RC25444 (AUS1401-001) January 30, 2014
Computer Science

IBM Research Report

OpenSample: A Low-Latency, Sampling-Based Measurement Platform for SDN

Junho Suh, Ted "Taekyoung" Kwon
Seoul National University
Seoul, South Korea

Colin Dixon, Wes Felter, John Carter
IBM Research Division
Austin Research Laboratory
11501 Burnet Road
Austin, TX 78758
USA

Research Division
Almaden – Austin – Beijing – Cambridge – Dublin - Haifa – India – Melbourne - T.J. Watson – Tokyo - Zurich

OpenSample: A Low-latency, Sampling-based Measurement Platform for SDN

Junho Suh Ted "Taekyoung" Kwon
Seoul National University
Seoul, Korea
jhsuh,tk@mmlab.snu.ac.kr

Colin Dixon Wes Felter John Carter
IBM Research
Austin, TX
ckd,wmf,retrac@us.ibm.com

***Abstract*—In this paper we propose, implement and evaluate OpenSample: a low-latency, sampling-based network measurement platform targeted at building faster control loops for software-defined networks. OpenSample leverages sFlow packet sampling to provide near–real-time measurements of both network load and individual flows. While OpenSample is useful in any context, it is particularly useful in an SDN environment where a network controller can quickly take action based on the data it provides. Using sampling for network monitoring allows OpenSample to have a 100 millisecond control loop rather than the 1–5 second control loop of prior polling-based approaches. We implement OpenSample in the Floodlight OpenFlow controller and evaluate it both in simulation and on a testbed comprised of commodity switches. When used to inform traffic engineering, OpenSample provides up to a 150% throughput improvement over both static equal-cost multi-path routing and a polling-based solution with a one second control loop.**

I. Introduction

Software-defined networking (SDN) replaces the distributed, per-switch control planes of traditional networks with a (logically) centralized control plane that programs the forwarding behavior of all the switches in a given network. This centralized control plane, run on a controller, can act as a control loop that (i) gathers traffic and other measurements from the network and (ii) uses the gathered information to compute and install forwarding behaviors in the switches. Although there are two logical components to this control loop—measurement and control—the focus of the vast majority of SDN research has been on control. Some prior SDN research has included a measurement component [1], [2], [3], but the closest related work also notes this bias toward control [4].

OpenFlow [5], the dominant protocol used to implement SDN, provides two measurement techniques to create a global view of the network: `packet_in` messages and per-port/per-rule counters. Typically when a packet matches no switch rule, the switch sends a `packet_in` message containing the packet header (and possibly payload) to the controller for handling, which may include installing new rules. Thus, in a typical setup, the controller receives one `packet_in` message[1] at the beginning of each flow. In addition, switches maintain counters to track the number of packets and bytes handled by each port and each OpenFlow rule.

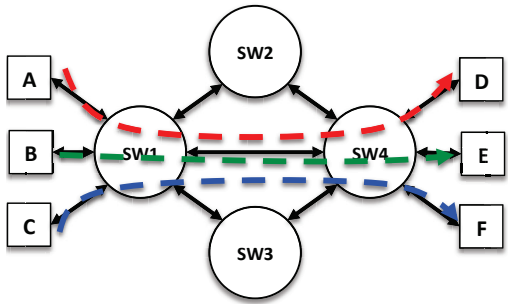
[1]It is possible to receive more than one `packet_in` message per flow especially if the flow is not connection oriented and thus sends many packets before hearing anything from the receiver.

In practice, neither of these measurement mechanisms enable a scalable, low-latency measurement system. The `packet_in` messages place a high burden on the local switch CPU and are typically limited to at most a few hundred per second [6], [7]. Further, they only provide data when a new flow appears or a rule expires. Allowing a rule to expire typically causes the flow to be paused until the `packet_in` message is delivered to the controller and a new rule installed, which typically takes a few 10s of milliseconds. In the wide-area this might be tolerable, but in data centers and other local-area networks this is long enough to cause TCP timeouts and thus likely unacceptable.

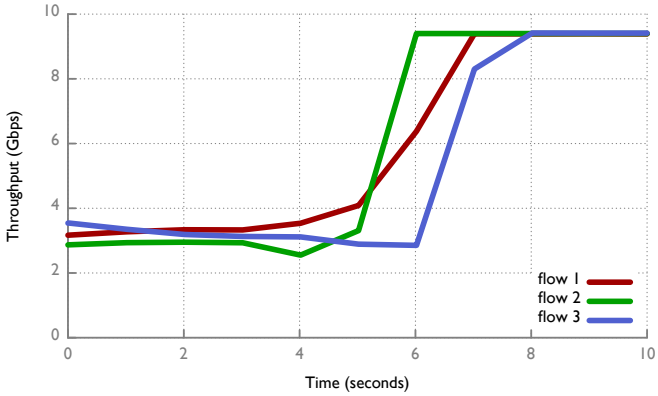
On the other hand, port and flow counters are typically only updated every second or so [2], which limits the control loop of a controller to operating at a speed that is too slow to catch any but the largest flows [8]. Further, the space-granularity of counters are either per-port or per-rule. Per-port counters do not provide flow-level information, which either limits visibility into the network or requires more processing, i.e., tomography, to (probabilistically) disaggregate port-level information into flow-level tracking. Per-rule counters require that rules be installed at the granularity of the desired measurement, which couples forwarding and measurement, with both mechanisms installing rules in the switch. For example, Frenetic [3] is forced to break apart OpenFlow rules when the monitoring requests do not directly correspond to the forwarding rules. This consumes even more of the already scarce switch TCAM space [6].

Ideally, a measurement system for software-defined networks would provide global visibility into the network and near–real-time data, i.e., latencies on the order of milliseconds. Further, it would scale to large traffic volumes without overtaxing switch control CPUs or the SDN controller.

