Synchronized Network Snapshots

Nofel Yaseen
University of Pennsylvania
nyaseen@seas.upenn.edu

John Sonchack
University of Pennsylvania
jsonch@seas.upenn.edu

Vincent Liu
University of Pennsylvania
liuv@seas.upenn.edu

ABSTRACT

When monitoring a network, operators rarely have a fine-grained and complete view of the network's state. Instead, today's network monitoring tools generally only measure a single device or path at a time; whole-network metrics are a composition of these independent measurements, i.e., an afterthought. Such tools fail to fully answer a wide range of questions. Is my load balancing algorithm taking advantage of all available paths evenly? How much of my network is concurrently loaded? Is application traffic synchronized? These types of concurrent network behavior are challenging to capture at fine granularity as they involve coordination across the entire network. At the same time, understanding them is essential to the design of network switches, architectures, and protocols.

This paper presents the design of a Synchronized Network Snapshot protocol. The goal of our primitive is the collection of a network-wide set of measurements. To ensure that the measurements are meaningful, our design guarantees they are both causally consistent and approximately synchronous. We demonstrate with a Wedge100BF implementation the feasibility of our approach as well as its many potential uses.

CCS CONCEPTS

• **Networks** → **Network measurement**; **Network monitoring**; *Programmable networks*;

KEYWORDS

Whole-network measurement, Network snapshots

ACM Reference Format:
Nofel Yaseen, John Sonchack, and Vincent Liu. 2018. Synchronized Network Snapshots. In *SIGCOMM '18: SIGCOMM 2018, August 20–25, 2018, Budapest, Hungary*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3230543.3230552

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCOMM '18, August 20–25, 2018, Budapest, Hungary
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5567-4/18/08…$15.00
https://doi.org/10.1145/3230543.3230552

1 INTRODUCTION

As networks continue to grow in size and bandwidth, a detailed understanding of their overall behavior is increasingly difficult to come by. Consider the question: does my network's load balancing protocol balance the network's load? A definitive answer to this question (and others like it) is out of the scope of traditional measurement tools.

In order to answer it, we would need visibility into the fine-grained behavior of the entire network. Instead, the target of traditional tools like switch counter polling and packet sampling are individual entities in the network. Comparison of measurements of different entities is difficult beyond just averages and long-term behavior. Slightly better are path-level metrics like those gathered at the end host [42], through Explicit Congestion Notification (ECN) [2], or In-band Network Telemetry (INT) [22]. These path-level metrics provide similar data as counters and packet sampling, but on the level of entire paths; measurements from different paths are, however, still only comparable at a coarse granularity.

Thus, when faced with questions about network-wide behavior, operators are forced to approximate the answer using tangential, but more easily collectible measurements. In the case of load balancing, they might redefine the definition of balance to a purely local metric (e.g., monitoring packet drops or buffer utilization for 'high' values) or look only at average load. Similar workarounds exist for most questions an operator might ask [1, 31, 42], but these approximations can be misleading, especially in networks with bursty load and/or high capacity [41]. The design of network switches, architectures, and protocols depend on understanding network behavior both in detail and at a network-wide scale.

This paper presents the design of a fine-grained, accurate, and precise measurement primitive that operates on the scale of an entire network. The goal of our primitive is the capture of a *Synchronized Network Snapshot*: a set of local measurements that together provide a coherent image of the entire network data plane at nearly a single point in time. Enabling our work is a recent trend toward highly programmable switch data and control planes. We leverage these tools to implement a system, Speedlight, for taking synchronized network snapshots on Wedge100BF-series switches. The implementation uses P4 and the code is open source.[1]

[1] `https://github.com/eniac/Speedlight`

Compared to more traditional measurement primitives, synchronized network snapshots are a fundamentally distributed operation—one that involves tight coordination of the control and data planes of multiple network devices. Through coordination, network snapshots are able to guarantee both causal consistency (i.e., that the measured values are coherent) and approximate synchronicity (i.e., that the measurements were taken near-contemporaneously). The primitive itself is agnostic to the type of local measurement and supports the collection of any variable accessible from the data plane: counters, packet samples, switch state, queue depth, etc. It is also amenable to partial deployment.

At its core, our system is inspired by distributed snapshot protocols [10, 23]. In a classical distributed snapshot, a snapshot initiator sends out a message that propagates among a set of distributed nodes to cause them to (without stopping the system or synchronizing clocks) take snapshots of their local state. The guarantee provided by these protocols is that the snapshot creates a causally consistent partition of the system's events. For any event e that is 'pre-snapshot', any event that can be construed as *causing* e is also pre-snapshot. In the context of networks, this might mean that if a snapshot of queue depth captures a packet p in a queue q, that p will not be counted as part of any other queue, and furthermore that the effects of every send and receive that led to p being in that particular queue at that particular time are also included in the snapshot. For that reason, distributed snapshots are an attractive abstraction; however their application to high-speed networks carries a few challenges.

First, while traditional snapshots provide a set of measurements that *could* have happened simultaneously, one of their primary criticisms is that they do not provide any guarantee of how close in time the measurements occurred. Second, snapshot protocols often make strong assumptions about the system, e.g., that nodes are single-threaded and capable of arbitrary computation, and that they are connected via reliable FIFO channels. Real switches, on the other hand, are highly parallel, extremely limited in their data plane processing capabilities, and exhibit non-FIFO behavior (e.g., prioritization, packet re-circulation, etc.). It can be difficult to adapt certain functionality to programmable data planes [35, 36], and distributed snapshots are no exception.

The key insight of Speedlight is that modern switches are two-level devices. The data plane can perform extremely fine-grained in-band processing of network traffic, but is fundamentally limited in the type of computation and resources available. Augmenting the data plane is a control plane with the opposite tradeoffs.

Speedlight therefore splits the responsibility of taking snapshots such that the data and control planes each mitigate the weaknesses of the other. At a high level, we first break the data plane of each switch into small, simple components that obey single-threaded and FIFO assumptions. The snapshot implementation at each of these data plane components is not fully featured, but provides two key properties: (1) it allows for multiple simultaneous snapshot initiators in the style of [38], and (2) it guarantees consistency and correctness in all cases, regardless of data plane limitations. The control plane CPU is then responsible for the global, PTP-coordinated initiation of a snapshot at all data plane components, as well as the stitching together of results.

The end result of our system is that all of the individual measurements in a synchronized network snapshot are not only consistent, they are guaranteed to occur almost contemporaneously. Our current implementation guarantees a drift of at most 10s of microseconds (less than a single RTT in most cases); drift can be decreased further using more advanced time synchronization techniques [25]. In addition to presenting a detailed design and implementation, we demonstrate the primitive on real workloads. To summarize, our work makes the following contributions:

- We present a Synchronized Network Snapshot algorithm for the collection of distributed state within the data plane of a network. Our design provides strong guarantees regarding both the semantics of the measured values and their timeliness.
- We then present the design and implementation of Speedlight, a practical realization of the Synchronized Network Snapshot algorithm. Our prototype, built for Wedge-100BF-series switches, is able to achieve microsecond-level synchronization of global network snapshots.
- Finally, we use our system to measure real workloads running on our testbed. This measurement study demonstrates both feasibility and usefulness of our approach.

2 BACKGROUND AND MOTIVATION

Network measurement is a method through which we seek to understand network behavior. This can be in the context of designing new protocols/architectures, evaluating existing ones, or diagnosing issues in live networks. Over the years, a wide range of network measurement tools and analyses have been created to assist in the aforementioned tasks.

Measuring path-level properties. One common approach is the use of end-to-end or flow/path-level measurement tools. Extremely flexible, these tools enable observers to evaluate, from the network edge, the aggregate effect of the network in the context of application-level measures like latency, throughput, and drop rate. An advantage to this approach is that it accurately reflects the overall experience of application traffic. They also often do not require additional network support, although there are recent exceptions [2, 22]. Though effective for some use cases, edge vantage points

typically lack visibility into fine-grained network behavior and details of the network's structure [31].

