare the performance of direct query with that of multi-level query. To understand which type of query suits well to a debugging application, we measure two key metrics: i) end-to-end response time, and ii) total data volume generated. We test two queries—flow size distribution of a link and top-k 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 11 and 12) as follows.



(a) Response time (b) Traffic amount

Figure 11: Average end-to-end response time and traffic amount of a flow size distribution query.

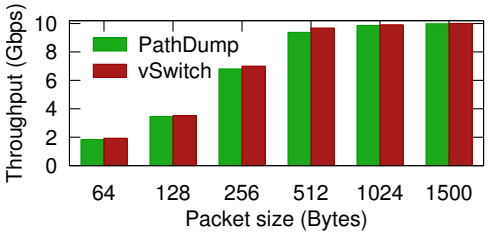


(a) Response time (b) Traffic amount

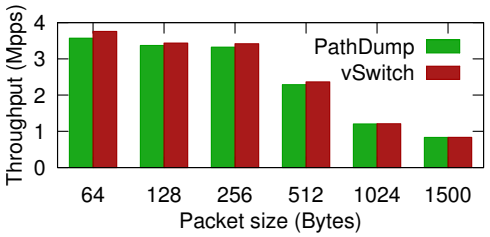
Figure 12: Average end-to-end response time and traffic amount of a top-$10,000$ flows query.

1) When more servers are involved in a query, multi-level query is in general better than direct query. Figure 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 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 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 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.



(a) Throughput in Gbits per second



(b) Throughput in million-packets per second

Figure 13: Forwarding throughput of PathDump and vSwitch. Each bar represents an average over 30 runs.

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 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).

6 Related Work

There has been a tremendous recent effort in building tools for efficient management and debugging of tasks. Each tool works at a unique operating point between supported classes of network debugging problems, accuracy, network bandwidth overheads, and desired functionality from network elements. We summarize the most related of these tools below.

Generality. Several tools support a fairly general class of network debugging problems with high accuracy — PathQuery [30], NetSight [17], NetPlumber [20], VeriFlow [22] and several other systems [21, 26]. However, these systems make arguably strong tradeoffs to achieve generality with accuracy. In particular, for many network debugging problems, these systems [20, 21, 22, 26] require a snapshot of the entire data-plane state and may only be able to capture events at coarse-grained time-scales. Netsight [17] captures per-packet per-switch log for out-of-band analysis; capturing per-packet per-switch logs leads to very high bandwidth requirements and out-of-band analysis typically leads to high latency between the time of occurrence of an event and when the event is diagnosed. Finally, PathQuery [30] supports network debugging by dynamically installing switch rules and using SQL-like queries on these switches; this not only requires dynamic installation of switch rules and large amount of data plane resources to achieve generality but also debugging at coarse-grained time-scales. PathDump, by pushing much of the debugging functionality to the end-hosts, makes a different tradeoff — it gives up on a small class of network debugging problems, but alleviates the overheads of dynamic switch rule installation, per-packet per-switch log generation and periodic data plane snapshots.

Accuracy. Several recent proposals alleviate the overheads of aforementioned systems using sampling [8, 16, 24, 33, 35, 39, 41], mirroring of sampled packets [33, 41], active packet probes [9, 40, 41], and a potpourri of these techniques [41]. These tools have two main limitations: (i) they make the functionality implemented at the network elements (precisely the elements that these tools are trying to debug) even more complex; and (ii) sampling and/or active probing, by definition, leads to missed network events (low accuracy). In contrast, PathDump avoids complex operations like packet sampling, packet mirroring, and/or active probing, by pushing much of the network debugging functionality to the end-hosts. PathDump, thus, performs debugging with high accuracy at finer-grained time-scales without incurring overheads.

End-host based tools. Several recent proposals have advocated to move the functionality to the edge devices [11, 29, 38]. SNAP [38] logs events (e.g., TCP statistics and socket-calls) at end-hosts to infer network problems. Felix [11] proposed a declarative query language for end-host based network measurement. Finally, independent to our work, Trumpet [29] proposes to push the debugging functionality to the end-hosts. PathDump differs from and complements these systems along several dimensions. First, the core idea of PathDump is to exploit the packet trajectories to debug a large class of network problems; capturing and utilizing packet trajectories for debugging purposes complements the techniques used in above tools. Second, in addition to the monitoring functionality of Trumpet [29], PathDump also allows the network operators to slice-and-dice the captured logs to debug a network problem.

7 Conclusion

This paper presents PathDump, a network debugger that 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 (*e.g.*, switch rules and packet header space) and end-host resources (*e.g.*, CPU and memory).