In this paper we present the design and evaluation of a sampling-based SDN measurement system called *OpenSample* that achieves these goals. Rather than using the OpenFlow measurement mechanisms, OpenSample leverages the sFlow packet sampling functionality present in most switches. sFlow supports uniform random sampling of packets on a per-port basis. The switch forwards the header of, on average, 1 in every N packets traversing a given port to a collector. From these random samples, the collector can infer a variety of information about the network including link utilization and



(a) The physical OpenSample testbed with four 10 Gbps IBM G8264 switches configured with 6 hosts and 3 large flows.



(b) The throughput of three large flows both before and after enabling OpenSample-based traffic engineering.

Fig. 1. An intuitive example of OpenSample-based traffic engineering running on a physical testbed.

the largest flows present on each link. Further, since each switch that a packet traverses can sample that packet, the effective sampling rate of a network grows as the network grows and paths get longer.

To demonstrate the value of OpenSample's faster network monitoring, we implemented a traffic engineering application that uses OpenSample to detect congested links and the large flows using those links. In our experiments, traffic engineering informed by OpenSample provides up to 150% more aggregate throughput than both equal-cost multi-path (ECMP) routing and polling-based traffic engineering (inspired by Hedera [1]) when flow sizes are small, i.e., an average of one megabyte. Further, OpenSample can be implemented without the end-host modifications required by Mahout [2] and MicroTE [8], and does not require the use of expensive OpenFlow rules for fine-grained measurement like Frenetic [3].

The following example, illustrated in Fig. 1(a), motivates and explains OpenSample-based traffic engineering, We configure a physical testbed with four IBM RackSwitch G8264 10GbE switches in a clique with three hosts attached to two of the switches. Hosts A, B, and C each generate one flow to hosts D, E, and F, respectively, using iperf to saturate the available throughput. Fig. 1(b) shows the throughput of each of the three flows both before and after we turn on traffic engineering. Initially, the three flows compete for bandwidth on the single shortest path (SW1-SW4) and converge to a fair share of approximately 3 Gbps each. However, there are sufficient redundant paths in the topology for each flow to follow its own disjoint path. After five seconds we enable traffic engineering, which is able to identify the three elephant flows and re-route two of them to uncongested paths. After that, each flow achieves 10 Gbps of throughput using the three different disjoint paths: 1-4, 1-2-4 and 1-3-4.

The remainder of the paper is organized as follows. Section II provides a more detailed look at sampling-based measurement as well as data center traffic characterizations. We describe the design and implementation of OpenSample in Section III. Section IV presents an evaluation of OpenSample using both emulation and results from a real testbed. We cover the work most closely related to OpenSample in Section V. Finally, in Section VI we draw conclusions and present ideas for future work.

II. Background

Before diving into the design, implementation and evaluation of OpenSample, we first present background material on both typical data center workloads and existing non-OpenFlow network monitoring approaches, which we use to drive the design of our traffic engineering application.

A. Data Center Network Workloads and Topologies

While the purpose of this paper is not to characterize modern data center workloads or topologies, we note certain properties in current data centers that others have observed and use these properties in our construction and evaluation of OpenSample. In particular, several characterizations of data center workloads [9], [10] have shown that while there are often *hot spots* in data center networks, the remainder of the networks is typically underutilized. Thus, if the topology has multiple paths for any given flow to select, it is likely that traffic experiencing congestion could be re-routed to an under-utilized path.

Fortunately, common current and future data center network topologies, e.g., Fat Tree [11], HyperX [12], and Jellyfish [13], offer many diverse paths between arbitrary endpoints. Despite this, most current data centers still use static routing configurations and/or equal-cost multi-pathing (ECMP). While these approaches attempt to minimize the occurrence of hot spots, hot spots still occur and these static approaches can do nothing in response to congestion once it occurs.

Recent research efforts [1], [6], [8] use SDN to reroute flows in reaction to congestion. The result is a variety of good techniques for selecting alternate paths. However, these approaches rely on switch-based measurements with latencies measured in seconds, and are thus limited to detecting and rerouting only the largest flows.

Thus, all of the ingredients for substantially improved traffic engineering are present, except for the timely network measurements OpenSample provides.

Further, we note that the traffic characterization indicates that flow sizes are approximately exponentially distributed and

flow inter-arrival times are also exponentially distributed, i.e., a poisson process. As a consequence, in our emulation results, we assume exponential flow sizes and inter-arrival times.

B. Non-OpenFlow Network Monitoring

While many readers are no doubt familiar with the network monitoring features of OpenFlow, i.e., per-port and per-rule byte and packet counters, they may be less familiar with other monitoring techniques such as NetFlow [14] and sFlow [15]. NetFlow produces per-flow statistics without requiring rules to be installed. sFlow provides real-time packet samples from individual switches. Mann et al. [16] recently compared NetFlow and sFlow for network monitoring at the hypervisor in virtualized data centers. Since congestion can occur anywhere in the network, we monitor both physical and virtual switches. And because the overhead of monitoring impacts its value, we are concerned with the overhead of monitoring techniques.

1) NetFlow: NetFlow [14] was originally developed by Cisco to provide a way to collect statistics about individual IP flows in a data network. In NetFlow, each switch (or router) maintains a flow cache that tracks flow statistics for each flow, usually identified by 5-tuple (source and destination IP address, source and destination TCP/UDP port, and IP protocol number) and type of service. As each packet arrives, its header fields are checked to see if it matches an existing entry in the flow cache. If it does, then the flow cache entry is updated appropriately, i.e., by incrementing the packet and byte counts. If the flow is not already present in the flow cache, a new entry in the flow cache is created. NetFlow has four policies to decide when to send the flow record to a NetFlow collector: (i) when a TCP packet is seen with a FIN or RST flag indicating flow completion, (ii) when a flow idle timeout expires, (iii) when a hard timeout fires indicating that the flow has been tracked for y seconds regardless of whether it is still sending traffic, and (iv) when the flow cache is full and an entry must be evicted. When any of these four conditions hold, the switch sends a NetFlow record including flow statistics to a collector for further analysis.