Measuring devices. A much more fine-grained approach is to measure individual network components directly. This typically takes the form of counters or packet sampling/mirroring, but recent proposals have explored the use of more complex metrics like flow-based queries, heavy-hitter analysis, and TCP-level statistics [29, 31, 39, 40]. Direct measurement is precise, and with sufficient device support, quite expressive.

2.1 Whole-network Measurement

Path-level and device-level metrics form the foundation of today's measurement tools. Unfortunately, by themselves both approaches typically provide little to no guarantees about the relationship between measurements, or the effect of clock drift and other asynchronous behavior.

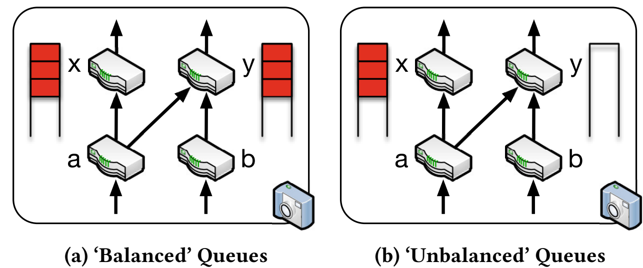
For bursty and/or high-capacity networks, even small amounts of unattended asynchronicity can lead to large inaccuracies in measurement. As an illustrative example, consider a datacenter network. A good NTP accuracy within a LAN is 1 ms; in contrast, typical datacenter RTTs are an order of magnitude lower, and there is evidence that traffic bursts can be even shorter ($O(10\,\mu s)$) [41]. In effect, for any two measurements of network behavior at different locations, their relationship is both tenuous and difficult to bound. This inaccuracy will only grow as network speeds increase.

Even within a single router, synchronized information is not always available. Counters may be on different line cards and most counter polling mechanisms are not optimized for polling more than one counter at a time. Without driver-level modifications, polling a single counter on a modern switch typically takes on the order of 1 ms [41].

For the above reasons, measurements are not often compared directly. Instead, when trying to examine network-wide behavior, most frameworks aggregate individual measurements, typically using statistical analysis over relatively long time periods so as to skirt the issue of unsynchronized clocks. Averaging and summation are particularly common mechanisms. An observer can compare average utilization of multiple components to determine how they differ over a given time span. They can also use a total path-level drop count in combination with network tomography to pinpoint lossy components. Network operators have become creative in their techniques to obliquely measure the whole network; however, as we will see in the next section, there are still fundamental limitations to existing tools.

2.2 A Case for Consistency

To illustrate the importance of consistent whole-network measurement, imagine we have the simple network depicted in Figure 1. The network consists of two ingress routers



(a) 'Balanced' Queues (b) 'Unbalanced' Queues

Figure 1: Asynchronous measurements can be misleading. These diagrams show two possible measurements of queue depth for x and y. In both cases, the network could be perfectly balanced or arbitrarily unbalanced—the measurements fail to distinguish between the two cases.

(a and b) connected to two egress routers (x and y) in an asymmetric fashion. Even for this simple case, many critical questions about network behavior are difficult to answer.

1. Is the network load balanced? We begin with the question asked in Section 1. Imagine that an operator deploys a new load balancing protocol to a and b. How does she evaluate its efficacy? How would she know if there was a performance bug in the protocol? How does she quantify the room for improvement?

One possible solution is to sample the queue depth at x and y; however, on their own, these samples do not answer the above questions. Particularly in the presence of bursty traffic, asynchronous measurements can provide misleading results. For instance, the balanced queue measurements shown in Figure 1a could be a result of (a) a perfectly balanced network in which queue depths never differ, (b) an entirely unbalanced network in which one queue is always empty, or (c) anything in between. All of the above is still true if we observed unbalanced queues as in Figure 1b.

Common workarounds include averaging many samples (an approach that captures biases and long-term effects, but is not general) or only analyzing relative performance compared to a previous solution (an approach that is not always possible, and whose utility is limited). Instead, a set of contemporaneous measurements would give a more meaningful view into the behavior of the network.

2. Where should we add capacity to the network? A related question is where an operator should add capacity to the network, i.e., the process of network provisioning. Today, they might examine tail utilization or drops over every link to identify bottlenecks in the network. Asynchronous measurements are sufficient for this, but fail to answer many followup questions. For instance, would adding a parallel path alleviate congestion or is a per-link capacity upgrade necessary? Balanced load among existing paths would indicate the former, while localized hotspots would indicate the

latter. They provide similarly limited insight into whether alleviating one bottleneck would lead to others. Again, contemporaneous measurements would provide more insight into network behavior.

3. Is traffic synchronized? Synchronized measurements can also assist in application-level debugging, especially in the case of TCP incast and related performance problems. Many of the same issues from the previous questions also apply here. Today, detection of synchronized behavior is typically done either empirically (e.g., testing if added jitter in TCP sends alleviates the problem), or obliquely (e.g., testing for characteristics of incast like high flow count, TCP timeouts, and drops [31, 42]). These workarounds are both inaccurate and only possible after performance has already been impacted. We argue that a whole-network measurement primitive is a more natural and effective alternative.

4. What is the global forwarding state? Finally, a classic problem in networking is the detection of bad forwarding state. Forwarding loops are the canonical example of an undesirable network state that is difficult to detect, especially if the loops are transient and/or flapping. This class of problems have taken a newfound importance in the context of RDMA and RoCE. RoCE's PFC mechanisms can cause network deadlocks, not only when there are routing loops, but in many other cases as well [17]. For a general method of verifying and diagnosing these issues, a consistent snapshot is crucial—otherwise we can observe states that are impossible.

3 OVERVIEW

We seek to design a measurement primitive that captures a set of measurements representing a meaningful view of the whole network as it appears at a single point in time. We note that in pursuit of this goal, a truly simultaneous network-wide snapshot is impossible without either freezing the network or using prohibitively expensive hardware like atomic clocks. Instead, our goal is a snapshot primitive with the following two properties:

- *Causal consistency:* If a measurement in snapshot S includes the effect of event e (e.g., a packet reception), S also includes the effects of every event that led to e.
- *Near synchronicity:* The time difference between every pair of measurements in the snapshot is guaranteed to be at most d, where $d < RTT$. Our prototype guarantees $d < 100\,\mu s$, even for large networks.

Rather than capturing the true instantaneous behavior, i.e., what one would have seen if we froze time to examine the network, causal consistency provides a record of what *could* have happened. Augmented with a tight bound on the maximum jitter of the snapshot, we argue that the combination of these two requirements preserves most useful metrics.

Architecture. Our design for synchronized network snapshots involves three types of entities: ***data-plane processing units*** of which each switch/router can have many, ***control planes*** running at each device, and ***snapshot observers*** running on hosts connected to the network. Our design allows for partial deployment (Section 10) as well as a wide range of networking technologies and configurations. It is also agnostic to the measured data—*any value accessible at line rate in the data plane can be snapshotted.* It is achieves all of this with minimal additional state and overhead.

Protocol. At the core of our design is a modified version of the Chandy-Lamport snapshot algorithm. Originally created in the context of distributed systems, snapshots seek to capture the global state of a system without a common clock or shared memory, and without affecting the operation of the system itself. What Chandy and Lamport proposed was a protocol in which an initiator can trigger a cascade of messages that, with causal consistency, partitions the system's events into 'pre-snapshot' and 'post-snapshot', then collects the state of every node and channel at the boundary.

There are two key differences between our version and the original. First, while most snapshot implementations begin with a single initiator, snapshots in our system are initiated at *all nodes simultaneously.* Second, our design is necessarily bipartite. Modern control planes, typically running on a general purpose CPU, can easily implement a fully featured snapshot protocol, but in terms of consistency, they are no better than a remote host. Recent proposals for programmable data planes, on the other hand, more closely adhere to the assumptions of the Chandy-Lamport protocol, but today's ASICs have limited functionality/resources. By leveraging both, we seek to mask each plane's deficiencies with the other's strengths—neither is sufficient on its own.

