Distributed Network Monitoring and Debugging with SwitchPointer

Praveen Tammana, ***University of Edinburgh;*** **Rachit Agarwal,** ***Cornell University;***
Myungjin Lee, ***University of Edinburgh***

https://www.usenix.org/conference/nsdi18/presentation/tammana

This paper is included in the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18).

April 9–11, 2018 • Renton, WA, USA

ISBN 978-1-931971-43-0

Open access to the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation is sponsored by USENIX.

Distributed Network Monitoring and Debugging with SwitchPointer

Praveen Tammana
University of Edinburgh

Rachit Agarwal
Cornell University

Myungjin Lee
University of Edinburgh

Abstract

Monitoring and debugging large-scale networks remains a challenging problem. 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).

We present SwitchPointer, a network monitoring and debugging system that integrates the best of the two worlds. SwitchPointer exploits end-host resources and programmability to collect and monitor telemetry data. The key contribution of SwitchPointer is to efficiently provide network visibility by using switch memory as a "directory service" — each switch, rather than storing the data necessary for monitoring functionalities, stores *pointers* to end-hosts where relevant telemetry data is stored. We demonstrate, via experiments over real-world testbeds, that SwitchPointer can efficiently monitor and debug network problems, many of which were either hard or even infeasible with existing designs.

1 Introduction

Managing large-scale networks is complex. Even short-lived problems due to misconfigurations, failures, load imbalance, faulty hardware and software bugs can severely impact performance and revenue [15, 23, 31].

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 at switches [7, 20, 30, 21, 18], and querying this data using new switch interfaces [24, 13, 25, 4] and hardware [19, 24]. 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 (§2).

At the other extreme are recent systems [23, 28] 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 folds. 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 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.

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 [28]). 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.

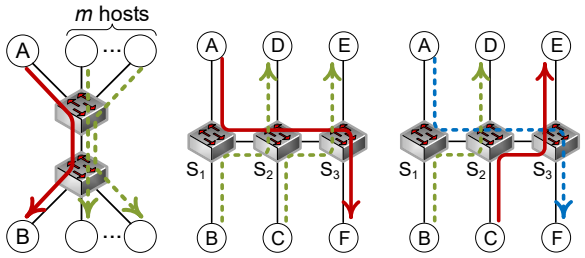
The key design choice of thinking about network switch storage as a directory service rather than a data store allows SwitchPointer to efficiently solve many problems that are hard or even infeasible for existing systems. For instance, consider the network problems shown in Figure 1. We provide an in-depth discussion in §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 1(a)), combining state distributed across multiple switches (required in Figure 1(b)), and in some cases, maintaining state even for flows that do not trigger network events (for the blue flow in Figure 1(c)).

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

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 data plane and 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 §4.1.1, and discuss how it offers a favorable 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,



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

Figure 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 §2.

making it hard to achieve line rate even for modest size packets. SwitchPointer instead uses a *perfect hash function* [1, 14] 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 §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 [4]; however, we also present a design in §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 datacenter. This allows SwitchPointer to handle asynchrony by carefully designing epoch boundaries in its switch data structures.

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 (§5, §6) demonstrates that SwitchPointer can monitor and debug network events at sub-second timescales while requiring minimal switch and end-host resources.

2 Motivation

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

2.1 Too much traffic

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

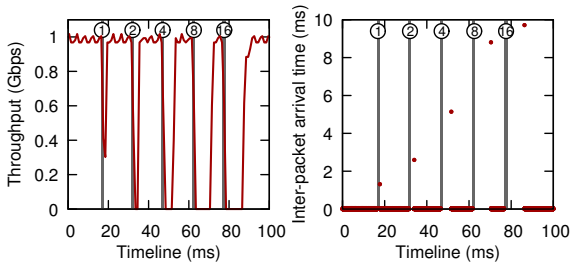
Priority-based flow contention. Consider the case of Figure 1(a), where a low-priority flow competes with many high-priority flows on an output port. As a result, the low priority flow may observe throughput drop, high inter-packet arrival times, or even TCP timeouts.

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 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.

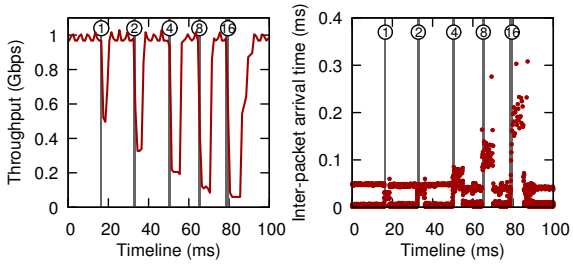
Figure 2(a) demonstrates that high-priority UDP bursts hurt the throughput and latency performance of the TCP flow significantly. With increasingly larger number of high-priority flows in the burst, the TCP flow observes increasingly more throughput drop eventually leading to starvation (*e.g.*, 0 Gbps for ∼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 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 detected and diagnosed using specialized switch hardware and interfaces [24]. 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 [23, 28] are insufficient since they only provide visibility at individual end-hosts.