Implementing NetFlow in hardware requires a dedicated CAM to track this information at line-rate. This hardware is not found in all switches and support for NetFlow is chiefly found in Cisco products and hypervisor vSwitches such as VMware ESX and Open vSwitch.

Further, NetFlow timeouts are specified at second granularity and in practice many implementations do not allow for values less than 30 seconds, so it provides little latency advantage over the low polling rates achievable using OpenFlow's per-rule counters. Since OpenSample is focused on low-latency network measurements, NetFlow is not a suitable choice for OpenSample. However, we note that NetFlow does not require TCAM rules to implement, which simplifies TCAM table management and decouples monitoring from control.

It should be noted that later versions of NetFlow also include a "Sampled NetFlow" [17] mode that produces NetFlow records based on sampling 1 in N packets that traverse a switch rather than every packet. However, the samples are still applied to the records in the flow cache and records are still sent according to the same policy. Thus Sampled NetFlow incurs the same coarse-grained timeouts that make NetFlow unsuitable for low-latency monitoring.

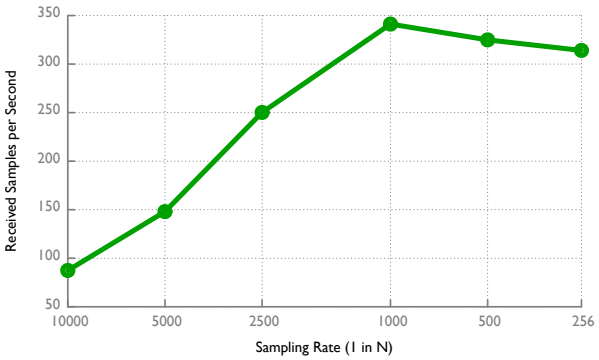


Fig. 2. The number of samples per second received at the collector as the sampling rate increases.

2) sFlow: The sFlow [15] standard aims to provide fine-grained network measurements without requiring per-flow state at switches. Instead it relies on two forms of sampling: *packet sampling* and *port counter sampling*.

For packet sampling, the switch captures one out of every N packets on each input port[2]. It then immediately forwards the sampled packet's header encapsulated with metadata to a central collector. The metadata includes the sampling rate, the switch ID, the timestamp at the time of capture, and forwarding information such as input and output port numbers.

The rate of samples produced by sFlow is not constant; it is equal to the packet rate on the port divided by the sampling rate. Since the packet rate varies dramatically based on network load and packet size, the rate of samples also varies. Note that a packet passing through multiple switches is eligible to be sampled by every switch along the path. If a flow passes through k switches, combining the samples from those switches gives an effective factor of k increase in the sampling rate.

From the gathered samples, the collector can probabilistically infer a number of flow statistics, e.g., it can estimate the number of packets and bytes in each flow by simply multiplying the number of sampled bytes and packets by the sample rate, N [18]. This approach is called *simple scaling* and is an unbiased estimator for the actual number of bytes and packets sent by the flow. In the remainder of this paper we refer to this technique for estimating the byte and packet counts of the flow as *Maximum Likelihood Estimation* (MLE).

This simple scaling approach has the limitation that it requires a large number of samples to provide accurate estimates of the true flow byte and packet counts. The expected relative error is inversely proportional to the square root of the number of samples, s, gathered from that flow. In particular, the

[2]In actuality N can be configured per-port and need not be the same for all ports. However, in OpenSample, we assume the sampling rate is fixed for all ports and leave exploration of per-port sampling rates to future work.

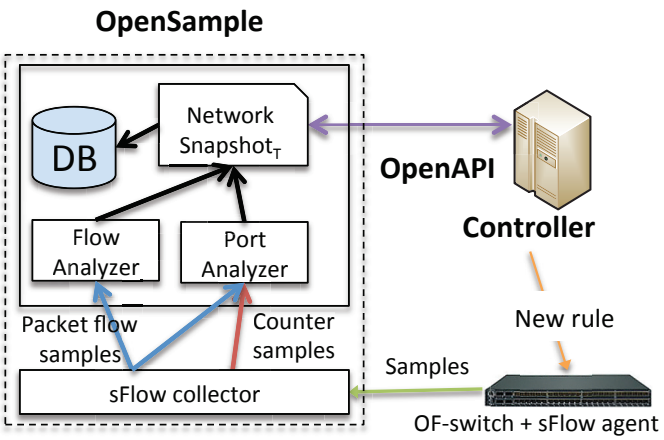


Fig. 3. The architecture of OpenSample providing measurement data to an SDN controller is depicted. sFlow agents running on switches provide samples to an sFlow collector which are then forwarded to flow and port analyzers. This information is aggregated into network snapshots and exposed via an open API. The SDN controller uses this API to make flow rerouting decisions.

expected error in percent can be estimated as:

$$\text{percent error} \leq 196 \cdot \sqrt{\frac{1}{s}}$$

An analysis of real data center workloads by Benson et al., [10] found that an average of 3,000 packets and 60 flows arrive at each top-of-rack switch in any given 100 ms window. This means the average flow has 50 packets in a 100 ms window. Even if all 50 packets from a given flow are sampled, we can only estimate the flow's actual rate with approximately 30% error. In practice with realistic sampling rates, even this is optimistic. Using the maximum likelihood estimation approach, there are only two ways to improve accuracy: (i) increase the sampling rate and/or (ii) increase the sampling period. The latter is not viable without violating OpenSample's goal of low-latency measurements.