Operation. A Synchronized Network Snapshot begins humbly: with a host acting as a snapshot observer. The observer broadcasts a request to every device in the network to take a snapshot of a given metric at a given time in the future. The control planes running on every device then coordinate among themselves using a protocol like PTP to achieve the synchronized, network-wide initiation of a data plane snapshot. The data plane, where processing elements most closely adhere to the requirements of Chandy-Lamport, implements the core of a multi-initiator snapshot protocol, while the control plane fills in missing pieces of the protocol as necessary.

4 NETWORK SNAPSHOT ALGORITHM

Before we describe the detailed design of our system for Synchronized Network Snapshots, we first introduce the design of an idealized data plane snapshot algorithm. In Sections 5 and 6, we describe how we adapt this algorithm to current hardware.

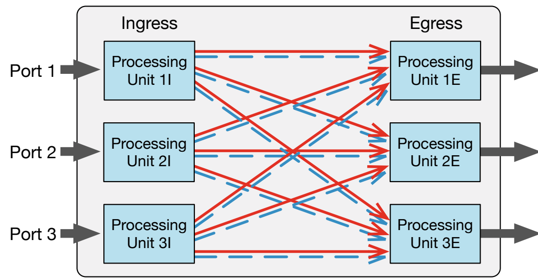


Figure 2: A conceptual model of a router in a network snapshot. At the lowest layer are the ingress/egress processing units of individual ports. Connecting the ingress and egress ports are unidirectional FIFO channels. Multiple channels may exist in the case of Class-of-Service queues (two in the diagram, represented by solid red and dotted blue arrows).

4.1 System Model

Abstractly, a network is a collection of switches and routers. Each switch or router can be subdivided at many different levels. At the highest level, modern switches will often contain one or more line cards, each line card being responsible for one or more ports. The ports can be subdivided further into ingress and egress processing units (see Figure 2), although in some designs, multiple logical processing units can be implemented with a single physical unit. Regardless, the *per-port, per-direction processing unit* forms the fundamental building block of packet processing: despite aggressive amounts of parallelism, for a single port and single direction, processing is guaranteed to be linearizable.

Connecting the ports are unidirectional communication channels. Within the device, the ingress processing unit of each port is logically connected to the egress processing unit of every other port.[2] These connections can potentially contain multiple sub-channels when the switch is configured to prioritize certain traffic. In those cases, individual classes of service (CoS) obey FIFO ordering, but the packets of different service classes can be interleaved. Between devices are physical links that connect the egress processing unit of one port to an ingress unit on a different network device.

In modern Ethernet, there is only one device sitting on either end of the channel. In other types of networks or in partial deployment scenarios, a logical channel exists between every connected egress and ingress.

4.2 Protocol

The original Chandy-Lamport algorithm relied on a few key assumptions: linearizable nodes, simplex FIFO channels, no

[2]Some devices allow for more complex internal packet communication, e.g., recirculation. If configured, those channels can be handled by adding additional logical channels to our model.

```
• state: Local state to be snapshotted.
• snaps[]: Set of snapshotted state.
• sid: Current snapshot ID. Starts at 0.
– lastSeen[]: The last IDs seen from each upstream neighbors.

1  Function onReceiveCS(pkt):
2      if pkt.sid > sid then
3          /* New snapshot */
4          for i ← sid + 1 to pkt.sid do
5              snaps[i] ← state
6          sid ← pkt.sid
7      else if pkt.sid < sid then
8          /* In-flight packet */
9          for i ← pkt.sid + 1 to sid do
10             Update channel state of snaps[i] with pkt
11     lastSeen[pkt.sender] ← pkt.sid
12     All snapshots up to min(lastSeen[*]) are complete
13     Update state and set pkt.sid ← sid

14 Function onReceiveNoCS(pkt):
15     if pkt.sid > sid then
16         for i ← sid + 1 to pkt.sid do
17             snaps[i] ← state
18         sid ← pkt.sid
19     All snapshots up to sid are complete
20     Update state and set pkt.sid ← sid
```

Figure 3: Per-processing-unit pseudocode for our idealized network snapshot protocol (w/ and w/o channel state). The match-action approximation and other details are described in Sections 5 and 6. Global state preceded by '–' is only necessary for channel state.

message drops, and bounded delay. When considering a network of routers, few if any of these assumptions hold. Instead, our network snapshot protocol operates over the network of per-port, per-direction processing units connected by logical communication channels (either a physical link or an internal, logical CoS queue). This formulation gives us a distributed system of linearizable nodes connected by FIFO channels. To handle drops and delays, we take inspiration from subsequent work (e.g., Li et. al. [27]) and classical network assumptions. While snapshot protocols exist for other, more relaxed system models, they typically require massive storage requirements, delaying of messages, or they limit the gathered state to packet/byte counts.

Figure 3 depicts our algorithm in pseudocode. Every processing unit keeps track of its current snapshot ID, s, initialized to 0. They also keep track of the local state that is the target of the snapshot. Note that this requires snapshots of shared state (e.g., a switch-wide packet counter) be taken as a set of local snapshots or re-implemented as local state.

Every packet carries a snapshot ID field, s_p that indicates the epoch from which it was sent (similar to [27]). 'Piggybacking' of markers on every packet ensures that snapshot

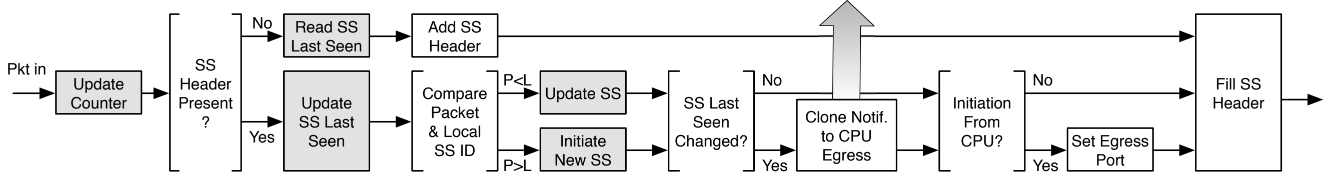


Figure 4: Pipeline of an ingress snapshot processing unit. Shaded boxes involve stateful registers.

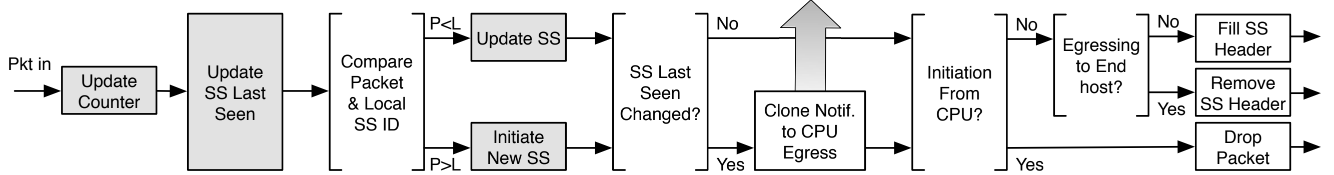


Figure 5: An egress processing unit. Shaded boxes involve stateful registers. Not shown is the CPU egress path.

ID updates are resilient to packet loss. On receipt of a packet, processing units compare the packet's carried snapshot ID with their local ID. If $s_p > s$, the upstream neighbor has begun a new snapshot, and the current node should as well. The local state is immediately saved and the local ID is updated ($s \leftarrow s_p$). If, on the other hand, $s_p < s$, the packet was in-flight when the snapshot occurred, and should therefore be included in the channel's state.

The specifics of how channel state should be recorded is metric-dependent. For instance, a network-wide packet count might require processing units to record the number of packets in their queue, then add in-flight packets to the count as they arrive. In other cases, the operator may not care about channel state at all (e.g., instantaneous queue depth measurements), and can omit this step. Either way, the processing unit sets $s_p \leftarrow s$ before forwarding. When packets arrive with $s_p = s$, no actions are necessary. The above process ensures causal consistency of recorded states.

Initiating a snapshot. Snapshots can be concurrently initiated at any number of processing units by incrementing their local snapshot ID. The affected processing units will tag all subsequent packets with the incremented ID. Assuming that the network is strongly connected and there is regular traffic flowing along every channel, even a single initiator will eventually cause all processing units to take the snapshot. When those assumptions break down, re-initiations may be necessary to ensure liveness. We discuss the details and practical challenges of snapshot initiation in Section 6.