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



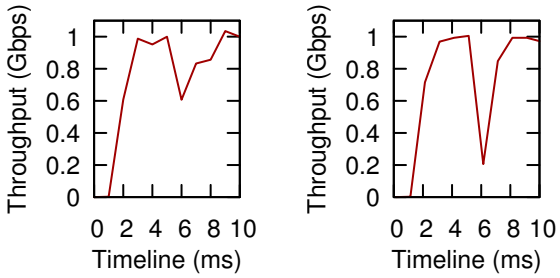
(b) Throughput (left) and inter-packet arrival time (right) of a TCP flow under microburst-based flow contention.

Figure 2: Too much traffic problem depicted in Figure 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.

2.2 Too many red lights

We now consider the network problem shown in Figure 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 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 3. The destination of the TCP flow sees a sudden throughput drop almost down to



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

Figure 3: Too many red lights problem depicted in Figure 1(b). UDP is used for flows B-D and C-E and TCP for flow A-F.

200 Mbps. This is a consequence of performance degradation accumulated across two switches S_1 and S_2 — Figures 3(a) and 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.

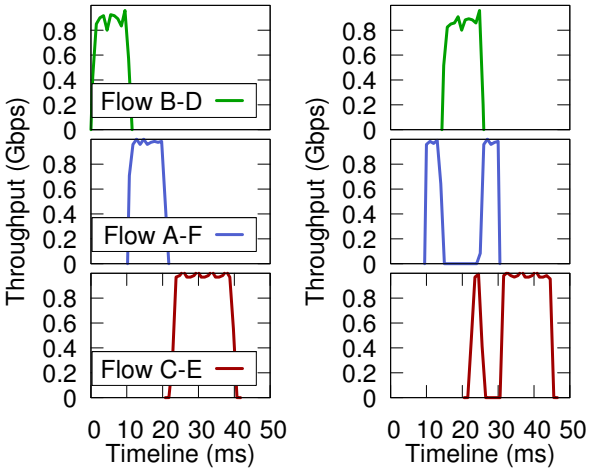
Limitations of existing techniques. The too many red lights problem highlights the importance of combining in-network and end-host based approaches to network monitoring and debugging.

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

2.3 Traffic cascades

Finally, we discuss the traffic cascade phenomenon from Figure 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



(a) (b)

Figure 4: Traffic cascades problem depicted in Figure 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.

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 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 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 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.

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 [8].

3 SwitchPointer 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.

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 [27, 28] 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 [4] to embed epochIDs into packet headers; however, we also present a design in §4.1.3 that extends the techniques in [27, 28] 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 §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 §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 [28, 23], uses end hosts to collect and monitor telemetry data carried in packet headers, and to trigger spurious network events. SwitchPointer uses PathDump [28] 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 §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 §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 1 and §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 §5).

4 SwitchPointer

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

4.1 SwitchPointer 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.

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 5 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

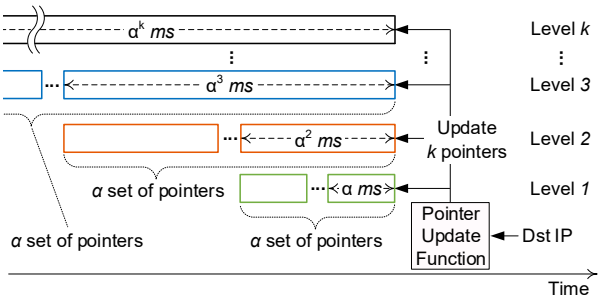


Figure 5: SwitchPointer's hierarchical data structure for storing pointers. For each packet that a switch forwards, SwitchPointer stores a pointer to the packet's destination end-host along a hierarchy of k levels. For epoch size α ms, level h ($1 \leq h \leq k-1$) stores pointers to destination end-hosts 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.

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. 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 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.

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 §6.1). 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.

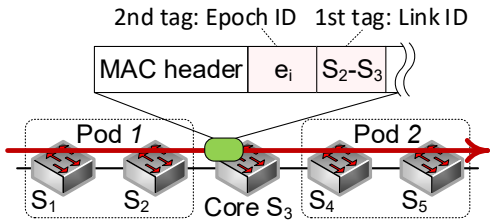


Figure 6: Telemetry data embedding using two VLAN tags using a modified version of the technique in [27]. See §4.1.3 and §4.2.1 for discussion.

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 [27, 28] 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) without any hardware modifications. Specifically, it is shown in [27, 28] 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).