Unfortunately, increasing the sampling rate is difficult as well. Fig. 2 shows the number of samples per second our sFlow collector receives from one of our testbed switches as we increase the sampling rate while keeping the amount of traffic going through the switch constant. The sample rate peaks at between 300 and 350 samples per second. We believe this limit is a consequence of the switch's control CPU being overwhelmed. With a limit of ∼350 samples per second, the expected number of samples for a given flow in a 100 ms time window that samples from 60 flows is less than one. While newer switches may provide faster control CPUs, it seems likely that it will be infeasible to get enough sFlow samples in a short period, i.e., 100 ms, to provide an accurate estimate of the flow throughput for the foreseeable future. Therefore, we need to do something other than simple scaling to estimate flow statistics accurately and in near-real-time, which is detailed in the next section.

III. OpenSample Design

The architecture of OpenSample is illustrated in Fig. 3. We use packet sampling, e.g., sFlow, to capture packet header samples from the network with low overhead and use TCP sequence numbers from the captured headers to reconstruct nearly-exact flow statistics. Simultaneously, we use the same packet samples to estimate port utilization at sub-second time scales, described in detail below. We use a single, centralized collector that combines samples from all switches in the network to construct a global view of traffic in the network at both flow and link granularities.

Detailed network monitoring information has a variety of uses, including traffic engineering, resource provisioning, VM placement/migration, and intrusion detection. For illustration purposes, we focus on traffic engineering as the consumer of OpenSample's ability to very quickly detect elephant flows and estimate link utilization. In the following subsections, we describe how OpenSample extracts flow statistics from the samples, detects elephant flows, estimates link utilization of each switch's ports, and enables traffic engineering.

A. Extracting Flow Statistics using Protocol-specific Information

As discussed earlier, accurately inferring flow statistics using maximum likelihood estimation requires many samples per flow. To overcome sFlow's sampling rate limitation, we exploit the fact that most traffic sent in data centers today is TCP traffic [19]. Each TCP packet carries a sequence number indicating the specific byte range the packet carries. Fortunately, when sFlow samples the header of TCP packets, this header also includes the TCP sequence numbers[3].

Thus, if we sample at least two distinct packets from a given TCP flow, we can compute an accurate measure of the flow's average rate during the sampling window by subtracting the two sequence numbers and dividing by the time between the samples.

Exploiting TCP information drastically increases estimation accuracy for any given sampling rate. This TCP-aware sFlow analysis is the key innovation OpenSample incorporates compared to prior sFlow monitoring frameworks. In the next section, we provide analytic and simulation-based analysis of the probability of sampling two different packets from a given flow for a given number of switches, sampling rate, and flow size. We also examine the expected time before receiving two samples from a flow for a variety or parameters.

Our OpenSample-based traffic engineering mechanism considers any TCP flow for which it receives two or more samples to be an elephant flow and all elephant flows are candidates for traffic engineering. Thus, it considers far more flows to be candidates for rerouting than prior work [1], [2], [8].

Finally, we note that this approach is not limited to TCP, but can be extended to any protocol that includes sequence

[3] The sFlow specification does not actually mention or require that samples include TCP sequence numbers, but it does specify that the preferred implementation should provide the raw packet header [20] and, in practice, both our physical switches and Open vSwitch [21] provide this information.

numbers in the header. Even if the sequence numbers represent packets and not bytes, as long as this is known a priori, the sequence numbers can be used to compute flow bandwidth rates substantially more accurately than maximum likelihood estimation. This is accomplished by multiplying the number of packets (inferred accurately from packet sequence numbers) by the average observed packet length.

B. Probability of Flow Rate Detection

To determine the probability of detecting a flow's rate using our enhanced TCP-based rate detection, we analytically calculate the probability of getting at least two different samples from a single switch and use a simple simulator to find the same probability across one or more switches. In both cases, we evaluate the probability under a variety of different flow sizes and sampling rates.

First, we develop an analytical model for a single switch. To ease exposition, we introduce the following variables:

$$\begin{aligned} n &= \text{number of packets in the flow} \\ p &= \text{sampling rate } (0 \leq p \leq 1) = \frac{1}{N} \\ k &= \text{number of switches} \end{aligned}$$

Since there is no risk of sampling the same packet twice at a single switch, the probability of getting two samples from the same flow at the same switch is the probability of getting two samples from the same flow. More formally:

$$\begin{aligned} Pr\{\text{2+ samples}\} &= 1 - Pr\{\text{get zero or 1 sample}\} \\ &= 1 - Pr\{\text{get no samples}\} - \\ &\quad Pr\{\text{get 1 sample}\} \\ &= 1 - \binom{n}{0}(1-p)^n - \binom{n}{1}p(1-p)^{n-1} \\ &= 1 - (1-p)^n - np(1-p)^{n-1} \end{aligned}$$

When considering the case with more than one switch, the analysis becomes more complex because it must account for the possibility of sampling the same packet twice at two different switches. However, for realistic numbers of switches, k, and sampling rates, p, the probability of sampling the same packet more than once at different switches is low enough that it effectively acts as the one switch model with a sampling rate of kp. More formally:

$$Pr\{\text{2+ samples}\} \approx 1 - (1-kp)^n - nkp(1-kp)^{n-1}$$

To avoid analytical inaccuracy due to this simplification, we use a simple simulator to estimate the probability of getting two distinct samples from k switches. Each point is the result of 1000 simulations and the error bounds are small enough that we omit them. The results from both our analysis and simulator appear in Fig. 4. Fig. 4(a) shows the results of both the analysis and simulator for a single switch at various sampling rates. The points are from the simulator and the lines are from the analysis. The two are almost identical, which provides validation for our model. Further, note that for sampling rates greater than 1 in 200, we are nearly guaranteed to get two distinct samples from flows with more than 1000 packets, i.e., 1.5 MB or more. In contrast, for a 1000-packet flow and a 1 in 200 sampling rate, MLE has an 87% estimated error.