Completing a snapshot. If channel state is not important to the measurement, a processing unit is finished with its snapshot as soon as it records its state and updates its local snapshot ID. Otherwise, it is finished when it sees that all of its upstream neighbors have updated their ID. At that point, there is no possibility of receiving additional in-flight packets ($s_p < s$). To detect this, each processing unit stores an array of the last seen ID from every upstream neighbor. Lines 11 and 12 in Figure 3 implement this process. With or without channel state, a network-wide snapshot s' is complete when all nodes in the system are finished with snapshot $s > s'$. As with snapshot initiation, we discuss the practical concerns of snapshot completion in real networks in Section 6.

Proof sketch. The proof of correctness for our algorithm mirrors that of prior work, but we provide a brief sketch of the proof here. For each state-affecting event e on node n, $e \in PRE$ ('pre-snapshot') if it occurs before the local snapshot on n. The algorithm is correct if, for all $e \in PRE$, if e' happens causally before e, then $e' \in PRE$.

(1) If e and e' are on the same processing unit, the above is trivially true.
(2) Otherwise, $e \in PRE \Rightarrow e' \in PRE$ by contradiction.
   (a) Assume for snapshot i that $e' \notin PRE$ is a send of packet p and $e \in PRE$ is the matching receive.
   (b) Since $e' \notin PRE$, p must be carrying snapshot ID i.
   (c) That is not possible since $e \in PRE$, thus there is a contradiction.
   (d) Similar logic can be applied to other relationships between e and e'.

5 DATA PLANE COORDINATION

This section is the first of two that describes in detail the design of Speedlight. Speedlight leverages the match-action stages and stateful memory found in emerging programmable ASICs such as the Barefoot Tofino [8]. Using these tools, each processing unit can execute limited computation over packet headers/metadata using state in the form of register arrays.

Though the ASICs are powerful, their limitations and other network-specific concerns make the translation from the preceding snapshot algorithm to Speedlight difficult. This

section describes the design of Speedlight's data plane while Section 6 describes the control plane that complements it.

5.1 Packet Headers

As mentioned in Section 4.2, network snapshots require additional header information. Speedlight does not require host cooperation, so headers are added by the first snapshot-enabled router, and removed before delivery to hosts. The required fields are as follows. If channel state is not desired, items preceeded by a **–** may be omitted.

- **Packet Type** can take one of two values: initiation or data. Most traffic is classified as data; initiation packets are special control messages that we describe in Section 6.
- **Snapshot ID** is set at each hop to be the processing unit's current snapshot ID. Conceptually, it specifies the snapshot to which the send of the packet is a member, and informs the current processing node whether the packet is part of a new one, or in-flight from an old one.

– **Channel ID** uniquely identifies each upstream neighbor. If there are multiple channels between neighbors, there should be an ID for each. Our reference implementation assumes switched Ethernet and no packet re-submission, so for ingress processing units, there is only one upstream neighbor (the external neighbor), and for egress units, the number of upstream neighbors is bounded by the number of ingress ports on the local router.

5.2 Stateful Variables

Some amount of inter-packet persistent state is also required in each processing unit. These mirror the state in Figure 3.

- **Counters** store target local state of the snapshot. These are managed separately from the snapshot protocol. This variable corresponds to *state* in Figure 3.
- **Snapshot ID** is an integer representing the node's current snapshot ID. This value corresponds to *sid*.
- **Snapshot Value[max snapshot id]** stores the snapshoted state and, if necessary, channel state. These must be encoded into a value that fits into available register space. Equivalent to *snaps*.

– **Last Seen[# of neighbors]** tracks the last snapshot ID from each upstream neighbor. See definition of Channel ID for a discussion of what constitutes an upstream neighbor in our system. Corresponds to *lastSeen*.

5.3 Packet Processing Procedure

Figures 4 and 5 show the operation of ingress and egress processing units in Speedlight. Both approximate the algorithm presented in Section 4 with a few notable differences.

In both types of processing units, the first step is to read the target state and update it. The update process is orthogonal to the snapshot logic, only intersecting if the target state requires it (e.g., to ignore snapshot traffic). The next step is to examine the snapshot header.

The core of the snapshot processing procedure is similar to the one described in Section 4.2. The processing unit updates the neighbor's last seen value and then tests to see if the packet's snapshot ID is less than, greater than, or equal to the processing unit's local ID. As mentioned in Section 4.2, in-flight packet handling is metric-specific and configured by the network operator, and much of the algorithm can be elided if channel state is not necessary.

Differences from the idealized algorithm. The primary differences between Speedlight's data plane and the algorithm in Section 4.2 derive from hardware limitations in high-speed programmable switches. One key limitation is that today's switches do not have the ability to loop through (at line rate) intermediate snapshot IDs when the packet's ID and the local ID differ by more than 1. Re-circulation loops are not possible as they would violate FIFO ordering. Instead, our implementation produces a complete and consistent snapshot iff the ID of all upstream neighbors and the local processing unit differ by at most 1. The following section describes how we detect and mitigate inconsistency.

Another is that the space of possible snapshot IDs and storage of the snapshot state are tightly constrained. As such, Speedlight enables rollover of the snapshot ID to 0 after reaching the maximum ID. For this, we assume that no ID in the system is ever 'lapped', i.e., that the maximum difference between any two snapshot IDs in the system is ($max\ snapshot\ id - 1$). This can be enforced by the snapshot observers out-of-band. This assumption allows us to rely on the contents of the Last Seen array as a reference to detect if the packet's ID and/or the local ID have rolled over.

Snapshot Notifications. We mask the above deficiencies using the control plane. Supporting that process is a notification channel between the two planes. After any update of either the local *Snapshot ID* or of any *Last Seen* array entry, the data plane exports a notification to the CPU to assist in determining snapshot progress/completeness. For an upstream neighbor n, this notification includes the former value of LastSeen[n] along with the former and new Snapshot ID. Depending on the case, the former and new values may not be distinct. It will become clear in the following section why we need all four values.

6 CONTROL PLANE COORDINATION

Speedlight's data plane is augmented with a control plane to form a two-tier, mutualistic system in which each is responsible for masking the limitations of the other. This section examines some of the key scenarios in which the control plane is necessary.

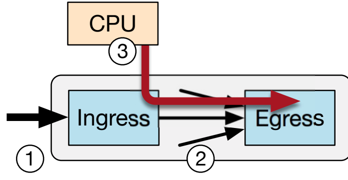


Figure 6: The three ways in which a processing unit can be induced to take a new snapshot. The initiation can come from: (1) a neighboring device, (2) another processing unit within the same device, or (3) from a control plane initiation message that, for every port, travels CPU→ingress→egress.

Synchronized snapshot initiation. One of the primary responsibilities of Speedlight's control plane is to initiate snapshots in a timely fashion. At a high level, it does this by (*a*) synchronizing clocks between the control planes of different network devices, and then (*b*) executing a global, coordinated network snapshot initiation. Clock synchronization is a well-studied field, and Speedlight leverages this existing work. In our implementation, we use PTP, although the choice of protocol is orthogonal to our design.

Coordinated snapshot initiation, (*b*), is executed using the synchronized time. A snapshot observer first schedules a snapshot i for a given time in the future by registering the event with all device control planes. When the time comes, the control planes broadcast a message to all local ingress processing units. The message includes a snapshot header with *snapshot ID* set to i, the newly initiated snapshot. The ingress unit will process this snapshot header much like a regular packet—initiating a new snapshot if i is larger than the current snapshot ID. The control plane in this case is treated as an additional neighbor for the last seen array, though this value is only used for rollover detection and not to detect snapshot completion. After processing is complete, the ingress processing unit forwards the initiation to the egress unit of the same port, which drops the packet after processing. Unlike regular snapshot header processing, the packet is not included in the update counter stage and is never considered an in-flight packet.