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 [4] to support trajectory tracing and epoch embedding over arbitrary topologies.

4.2 SwitchPointer End-hosts

SwitchPointer uses PathDump [28] 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.

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 [4], this is fairly straight forward. For implementation using commodity switches (using techniques discussed in §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 [27, 28], 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 $[e_i - (\varepsilon + j \cdot \Delta)/\alpha, e_i + \varepsilon/\alpha]$ and for the downstream switches of S, it is $[e_i - \varepsilon/\alpha, e_i + (\varepsilon + j \cdot \Delta)/\alpha]$, where j is hop difference between an upstream (or downstream) switch and switch S. Suppose $\alpha = 10$ ms, $\varepsilon = \alpha$ and $\Delta = 2 \cdot \alpha$. For instance, in the example of Figure 6, 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 [9]). Second, millisecond-level precision is sufficient as SwitchPointer epochs are of similar granularity.

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 [3], and implements a variety of techniques (similar to those in existing end-host based systems [28, 23]) to monitor spurious network events.

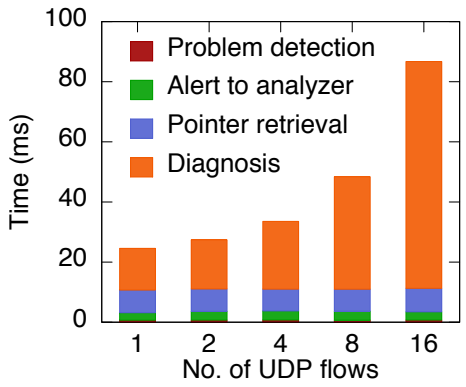


Figure 7: Debugging time of the priority-based flow contention problem depicted in Figure 2(a). SwitchPointer is able to monitor and debug the problem in less than 100ms. We provide a break down of the diagnosis latency later in Figure 12.

4.3 SwitchPointer 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 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.

5 SwitchPointer Applications

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

5.1 Too much traffic

We debug the problem discussed in §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 7 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 7, 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 §6.2.

5.2 Too many red lights

This problem illustrated in Figure 1(b) (for its behavior, see Figure 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 §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.

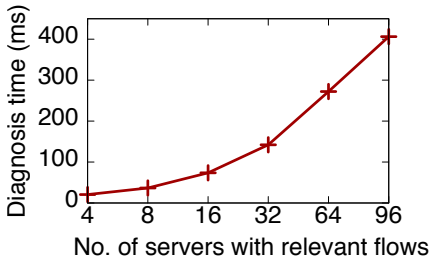


Figure 8: Latency for diagnosing load imbalance problem.

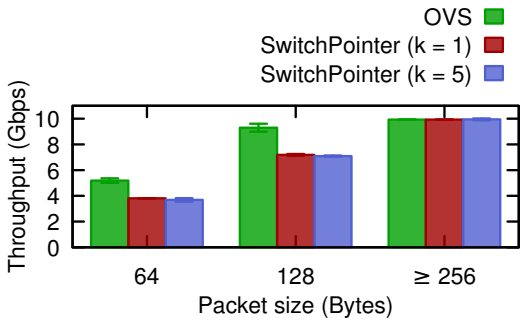


Figure 9: For smaller packet sizes, SwitchPointer is unable to sustain line rate due to overheads of perfect hash function. SwitchPointer is able to achieve line rate for a 10GE interface for packets of size 256**bytes and more.**

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 1(c) for the problem illustration and Figure 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 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 $\sim$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.

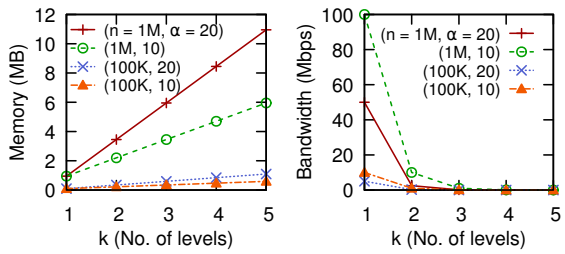
5.4 Load imbalance diagnosis

To demonstrate the way SwitchPointer works for diagnosing load imbalance, we create the same problematic setup used in [28]. In that setup, a switch that is configured 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 8 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 §6.2 for understanding individual factors that contribute to the diagnosis time.

6 SwitchPointer Evaluation

We prototype SwitchPointer on top of Open vSwitch [6] over Intel DPDK [2]. To build a minimal perfect hash function, we use the FCH algorithm [14] among others in CMPH library [1]. We also implement the telemetry data extraction and epoch extrapolation module (§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 [5]. We now evaluate SwitchPointer in terms of switch overheads and query performance under real testbeds that consist of 96 servers.



(a) Memory overhead (b) Bandwidth overhead

Figure 10: 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.