Fig. 4(b) shows the simulation results for varying number of switches and sampling rates. This shows that even for low sampling rates, increasing the number of switches drastically improves the probability that we will get two distinct samples. It also confirms the intuitive approximation to the one switch model. The lines with half the sampling rate, but twice the switches closely follow each other, e.g., one switch with 1 in 5000 sampling produces the same result as two switches with 1 in 10000 sampling.

C. Flow Detection Delay

To determine how long it takes to acquire two samples from a given flow, we analytically calculate the expected value of delay, D, by sum of two random variables, X_1 and X_2, which are the inter-arrival times of the first and second packets that are sampled. If we model packet arrivals at a switch as a Poisson process with average packet inter-arrival rate λ, then sample arrivals are a Poisson process with an average inter-arrival rate of λp. Hence, X_1 and X_2 are *i.i.d.* exponential random variables with mean λp. Therefore, the expected delay of getting the first and second samples is the sum of the mean values of two random variables. Assuming a single switch, this is more formally stated as:

$$\begin{aligned} E[D] &= E[X_1 + X_2] \\ &= E[X_1] + E[X_2] = \frac{1}{\lambda p} + \frac{1}{\lambda p} = \frac{2}{\lambda p} \end{aligned}$$

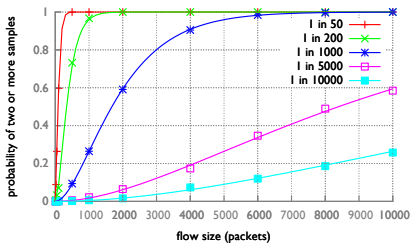
For the case with k switches, this is approximately $\frac{2}{\lambda kp}$.

Further, D, is an Erlang distributed [22] random variable with shape $\widehat{k} = 2$ and rate $\widehat{\lambda} = \lambda kp$. The cumulative distribution function for D is shown in Fig. 4(c). We present values for $k = 3$ representing a typical 3-hop path in data center and for packet inter-arrival rates of 1000 μs and 10 μs representing slow (12 Mbps) and fast (1.2 Gbps) flows (assuming 1500 byte packets). As can be seen, we easily detect all fast flows in less than 100 ms even with sampling rates of 1 in 1000.

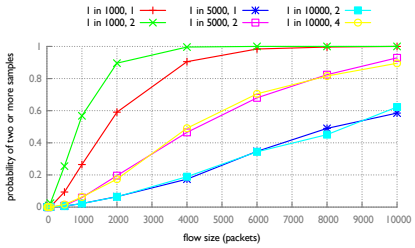
D. Estimating Switch Port Utilization

In addition to packet samples, sFlow reports exact packet and byte counter values for each port in a switch every 5 seconds, similar to OpenFlow. This data is not useful for our goal of sub-second monitoring.

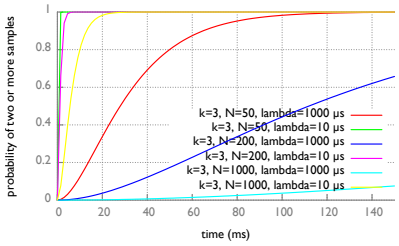
Thus we use packet samples to estimate link utilization at small timescales. OpenSample estimates the utilization of each link during a given interval by multiplying the number of sampled packets in that interval by the average packet size. This is akin to treating all packets going through a particular link as a single "superflow" and using MLE to estimate its throughput. This estimate is fairly accurate, despite MLE's poor error bounds, because it includes all samples from a given port.



(a) The analytical and simulated probability for one switch. The lines show the analytical values while the points show the simulated values.



(b) The simulated probability for one to four switches.



(c) The cumulative distribution function of the delay to get two packets from a flow based on packet inter-arrival rate lambda.

Fig. 4. The probability of getting at least two different samples from the same flow for varying flow sizes, sampling rates and number of switches. Each line is labeled "1 in N" for the single switch case or "1 in N, k" for multiple switches where the sampling rate is $\frac{1}{N}$ and there are k switches.

E. Network State Snapshot Database

Every 100 ms, OpenSample generates a snapshot of the network state for consumption by other applications. This state includes the network topology (retrieved from the SDN controller), estimated utilization of every switch port, and the set of detected elephant flows. For each elephant flow, the snapshot includes the flow's five-tuple, its estimated bandwidth, and its current path. Applications such as traffic engineering can query the latest snapshot through an API, making them loosely coupled with the internals of OpenSample.

F. Traffic Engineering

Traffic engineering is a natural application of software-defined networks because the controller has a global view of the topology and it controls all of the switches' forwarding tables. An SDN controller can choose to forward a flow over a non-shortest path just as easily as over a shortest path, with no concerns about convergence time, forwarding loops, or black holes. Unfortunately, the high latency of network measurements has limited how effective SDN-based traffic engineering can be.

Since we use exponentially distributed flow sizes, statistically a flow is expected to last as long as it has already lasted. Thus, if we detect an elephant flow that has sent a significant amount of traffic, we can expect it to send that much again in the near future. By moving such flows from congested to uncongested paths, that future traffic will achieve higher throughput, as will any traffic with which it was competing. Because only the packets sent after the flow is scheduled can benefit from traffic engineering, fast detection and scheduling are crucial to the efficiency of traffic engineering. Thus, OpenSample's low-latency monitoring is an ideal candidate for traffic engineering.

While we experimented with a variety of scheduling algorithms, we found that, in general, the speed at which the scheduling control loop could operate made a more significant difference than the algorithm used. As a consequence, we use the *global first fit* algorithm presented in Hedera [1] for its simplicity. Our technical innovations focus purely on improving the speed of congestion and large flow detection rather than improved flow scheduling techniques.