Including control plane initiation, there are three ways by which a processing unit can be induced to take a new snapshot. The three methods, illustrated in Figure 6, cover normal snapshot-enabled packets from external (1) and internal (2) neighbors that have already begun the snapshot, as well as the control plane initiation messages (3). With these three initiation methods, Speedlight ensures a level of start-time synchronization beyond what a similar counter polling framework could achieve (see Section 8). That is in addition to the consistency provided by the snapshot protocol itself.

Detecting snapshot completion and inconsistency. In a classical distributed snapshot, a node's local state is valid as soon as it takes a local snapshot, and the state of the channel is valid when it receives an up-to-date snapshot marker on that channel. The global snapshot is complete when all such state is valid. In Speedlight, the control plane is responsible for gathering state and detecting the completion of snapshots. It is also responsible for detecting when snapshotted values become inconsistent. This scenario only occurs when channel state is required, and is not present in the original Chandy-Lamport algorithm. Rather, it is the direct result of the hardware limitations described in Section 5.

```
• lastRead[unit]: Latest finalized snapshot for each unit.
– ctrlSnapID[unit]: Controller's view of units' current IDs.
– ctrlLastSeen[unit][neighbor]: Controller's view of the last
  seen array for each processing unit.
1  Function OnNotifyCS(unit, currentID, neighbor, currentLS):
2      if currentID ≠ ctrlSnapID[unit] then
3          /* New snapshot ID */
4          done ← min(ctrlLastSeen[unit][*])
5          for i ← done + 1 to currentID do
6              Mark i as inconsistent
7          ctrlSnapID[unit] ← currentID
8      if currentLS ≠ ctrlLastSeen[unit][neighbor] then
9          /* New last seen ID */
10         ctrlLastSeen[unit][neighbor] ← currentLS
11     toRead ← min(ctrlLastSeen[unit][*])
12     for i ← lastRead[unit] + 1 to toRead do
13         if i is not inconsistent then
14             Read snapshot value for i from unit
15     lastRead[unit] ← toRead
16 Function OnNotifyNoCS(unit, currentID):
17     if currentID ≠ lastRead[unit] then
18         validValue ← Read value for currentID from unit
19         for i ← currentID to lastRead[unit] + 1 do
20             value ← Read snapshot value i from unit
21             If value is uninitialized use validValue, otherwise
                 validValue ← value
22         lastRead[unit] ← currentID
```

Figure 7: Control plane detection of complete and inconsistent snapshots with and without channel state. Note that min() must be rollback aware, but *lastRead* can be used as a reference. Global state preceeded by '–' are only necessary for channel state.

Figure 7 shows how a Speedlight control plane processes snapshot notifications to detect completion/inconsistency both with and without channel state.

(1) *w/ Channel State:* Recall that in the common case, a processing unit is finished with snapshot i when $\forall u : lastSeen[u] \geq i$. Hardware limitations introduce an extra requirement: that the snapshot ID advances by exactly 1 each time. For example, if unit's snapshot ID is 5 and

it receives a message from the snapshot 2 epoch, ideally the data plane would increment associated channel state for snapshots 3–5. Unfortunately, current ASICs cannot execute (at line rate) the required instructions to keep those intermediate snapshot values consistent. Speedlight marks them as inconsistent and handles notification drops conservatively.

(2) *w/o Channel State:* The simpler case, a processing unit is done with a snapshot as soon as it increments its ID, records its local state, and sends a notification to the CPU. The snapshot ID can still skip forward; however, in this case, the CPU can infer the proper snapshot value. See lines 19–21 in Figure 7. Note that we must check for value initialization to account for notification drops.

All values are shipped to the snapshot observer, which assembles snapshots from all the devices with which it registered the snapshot. The observer computes completion and executes retries. If a device fails, it may timeout and be excluded from the global snapshot.

Ensuring liveness. An extension of the above two responsibilities, the control plane is also responsible for ensuring that snapshots are eventually initiated and completed at every processing unit. There are two reasons why this may not happen without assistance.

The first is packet drops of either the initiation message or update notifications. Especially for ingress processing units whose upstream neighbor is not snapshot enabled (e.g., a unit connected to an end host), a dropped initiation means that the processing unit will never advance its snapshot ID. Dropped notifications can also be problematic as they may cause snapshots to be incorrectly marked as inconsistent. To address both issues, Speedlight control planes will resend initiations for incomplete snapshots after a timeout. This is safe as duplicate and outdated control plane initiations are ignored by the data plane, and duplicate notifications are dropped at the control plane. Speedlight's control plane can also proactively poll the data plane registers to help recover from simple cases of notification drops.

The second is a lack of traffic when channel state is required. As completion of the snapshot is gated on receiving an up-to-date snapshot marker from all upstream neighbors, if there is no such traffic on which to piggyback, the snapshot may never complete. This can happen due to traffic patterns, or it can be a natural consequence of the routing configuration (e.g., when using spanning trees or up-down data center routing). Speedlight has separate mechanisms for each situation. For a traffic-related absence of packets, we can inject broadcasts into the network that force propagation of snapshot IDs. For a lack of traffic due to network structure, operators can configure the removal of non-utilized upstream neighbors from *ctrlLastSeen* consideration.

Node attachment. Finally, we discuss the process of adding new devices to the network. For every snapshot, the snapshot observer keeps a list of all currently active devices. When adding a new device, it must be registered with the snapshot observer before it is included in the next snapshot. New devices will not start with the current snapshot ID. Instead the control plane initializes all state (registers in the data plane and tracking state at the control plane) to 0. As soon as traffic arrives from neighboring devices, the snapshot will jump ahead to the current value, if it is not 0. If it does jump ahead, the snapshot observer will ignore any spurious snapshot completions as the device would not have been in its expected device set when initiating the snapshot.

7 IMPLEMENTATION

We implemented a prototype of Speedlight with all of the data plane and control plane functionality described in Sections 5 and 6 for Wedge 100BF-series switches [19]. Wedge 100BF switches are driven by the Barefoot Tofino, a commodity multi-Terabit data plane ASIC that integrates recent designs for programmable line rate packet parsing [14], match-action forwarding [9], and stateful processing [36].

7.1 Data Plane

The Speedlight data plane is a pipeline of P4 match-action tables that compiles to the Tofino. We created multiple versions for different metrics, with and without wraparound and channel state support. Each implementation contains around 1000 lines of P4-14 code. Figures 4 and 5 show the logical ingress and egress match-action pipelines, assuming a snapshot that requires channel state.

Table 1 summarizes the key resources required by our prototype, broken down by the resources' logical functionality. We make no guarantee of the optimality of our prototype; the statistics represent a rough upper bound on the resource utilization of Speedlight. Even so, the prototype occupies less than 25% of any given type of dedicated resource—the remainder can be used for other data plane functionality.

As Table 1 shows, the prototype utilizes 10 to 12 physical processing stages in the Tofino to satisfy sequential dependencies in its control flow. It does *not* prohibit those stages from also implementing other ingress or egress data plane functions. Anything independent of the snapshot logic, such as forwarding or access control, can be compiled into the same stages and operate in parallel.

Speedlight fits well with other switch responsibilities. Its data plane is most expensive in terms of stateful ALUs (sALU), used to implement operations on register arrays, e.g., updating or initializing a snapshot. This is opposite of typical data plane functionality, which tends to apply mostly stateless operations to packet headers.

Variant	Packet Count	+ Wrap Around	+ Chnl. State
Computational Resources			
Stateless ALUs	17	19	24
Stateful ALUs	9	9	11
Control Flow Resources			
Logical Table IDs	27	35	37
Conditional Table Gateways	15	19	19
Physical Stages	10	10	12
Memory Resources			
SRAM	606 KB	671 KB	770 KB
TCAM	42 KB	59 KB	244 KB

Table 1: Resource usage for the Speedlight data plane on the Tofino. Numbers are for a snapshot of per-port packet counters and 64 ports.

Resource requirements for Speedlight increase with the use of wraparound and channel state, features that require more complex logic. Memory requirements also grow with the number of ports in the snapshot, as the data plane must allocate larger register arrays and tables to store and address the per-port statistics. The configuration shown in Table 1 is for 64 port snapshots, the maximum number of ports that a single processing engine in the Wedge100BF's Tofino can support. A configuration with wraparound and channel state for 14 port snapshots, as used for evaluation in Section 8, requires 638 KB of SRAM and 90 KB of TCAM.