Application	Description	PathDump	PathQuery[30]	Everflow[41]	NetSight[17]	TPP[18]
Loop freedom [17]	Detect forwarding loops	✓	✓	✓	✓	?
Load imbalance diagnosis [41]	Get fine-grained statistics of all flows on set of links	✓	✓	✓	✓	✓
Congested link diagnosis [30]	Find flows using a congested link, to help rerouting	✓	✓	✓	✓	✓
Silent blackhole detection [41, 30]	Find switch that drops all packets silently	✓	✓	✓	✓	✗
Silent packet drop detection [41]	Find switch that drops packets silently and randomly	✓	✓	✓	✓	✗
Packet drops on servers [41]	Localize packet drop sources (network vs. server)	✓	✓	✓	✓	✓
Overlay loop detection [41]	Loop between SLB and physical IP	✗	✓	✓	✓	?
Protocol bugs [41]	Bugs in the implementation of network protocols	✓	✓	✓	✓	?
Isolation [17]	Check if hosts are allowed to talk	✓	✓	✓	✓	✓
Incorrect packet modification [17]	Localize switch that modifies packet incorrectly	✗	✓	?	✓	✗
Waypoint routing [17, 30]	Identify packets not passing through a waypoint	✓	✓	✓	✓	✓
DDoS diagnosis [30]	Get statistics of DDoS attack sources	✓	✓	✓	✓	✓
Traffic matrix [30]	Get traffic volume between all switch pairs in a switch	✓	✓	✓	✓	✓
Netshark [17]	Nework-wide path-aware packet logger	✓	✓	✓	✓	✓
Max path length [17]	No packet should exceed path length of size n	✓	✓	✓	✓	✓

Table 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.

Acknowledgments

We would like to thank anonymous reviewers and our shepherd George Porter for their feedback and suggestions. We would also like to thank Ratul Mahajan for many interesting discussions. This work was in part supported by EPSRC grant EP/L02277X/1.

Appendix

Table 2 summarizes the set of applications discussed in several recent papers, and outlines whether a tool supports an application or not (the table intentionally ignores the resource requirements and/or complexity of supporting each individual application for the respective tools).

References

[1] Amazon EBS failure brings down Reddit, Imgur, others. http://tinyurl.com/oxmugps.

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

[3] Azure outage raises questions about public cloud for mission-critical apps. http://tinyurl.com/no92ojy.

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

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

[6] Google outage: Internet traffic plunges 40%. http://tinyurl.com/l7hegn6.

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

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

[9] K. Agarwal, E. Rozner, C. Dixon, and J. Carter. SDN Traceroute: Tracing SDN Forwarding Without Changing Network Behavior. In *ACM HotSDN*, 2014.

[10] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: minimal near-optimal datacenter transport. In *ACM SIGCOMM*, 2013.

[11] H. Chen, N. Foster, J. Silverman, M. Whittaker, B. Zhang, and R. Zhang. Felix: Implementing traffic measurement on end hosts using program analysis. In *ACM SIGCOMM SOSR*, 2016.

[12] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *ACM Workshop on Research on Enterprise Networking*, 2009.

[13] R. Cole, K. Ost, and S. Schirra. Edge-Coloring Bipartite Multigraphs in O(E log D) Time. *Combinatorica*, 21(1), 2001.

[14] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: scaling flow management for high-performance networks. In *ACM SIGCOMM*, 2011.

[15] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. On the Impact of Packet Spraying in Data Center Networks. In *IEEE INFOCOM*, 2013.

[16] N. G. Duffield and M. Grossglauser. Trajectory Sampling for Direct Traffic Observation. *IEEE/ACM ToN*, 9(3), 2001.

[17] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *USENIX NSDI*, 2014.

[18] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. In *ACM SIGCOMM*, 2014.

[19] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The Nature of Data Center Traffic: Measurements & Analysis. In *ACM IMC*, 2009.

[20] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *USENIX NSDI*, 2013.

[21] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *USENIX NSDI*, 2012.

[22] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *USENIX NSDI*, 2013.

[23] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Detection and localization of network black holes. In *IEEE INFOCOM*, 2007.

[24] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A Better NetFlow for Data Centers. In *USENIX NSDI*, 2016.

[25] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-situ MapReduce for Log Processing. In *USENIX ATC*, 2011.

[26] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. In *ACM SIGCOMM*, 2011.

[27] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *VLDB*, 2010.

[28] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Dream: dynamic resource allocation for software-defined measurement. In *ACM SIGCOMM*, 2014.

[29] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and Precise Triggers in Data Centers. In *ACM SIGCOMM*, 2016.

[30] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker. Compiling Path Queries. In *USENIX NSDI*, 2016.

[31] O. Pearce, T. Gamblin, B. R. de Supinski, M. Schulz, and N. M. Amato. Quantifying the Effectiveness of Load Balance Algorithms. In *ACM ICS*, 2012.

[32] P. Prakash, A. Dixit, Y. C. Hu, and R. Kompella. The TCP Outcast Problem: Exposing Unfairness in Data Center Networks. In *USENIX NSDI*, 2012.

[33] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. In *ACM SIGCOMM*, 2014.

[34] A. Singh et al. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *ACM SIGCOMM*, 2015.

[35] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. B. Carter. OpenSample: A Low-Latency, Sampling-Based Measurement Platform for Commodity SDN. In *IEEE ICDCS*, 2014.

[36] P. Tammana, R. Agarwal, and M. Lee. CherryPick: Tracing Packet Trajectory in Software-defined Datacenter Networks. In *ACM SIGCOMM SOSR*, 2015.

[37] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *USENIX ATC*, 2011.

[38] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling Network Performance for Multi-tier Data Center Applications. In *USENIX NSDI*, 2011.

[39] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with OpenSketch. In *USENIX NSDI*, 2013.

[40] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic Test Packet Generation. *IEEE/ACM ToN*, 22(2):554–566, 2014.

[41] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-Level Telemetry in Large Datacenter Networks. In *ACM SIGCOMM*, 2015.