While our algorithm is borrowed from Hedera, it operates on a 100 ms interval—50 times faster than Hedera's five second interval. As data center networks are upgraded from 1 Gbps to 10 and 40 Gbps, we expect flow duration to shrink, which means that traffic engineering must become proportionately faster to remain effective. Alternately, with a constant link speed faster traffic engineering can make better decisions, as shown in our evaluation.

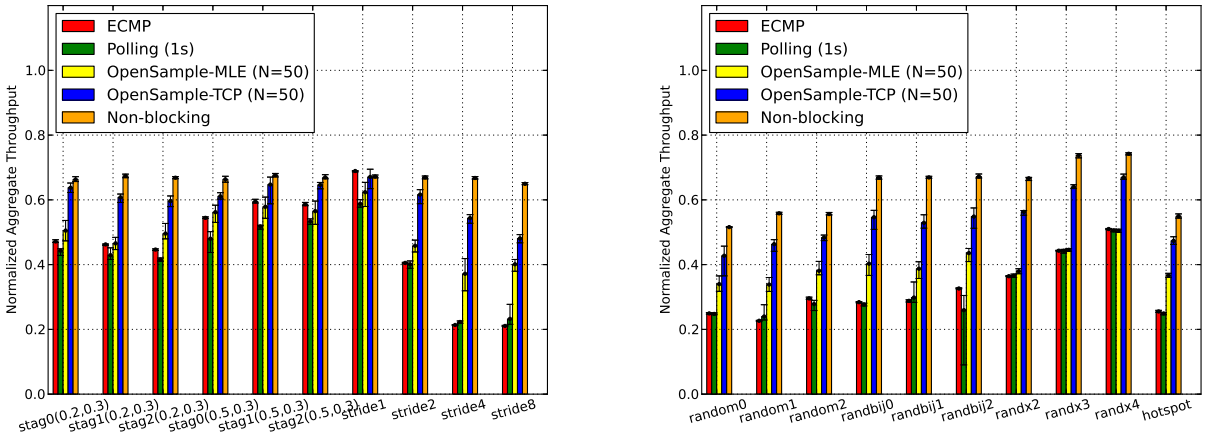
By default, all traffic in our network follows shortest paths—we do not use Spanning Tree Protocol or equivalent. When multiple shortest paths exist, ties are broken by using equal-cost multi-path (ECMP) hashing based on the TCP/IP 5-tuple. Every 100 ms interval, the controller estimates the utilization of every link in the network and attempts to reduce congestion by moving elephant flows from highly-utilized links to less utilized ones. The controller considers the set of detected elephant flows that traverse at least one congested link and uses a global first-fit algorithm to re-route them.

In OpenSample, both default forwarding and traffic engineering use OpenFlow. Engineered paths use high-priority OpenFlow rules and default paths use lower-priority rules. Scheduling a flow along a different path simply requires installing one new high-priority rule in each switch along the path. After an elephant flow ends, its rules time out and the switches automatically remove them. The time to install an OpenFlow rule—approximately 10 ms—is fairly small compared to OpenSample's 100 ms control interval.

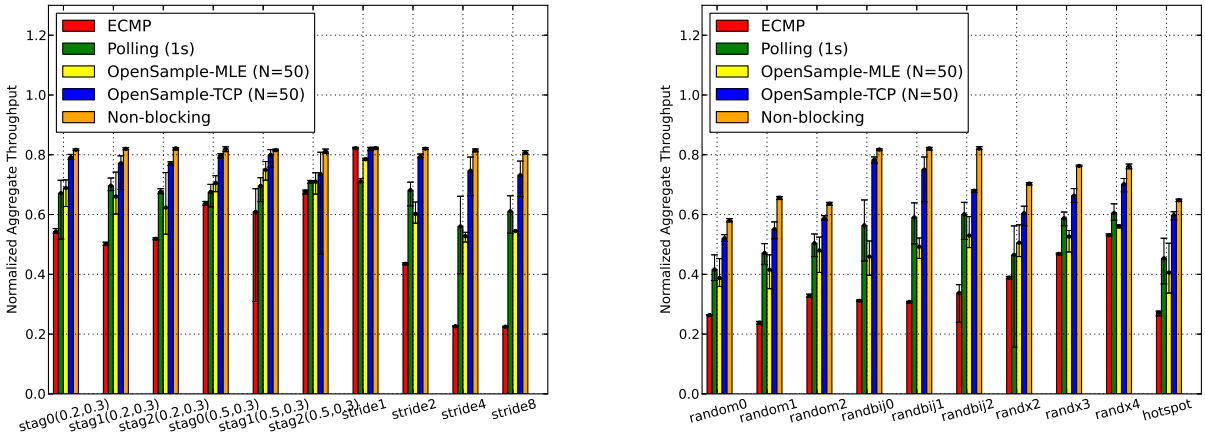
IV. Evaluation

In this section, we present the results of our experimental evaluation of OpenSample. Our goal is to compare OpenSample's fast control loop against previous counter-polling–based approaches. To this end, we implemented OpenSample and a traffic-engineering application as modules for Floodlight [23], an open-source OpenFlow controller written in Java. We tested OpenSample on both the Mininet-HiFi [24] emulator and a physical network testbed of x86 servers connected with IBM RackSwitch G8264 switches.

Because of our physical testbed's limited scale—only four switches—we predominantly used it to verify that our techniques work in practice, to validate our simulator framework, and to inform our design with the constraints of real-world



(a) Various traffic patterns with exponentially-distributed flow size with 1 MB average



(b) Various traffic patterns with exponentially-distributed flow size with 1 GB average

Fig. 5. Aggregate throughput for ECMP, Polling, OpenSample-MLE and OpenSample-TCP traffic engineering on a k=4 fat free compared to a single non-blocking switch all with 10 Mbps links. Each bar represents the average of 5 runs with error bars showing standard deviations. Flow inter-arrival times are exponentially distributed with a 1 ms average. OpenSample uses a sampling rate of N=50 and a 100 ms scheduling interval while polling uses a 1 s interval.

hardware. As a consequence, we omit the testbed results except for the simple demonstration of traffic engineering, shown in Fig. 1, and an evaluation of the sampling rates supported on our switches, shown in Fig. 2.