7.2 Control Plane

We wrote the snapshot control plane in Python (~2000 lines of code) and ran it on the switch CPU, which has a PCIe-3.0 X4 link to the Tofino ASIC. The control plane uses a compiler generated Thrift API to initialize tables, set up mirroring, and poll register arrays. Time synchronization was done via `ptp4l` and `phc2sys`.

The snapshot control plane receives notifications from the Tofino using a raw socket implemented by a kernel-level DMA packet driver. It listens for notifications, which trigger its main event handler as depicted in Figure 7. There are alternatives to this approach, e.g., a P4 digest stream, but we found that raw sockets made the implementation straightforward and offered significantly better performance.

8 EVALUATION

We evaluated Speedlight in a hardware testbed and used it to perform measurement campaigns that study widely used distributed applications and protocols. Our testbed consists of a Barefoot Wedge100BF-32X programmable switch with 128 25 GbE ports connected to six servers with Intel(R) Xeon(R) Silver 4110 CPUs via 25 GbE links. We emulated a small leaf-spine topology in our testbed, as depicted in Figure 8. We did this by splitting the 128 port switch into 4 fully isolated logical switches with lower fan-outs.

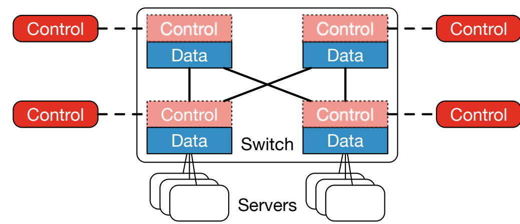


Figure 8: Depiction of our testbed topology.

As in a real deployment, the virtual switches were connected with 100 GbE passive copper links. At the data plane, all forwarding tables were replicated for each virtual switch. At the control plane, we ran duplicate versions of the protocol. To emulate clock drift between switch control planes, snapshots were initiated based on the local system clock of four distinct PTP-synchronized servers. With the inclusion of network latency, our synchronization numbers therefore represent an upper bound.

To load balance traffic along the multiple paths in our testbed, we implemented two different algorithms alongside the snapshot logic in the switch data plane ASIC: ECMP [16] and flowlet switching [20].

Workload. We used three distributed applications in our testbed. The first is Hadoop running a Terasort [4] benchmark workload with 5B rows of data. Our Hadoop instance ran version 2.9.0 with YARN [5] on 10 mappers and 8 reducers. The second is Spark's GraphX [7] running a PageRank [6] synthetic benchmark workload with 100,000 vertices. Our Spark instance ran version of 2.2.1 with Yarn on 5 servers. Finally, we implemented memcache [3], running an `mc-crusher` 50-key multi-get workload [13]. We populated the Hadoop and memcache instances with data during a setup phase that was not measured.

Counters. We implemented a variety of performance counters including per-port packet and byte counters along with queue depth measurements. However, in this section we primarily focus on an exponentially-weighted moving average (EWMA) of packet interarrival time. The EWMA counter was implemented in two phases due to hardware limitations on register computation:

```
interarrival = pkt_timestamp - last_ts[port]
last_ts[port] = pkt_timestamp
if packet_count[port] is even:
    temp_ewma[port] += interarrival
else:
    temp_ewma[port] /= 2
    ewma[port] /= temp_ewma[port]
```

Underlined variables are implemented with stateful registers. The EWMA updates on every other packet with the average

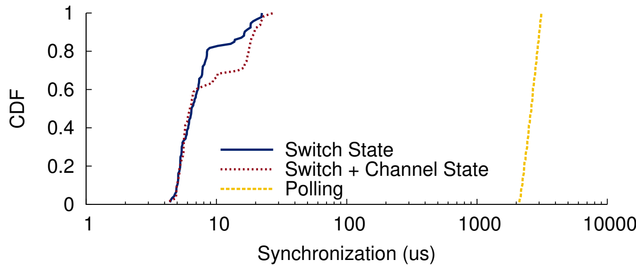


Figure 9: Synchronization of network-wide measurements using snapshots and traditional polling.

interarrival of the last two packets. As shown in the code, our implementation is functionally equivalent to an EWMA with a decay factor of .5.

8.1 Synchronization of Network Snapshots

We begin by evaluating the synchronization properties of Speedlight. For this, we configured processing units to tag snapshot notifications with the current timestamp. Recall that notifications are sent on any update of either the local snapshot ID or the last seen array, i.e., on any progress in the algorithm. In the experiment, we sent a command to each of the four virtual control planes in our testbed to schedule a snapshot. At the scheduled time, they sent initiations to every processing unit (ingress and egress) under their control as described in Section 6. *Synchronization* of a snapshot ID is defined as the difference between the earliest and latest timestamps on any notification with that ID.

Figure 9 shows a CDF of synchronization for three different approaches: (1) traditional counter polling, (2) Speedlight w/o channel state, and (3) Speedlight w/ channel state. In both configurations of Speedlight, median synchronization was ~6.4 µs. The maximum synchronization delta we observed was 22 µs w/o channel state, and 27 µs w/ channel state, likely due to randomness in PTP, queuing, and scheduling. These values are well-within a single RTT for most networks. As one might expect, channel state synchronization has a longer tail as completion depends on all upstream neighbors advancing to the current snapshot.

For comparison, we also measured the synchronization of a typical counter polling framework where an observer polls the statistic for each port individually via a control plane agent that reads and returns the value on-demand. For a full sequence of network-wide measurements, the median difference between the first and last poll was 2.6 ms.

8.2 Scalability of Speedlight

We also evaluate how Speedlight scales with the size and complexity of the network. In particular, we ask two questions: (1) how does the scale of the network affect the *frequency* with which Speedlight can take snapshots, and (2) how does the scale affect the *time synchronization* of those snapshots. Storage scalability was briefly addressed in Section 7.1.

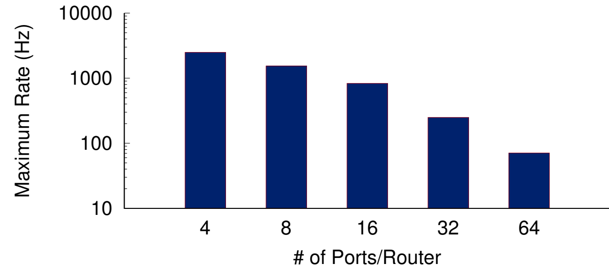


Figure 10: Max. sustained snapshot rate before notification queue buildup. Results are shown for a range of router port counts and assume no channel state.

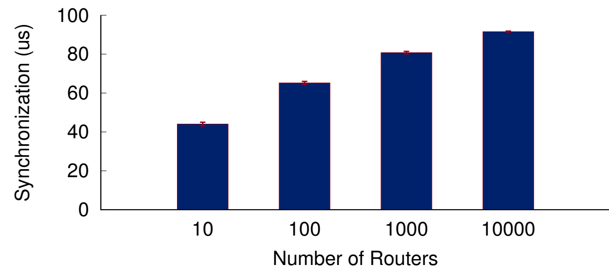
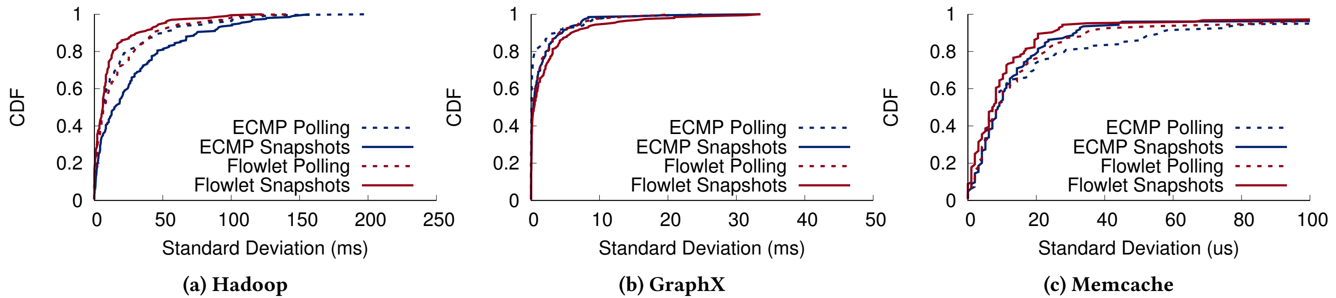


Figure 11: Average synchronization of Speedlight snapshots in larger network deployments. The snapshot assumes 64-port routers and no channel state.