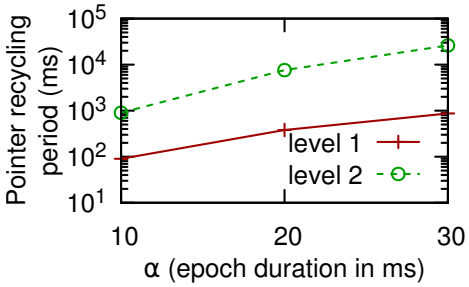


Figure 11: Recycling period of a pointer when $k = 3$.

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 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 9, 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 [10], median value of 250 bytes for hadoop traffic [26]), the throughput of SwitchPointer can be acceptable. We also envision that a hardware implementation atop programmable switch [11, 19] 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 10(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.

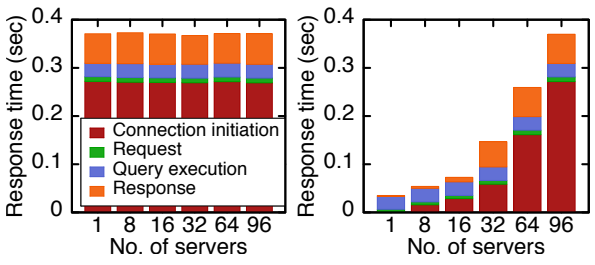
Bandwidth. In contrast to memory overhead, the bandwidth requirement of system bus between SRAM and off-chip storage decreases as we increase k and α because larger values of those parameters make the pointer flush less frequent. In particular, k has a significant impact in controlling the bandwidth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

The results in Figures 10(a) and 10(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 11 shows a tradeoff between α 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.

6.2 Query performance

We now evaluate the query performance of SwitchPointer, which we compare with that of PathDump [28]



(a) PathDump (b) SwitchPointer

Figure 12: 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.

(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 12 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 implementation 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.

7 Related Work

SwitchPointer's goals are related to two key areas of related work on network monitoring and debugging.

End-host based approaches. These approaches [23, 28, 29, 12, 15, 22] typically exploit the fact that end-hosts have ample resources and support for programmability needed to monitor and diagnose spurious network events. As discussed in §2, these approaches lack the network visibility needed to debug a class of network problems. SwitchPointer incorporates such visibility by using switch memory as a directory service thus enabling monitoring and debugging for a larger class of network problems [8].

In-network approaches. In-network approaches to network monitoring and debugging have typically focused on designing novel switch data structures [30, 20, 21, 18, 7], abstractions [17, 24, 16, 25, 13] and even switch hardware [24] to capture telemetry data at switches. While interesting, these approaches are often limited by switch and data plane resources required to store and query the telemetry data. Moreover, as discussed in §2, existing in-network approaches are insufficient to debug network problems that require analyzing data captured across multiple switches. SwitchPointer is able to overcome these limitations of in-network approaches using limited switch resources (4-6 MB of SRAM and 1-2 Mbps of bandwidth between the data plane and the control plane) by delegating the tasks of collecting and monitoring the telemetry data to the end-hosts, and by using switch memory as a distributed directory service.

8 Conclusion

SwitchPointer is 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.

Acknowledgments

We would like to thank anonymous NSDI reviewers and our shepherd Mohammad Alizadeh for their insightful comments and suggestions. We would also like to thank Minlan Yu for many discussions during the project. This work was in part supported by EPSRC grants EP/L02277X/1 and EP/N033981/1, a Google faculty research award, and NSF grant F568379.

References

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

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

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

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

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

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

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

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

[9] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM*, 2010.

[10] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding Data Center Traffic Characteristics. *ACM SIGCOMM CCR*, 40(1), Jan. 2010.

[11] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *ACM SIGCOMM*, 2013.

[12] 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.

[13] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ACM SIGPLAN ICFP*, 2011.

[14] E. A. Fox, Q. F. Chen, and L. S. Heath. A Faster Algorithm for Constructing Minimal Perfect Hash Functions. In *ACM SIGIR*, 1992.

[15] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *ACM SIGCOMM*, 2015.

[16] A. Gupta, R. Birkner, M. Canini, N. Feamster, C. Mac-Stoker, and W. Willinger. Network monitoring as a streaming analytics problem. In *ACM HotNets*, 2016.

[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] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang. Sketchvisor: Robust network measurement for software packet processing. In *ACM SIGCOMM*, 2017.

[19] 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.

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

[21] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *ACM SIGCOMM*, 2016.

[22] 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.

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

[24] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *ACM SIGCOMM*, 2017.

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

[26] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM*, 2015.

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

[28] P. Tammana, R. Agarwal, and M. Lee. Simplifying Datacenter Network Debugging with PathDump. In *USENIX OSDI*, 2016.

[29] 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.

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

[31] 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.