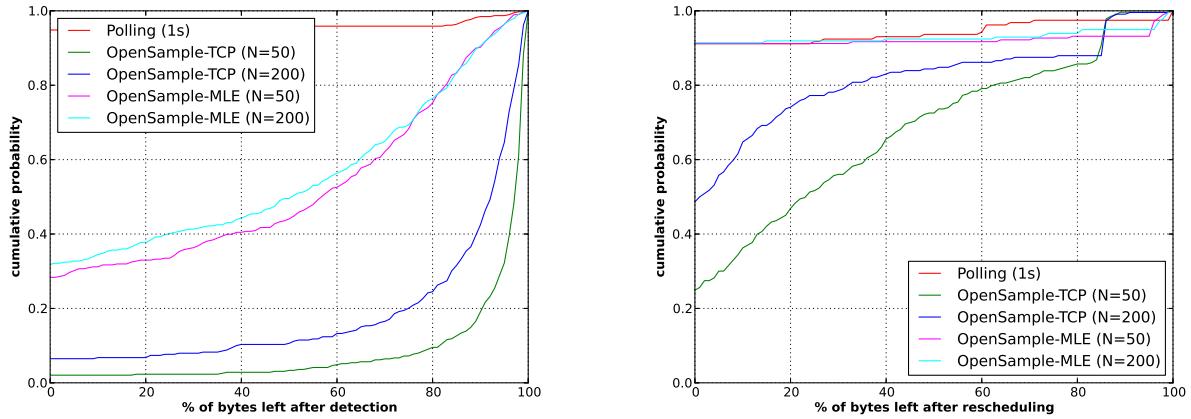
The remainder of this section focuses entirely on the emulation results, which provide an insight into OpenSample's operation at reasonable scales.

A. Methodology

We use the Mininet-HiFi [24] network emulator to evaluate OpenSample in a controlled and repeatable environment. Mininet uses Linux containers to emulate hosts and Open vSwitch (OVS) to emulate switches, allowing a whole network to be emulated on a single computer. Mininet-HiFi uses Linux traffic shaping to emulate fixed-speed links, giving the emulated network realistic congestion and queueing delays. We set the link speed to 10 Mbps to allow for faster emulation. However, note that OpenSample can control either Mininet or a physical network with no changes.

We replicate the testbed benchmark of Hedera [1] with identical settings, including topology and workloads. We use a three-level *k=4 FatTree* as an example of a network topology with a realistic diameter and degree of multipathing (a real network would use a much larger switch radix such as k=64). We also run the workloads on an emulated single large *non-blocking* switch to determine the maximum throughput when constrained only by host NIC speeds.

We evaluate OpenSample with two sampling rates: N=50 and N=200. Simple calculations indicate that to avoid exceeding the 350 sample per second limit on our physical switches, the sampling rate should be closer to N=250,000 to handle line-rate traffic on all ports. The bulk of the discrepancy comes from the 1000x difference between link speeds in our emulations and our physical switches. The remainder stems from the fact that (i) in practice, not all ports operate at line rate simultaneously under realistic workloads and (ii) newer



(a) CDF of bytes left at the time of detection

(b) CDF of bytes left at the time of rerouting

Fig. 6. The percent of bytes remaining in flows at the time of detection and time of rerouting in the *Stride8* workload with 1 MB average flow size.

switches [25] have significantly faster control CPUs allowing for more than 350 samples per second.

We use a workload generator [26] originally written for Hedera. We generate the traffic with the same spatial pattern such as *Stride(i)*, *Staggered Prob (EdgeP, PodP)*, and *Random(u)* to stress the network links, but use different parameters for flow statistics such as an average flow size and an average inter-arrival time. We generate two classes of flows: (i) short flows with a mean flow size of 1 MB and an exponential distribution and (ii) long flows the same exponential distribution but a 1 GB mean flow size. Both short and long flows follow the same inter-arrival time distribution, exponential with a mean of 1 ms.

For our performance baseline we use shortest-path ECMP forwarding with no traffic engineering. We compare three different measurement approaches: (i) Polling, (ii) OpenSample-MLE using a maximum likelihood estimator as described in Section II-B2, and (iii) OpenSample-TCP using TCP sequence numbers to infer throughput as described in Section III.

The Polling approach is designed to model measurement based on querying all of the flows in each switch as presented in prior work [1], [7], [27], [28], [29]. Based on their observations, we chose a polling rate of once per second as it matched the typical performance reported for the hardware switches evaluated. However we note that depending on the number of flows present in the switch and implementation details this latency varies from 75 ms to 15 s. We note that the systems which provided performance noticeably better than one second either used specialized interfaces to query counters or assumed that only a very small number of flows would be present.

To evaluate the performance of each approach, we measure the aggregate throughput achieved on the various spatial workload patterns after applying traffic engineering as described earlier. In all cases, we employ the same (Hedera) traffic engineering algorithm and change only the underlying measurement system. We also measure the total number of bytes sent and the fraction of those bytes that we were able to schedule on alternate routes for each case.

Technique	Total bytes sent	% of bytes scheduled
Polling (1s)	133 MB	25%
OpenSample-MLE (N=200)	166 MB	30%
OpenSample-MLE (N=50)	185 MB	36%
OpenSample-TCP (N=200)	226 MB	41%
OpenSample-TCP (N=50)	264 MB	62%

TABLE I
THE TOTAL BYTES SENT IN 30 S AND THE PERCENT OF THOSE BYTES SCHEDULED BY TRAFFIC ENGINEERING FOR THE *Stride8* WORKLOAD.