Speedlight's architecture lends itself well to scalability; control planes are responsible for their own switch, and each processing unit has at most one external neighbor regardless of how many routers are added to the network. Instead, the primary factor in performance is number of ports per router.

Figure 10 shows the maximum sustained snapshot frequency versus router port count. In the experiment, we initiated a series of snapshots on a single switch with fixed interval. Snapshot frequencies that were too high eventually resulted in notification drops. The graphs plot the highest frequency without drops. Even for 64 ports (a full linecard), Speedlight can sustain over 70 snapshots per second. Note that the ASIC-CPU channel is more than sufficient; rather, the bottleneck is in our unoptimized control plane processing latency. Thus, Speedlight supports bursts of higher frequency snapshots given a sufficiently large socket receive buffer.

Network size primarily affects Speedlight's synchronization. Figure 11 shows average whole-network synchronization for several large simulated networks. Our simulation included PTP time drift, OpenNetworkLinux scheduling effects, and the latency between initiation and data plane snapshot execution. Distributions for all of these values were collected from our hardware testbed. While Speedlight's multi-initiator design limits time drift, additional routers



(a) Hadoop **(b) GraphX** **(c) Memcache**

Figure 12: Standard deviation of uplink load balancing in our leaf-spine topology. We compared two approaches: flow-based ECMP and flowlet load balancing. We tested Hadoop, GraphX, and memcache as well as polling versus snapshots. Note the difference in units on the x-axis.

and ports can make encountering tail effects more likely; however, this effect is asymptotic and still stays under typical RTTs.

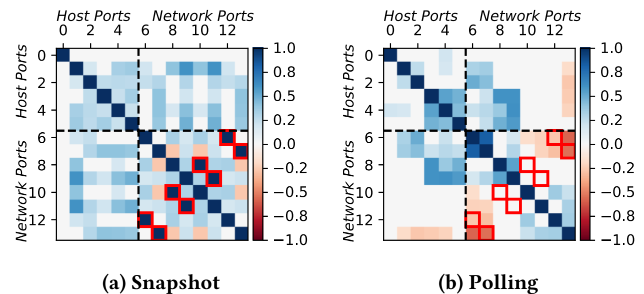
8.3 Use Case: Evaluating Load Balancing

We began this paper with a running example of a question an operator might want to ask about a network: how well is my load balancing protocol working? We demonstrate Speedlight's ability to answer this question by comparing the performance characteristics of ECMP and Flowlet load balancing algorithms in the presence of Hadoop, GraphX, and memcache. In theory, Flowlet forwarding should balance load more fairly because it splits traffic at a finer granularity [20]. In practice, our understanding of the impact of flowlets on load balance is limited to average utilization, drop rate, flow completion time, and other carefully crafted proxies for the property in which we are actually interested.

In this experiment, we took a series of snapshots, and computed the standard deviation of the EWMA of packet interarrival times across uplink ports. To account for workload deviations, uplinks were compared only to other uplinks on the same switch. Figure 12 shows CDFs of the standard deviations for our Hadoop, GraphX, and memcache workloads taken with both snapshots and traditional polling. The three workloads showcase three different behaviors.

For Hadoop, polling shows little-to-no gain for flowets, when in reality flowlets improve balance significantly. For GraphX, polling consistently underestimates the imbalance in the network. Our Memcache workload is very evenly distributed, but exhibits the opposite behavior—polling consistently overestimates the imbalance.

Together, these experiments illustrate an important point. For measures of whole-network behavior, the issue is not just that polling might provide an incorrect view of the network, but that it is difficult to place a bound on the inaccuracy.



(a) Snapshot **(b) Polling**

Figure 13: Pairwise correlation coefficients for egress ports while running GraphX. The red boxes highlight port pairs on the same ECMP paths, which are expected to have high positive correlations.

8.4 Use Case: Synchronized Traffic

The second use case we target is the detection of synchronized application traffic patterns for understanding behavior or debugging performance issues. For this experiment, we measured EWMA of packet rates at egress of all ports, in 100 snapshots taken 1 second apart. We then calculated pairwise correlation between ports using Spearman [12] tests.

Figure 13 shows the statistically significant ($\rho < 0.1$) correlation coefficients found while running GraphX. With snapshots, the Spearman test found correlations for 43% more of the port pairs. To validate correctness, we analyzed the output for evidence of two ground truths related to the application and network topology. First, we expected to see no significant correlations between the port egressing to the master server (server 0) and any other port because the master server did not participate in the distributed computation. Second, we expected to find high correlations between possible ECMP next-hops.

With snapshots, the correlation coefficients matched both expected ground truths. Polling, on the other hand, failed to identify the positive correlations between ECMP ports. As shown by the red boxes in Figure 13, the correlations found with polling were either statistically insignificant or, worse,

statistically significant but *negative*. Results were qualitatively similar for other applications and ρ values.

9 RELATED WORK

Network measurement is a well-studied field, with many proposals for better and more expressive measurement tools [15, 22, 31, 42]. As networks grow, it becomes even more important to have good monitoring and debugging tools. Our work is, to the best of our knowledge, the first to demonstrate practical, synchronous, and consistent network-wide measurement. A large body of prior work has tackled related goals and solutions. We discuss that work below.

Hardware-assisted measurement. With the recent rise of programmable data and control planes, there has been increased interest in novel measurement applications [28, 29, 31, 32, 37, 40]. Thus far, these approaches have concentrated on exploring the limits of what can be feasibly collected. Together, they are a testament to the expressiveness and utility of programmable switches. Our work is complementary—network snapshots can be of *any* local state, including the statistics generated by these systems.

Multi-device measurement. One method to move beyond single-component measurement is to leverage traffic to capture relevant state as it traverses the network [1, 2, 15, 22]. For example, packets could record the minimum queue depth at any intermediate switch. These techniques have the advantage of enforcing causal consistency at the level of an single sample; however, like single component measurement, it is still difficult to compare across samples and paths.

Measurement aggregation. Another approach for trying to understand network-wide behavior is to take measurements of individual devices or paths and build larger insights on top of their aggregates. There are too many such approaches to cover here, but these largely rely on statistics, thresholds, and similar techniques. Network tomography [11, 21, 26, 30] is a common example that uses statistics to tease out interesting behavior from long-term traces of multiple devices. While this class of approaches can assist in a variety of use cases, they lack the granularity to answer the types of questions addressed in the preceding section.

Distributed snapshots. The literature on distributed snapshot algorithms is similarly rich. The original paper on the topic [10] inspired a wide variety of improvements and refinements. Of particular note are piggybacking-based protocols like [24, 27]. Originally designed to allow for non-FIFO channels, we borrow their techniques for handling packet drops, but prohibit out-of-order delivery for efficiency. Finally, we note that others have discussed the practicality of distributed snapshots in networks [18, 34], but in the control plane rather than the data plane.

10 DISCUSSION

Measuring Forwarding State. In Section 2.2, we remarked that it may be useful to snapshot forwarding state. While ASIC data planes are not typically able to record table entries directly, they *can* record version information. Specifically, the control plane can ensure every FIB rule and version tags passing packets with a unique ID that is then stored back into processing unit state. A snapshot of the state would then give hints as to the entire network's forwarding state.

Partial Deployment. Speedlight is amenable to partial deployment. In this case, the snapshot would be of participating devices and the communication channels between them. For instance, in a data center, an operator might want for only ToR switches or a particular cluster to be snapshot-enabled.

For snapshots without channel state, the only requirement is that the snapshot header is appended and removed at the proper time. The simplest method is to append the header whenever an ingress processing unit encounters a packet without one, and configure the remaining hosts to ignore IP options in which the snapshot header is contained. If that is not possible (e.g., due to security concerns with IP options), the header should be removed at the last snapshot-enabled device in the packet's path. Causal consistency is maintained even when there are multiple paths between devices.

Snapshots with channel state are slightly more complex. In order to gather channel state, devices must be able to reduce communication to FIFO channels. More specifically, devices must tag packets with the physical path they take between snapshot-enabled devices. We note that in the case of data centers and snapshot-enabled ToRs, this requires only minor modifications to the configuration of existing devices [33].

11 CONCLUSION

The technique described in this paper, Synchronized Network Snapshots, and its realization, Speedlight, provide unprecedented visibility into the behavior of the network as a whole. Whether for evaluating a design, diagnosing an issue, or simply trying to understand an existing network, these techniques help to answer critical questions. We demonstrate that this approach is practical by implementing and deploying on a testbed a working version of our system, then using it to collect interesting measurements of real workloads.

ACKNOWLEDGEMENTS

We gratefully acknowledge Sameera Gajjarapu, our shepherd Aditya Akella, and the anonymous SIGCOMM reviewers for all of their help and thoughtful comments.

REFERENCES

[1] Aijay Adams, Petr Lapukhov, and Hongyi Zeng. 2016. NetNORAD: Troubleshooting networks via end-to-end probing. (2016). https://code.facebook.com/posts/1534350660228025/netnorad-troubleshooting-networks-via-end-to-end-probing/.

[2] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. ACM, New York, NY, USA, 503–514. https://doi.org/10.1145/2619239.2626316

[3] Dormando Anatoly Vorobey, Brad Fitzpatrick. 2009. Memcached. (2009). https://memcached.org

[4] Apache Software Foundation. 2012. Hadoop, Terasort. (2012). https://hadoop.apache.org/docs/r2.7.1/api/org/apache/hadoop/examples/terasort/package-summary.html

[5] Apache Software Foundation. 2012. Hadoop, YARN. (2012). https://hadoop.apache.org/docs/r2.7.0/

[6] Apache Software Foundation. 2014. PageRank, GraphX. (2014). https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/graphx/SynthBenchmark.scala

[7] Apache Software Foundation. 2016. Spark. (2016). https://github.com/apache/spark/

[8] Barefoot. 2017. Barefoot Tofino. https://www.barefootnetworks.com/technology/. (2017).

[9] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*. ACM, New York, NY, USA, 99–110. https://doi.org/10.1145/2486001.2486011

[10] K Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)* 3, 1 (1985), 63–75.

[11] Yan Chen, David Bindel, Hanhee Song, and Randy H. Katz. 2004. An Algebraic Approach to Practical and Scalable Overlay Network Monitoring. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '04)*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1015467.1015475

[12] Christophe Croux and Catherine Dehon. 2010. Influence Functions of the Spearman and Kendall Correlation Measures. *Statistical methods & applications* 19, 4 (2010), 497–515.

[13] Dormando. 2016. mc-crusher. (2016). https://github.com/memcached/mc-crusher

[14] Glen Gibb, George Varghese, Mark Horowitz, and Nick McKeown. 2013. Design Principles for Packet Parsers. In *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*. IEEE, Washington, D.C., USA, 13–24.

[15] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. 2015. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, New York, NY, USA, 139–152. https://doi.org/10.1145/2785956.2787496

[16] C. Hopps. 2000. *Analysis of an Equal-Cost Multi-Path Algorithm*. RFC 2992. RFC Editor. 1–8 pages. https://tools.ietf.org/html/rfc2992

[17] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. 2017. Tagger: Practical PFC Deadlock Prevention in Data Center Networks. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '17)*. ACM, New York, NY, USA, 451–463. https://doi.org/10.1145/3143361.3143382

[18] John P. John, Ethan Katz-Bassett, Arvind Krishnamurthy, Thomas Anderson, and Arun Venkataramani. 2008. Consensus Routing: The Internet as a Distributed System. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI '08)*. USENIX Association, Berkeley, CA, USA, 351–364.

[19] Prem Jonnalagadda. 2017. Disaggregation and Programmable Forwarding Planes. https://barefootnetworks.com/blog/disaggregation-and-programmable-forwarding-planes/. (2017).

[20] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur W. Berger. 2007. Dynamic Load Balancing Without Packet Reordering. *Computer Communication Review* 37, 2 (2007), 51–62. https://doi.org/10.1145/1232919.1232925

[21] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. 2009. Detailed Diagnosis in Enterprise Networks. *ACM SIGCOMM Computer Communication Review* 39, 4 (2009), 243–254.

[22] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. 2015. In-band Network Telemetry via Programmable Dataplanes. In *Demo paper at SIGCOMM '15*.

[23] Ajay D Kshemkalyani, Michel Raynal, and Mukesh Singhal. 1995. An introduction to snapshot algorithms in distributed computing. *Distributed systems engineering* 2, 4 (1995), 224.

[24] Ten H Lai and Tao H Yang. 1987. On distributed snapshots. *Inform. Process. Lett.* 25, 3 (1987), 153–158.

[25] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. 2016. Globally Synchronized Time via Datacenter Networks. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 454–467. https://doi.org/10.1145/2934872.2934885

[26] Ma łgorzata Steinder and Adarshpal S Sethi. 2004. A survey of fault localization techniques in computer networks. *Science of computer programming* 53, 2 (2004), 165–194.

[27] Hon Fung Li, Thiruvengadam Radhakrishnan, and K. Venkatesh. 1987. Global State Detection in Non-FIFO Networks. In *International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, Washington, D.C., USA, 364–370.

[28] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI '16)*. USENIX Association, Berkeley, CA, USA, 311–324.

[29] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 101–114. https://doi.org/10.1145/2934872.2934906

[30] Radhika Niranjan Mysore, Ratul Mahajan, Amin Vahdat, and George Varghese. 2014. Gestalt: Fast, Unified Fault Localization for Networked Systems. In *USENIX ATC*. USENIX Association, Philadelphia, PA, 255–267. https://www.usenix.org/conference/atc14/technical-sessions/presentation/mysore

[31] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 85–98.

[32] Remi Philippe. 2016. *Next Generation Data Center Flow Telemetry*. Technical Report. Cisco.

[33] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C. Snoeren. 2017. Passive Realtime Datacenter Fault Detection and Localization. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI '17)*. USENIX Association, Berkeley, CA, USA, 595–612.

[34] Liron Schiff, Michael Borokhovich, and Stefan Schmid. 2014. Reclaiming the Brain: Useful OpenFlow Functions in the Data Plane. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets-XIII)*. ACM, New York, NY, USA, 7:1–7:7. https://doi.org/10.1145/2670518.2673874

[35] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Changhoon Kim, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. 2017. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI '17)*. USENIX Association, Berkeley, CA, USA, 67–82.

[36] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 15–28.

[37] John Sonchack, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. 2018. Turboflow: Information Rich Flow Record Generation on Commodity Switches. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, Article 11, 16 pages. https://doi.org/10.1145/3190508.3190558

[38] Madalene Spezialetti and Phil Kearns. 1986. Efficient Distributed Snapshots. In *International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, Washington, D.C., USA, 382–388.

[39] Niels LM Van Adrichem, Christian Doerr, and Fernando A Kuipers. 2014. Opennetmon: Network monitoring in OpenFlow software-defined networks. In *Network Operations and Management Symposium (NOMS)*. IEEE, Washington, D.C., USA, 1–8.

[40] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI '13)*. USENIX Association, Berkeley, CA, USA, 29–42.

[41] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. 2017. High-resolution Measurement of Data Center Microbursts. In *Proceedings of the 2017 Internet Measurement Conference (IMC '17)*. ACM, New York, NY, USA, 78–85. https://doi.org/10.1145/3131365.3131375

[42] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. 2015. Packet-Level Telemetry in Large Datacenter Networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, New York, NY, USA, 479–491. https://doi.org/10.1145/2785956.2787483