B. Results

Fig. 5 shows the throughput of a variety of workloads on our emulated configuration. Note that even the non-blocking switch does not achieve normalized throughput of 1.0 because sometimes two hosts transmit to the same destination, causing unavoidable congestion. Although a fat tree is a rearrangeably non-blocking topology, there is a significant gap between naive ECMP forwarding and the hypothetical single non-blocking switch due to collisions where multiple flows are hashed onto the same link. This gap makes the case for traffic engineering: with perfect flow scheduling it should be possible to approach the throughput of a non-blocking switch.

Polling is generally ineffective at detecting short flows (shown in Fig. 5(a)), because these flows are almost always finished before the controller can detect them. Polling is much more effective when elephant flows are longer than the polling interval (in this case one second), as in Fig. 5(b).

OpenSample-MLE outperforms polling in a few cases, but in general it suffers from the sampling bottleneck described earlier; by the time it receives enough samples to be confident that a flow is large, the flow is almost over. Thus OpenSample-MLE schedules relatively few flows.

OpenSample-TCP performs significantly better than either polling or OpenSample-MLE, because it detects and schedules elephant flows earlier. In most cases it achieves performance

close to a non-blocking switch, even for fairly small (1MB) flows. Often it outperforms the alternatives by 25-50%.

Table I gives an intuition of the source of the performance gains. It shows both the total bytes transferred in the 30 s duration of the experiment and the percentage of those bytes that the traffic engineering manages to schedule for the *stride8* benchmark. The fraction of bytes scheduled can be considered a figure of merit for traffic engineering, since any bytes that are not scheduled are more likely to be subject to congestion. By this metric, OpenSample-TCP schedules over twice the fraction of bytes as the polling system. We can also see that the reduced congestion allowed the workload to send twice as much data in the same time, doubling throughput.

Fig. 6 provides deeper insight into the behavior of the measurement systems in the context of traffic engineering. Fig. 6(a) shows the fraction of bytes left in a flow at the time it is detected by each measurement system and Fig. 6(b) shows the fraction of the bytes left in a flow at the time it is actually rerouted, i.e., after the new forwarding rules have been installed. The results show that OpenSample-MLE and OpenSample-TCP dramatically outperform Polling when it comes to detecting short flows. When accounting for the scheduling interval, the time needed to compute routes and install the new routes, the advantage that OpenSample-MLE had vanishes, but OpenSample-TCP is still able to significantly outperform the alternatives.

In conclusion, OpenSample-TCP can detect elephant flows far earlier than the alternatives and, when used to drive traffic engineering, it enables the traffic engineering mechanism to schedule up to 60% of the bytes that hosts send (for *Stride8*) and to a 150% improvement in aggregate throughput (for *Stride4*) when flow sizes are small.

C. Scalability

The OpenSample collector is currently implemented with the assumption that a single machine running the collector will gather samples from all the switches in the network[4]. Thus, the rate of samples that it can handle will limit the number of switches a single collector can monitor. Therefore, to evaluate how many samples per second a single OpenSample collector can handle, we implemented a benchmark tool by modifying Cbench [30]. Cbench is a tool intended to measure the performance of OpenFlow controllers by sending large numbers of `packet_in` messages as if they were from a collection of switches. Our modified version sends sFlow datagrams rather than sending `packet_in` messages.

While we omit a full presentation of these results for varying imposed loads, we found the OpenSample collector was able to process more than 100,000 samples per second. This means our current OpenSample implementation can handle samples from at least 285 switches assuming each switch sends 350 samples per second. That implies OpenSample is able to handle production data centers servicing 4K or 8K hosts with 1:2 or 1:5 oversubscription ratio, respectively [6]. Further, recent SDN controller efforts [31] have shown the ability to process and respond to as many as 10 million events per second. As a consequence, we believe that with more careful engineering, we could handle as many as 28,500 switches with a single collector, but we leave this optimization to future work.

[4]While we believe that it is possible to build a hierarchical version of OpenSample which uses multiple collectors to monitor more switches than a single collector can handle and aggregates their different network views, we leave this as a topic of future work.

V. Related Work

There has been a significant amount of work on WAN traffic engineering, but much less work on traffic engineering on data center networks. Traffic engineering in data centers has only become a topic of interest in recent years due to the adoption of multipath topologies; without multiple paths there is nothing to engineer.

Miura et al. [32] describe cases where parallel workloads can generate traffic patterns that cause congestion in data center networks that use single-path routing. They show that it is possible to increase network utilization by hand-optimizing routing tables, but do not provide any algorithmic solution.

The first practical data center traffic engineering work we know of is Hedera [1]. They found that congestion can occur even in full-bisection-bandwidth networks due to routing collisions—ECMP reduces collisions compared to single-path routing but does not eliminate them. They provide algorithms to estimate demand of network-limited flows and to schedule flows in a Clos network. They poll switch counters every five seconds to detect congestion and elephant flows and they use OpenFlow to reroute flows. Our work is significantly influenced by Hedera, while also taking into consideration the realities and limits of existing switches.

DevoFlow [7] addresses many inefficiencies of OpenFlow by "devolving" control of some things, such as microflow creation and multipath, to switches. They also propose using sFlow sampling but do not implement it using real hardware.

Helios [28] discusses building a fast control loop for hybrid optical/electrical data center networks and is able to complete a full control loop in approximately 100 ms, but does so by minimizing the number of installed flows to read and they make use of a proprietary RPC mechanism rather than a standard measurement mechanism like sFlowm.

MicroTE [8] characterizes several data center workloads, finding similar collision-induced congestion as Hedera and DevoFlow. They modify servers to perform network measurement and report data to a central controller every second, which infers congestion based on server-level information. They also describe optimizations such as server-based aggregation to reduce the overhead of network monitoring. Like Hedera and DevoFlow, MicroTE uses OpenFlow to perform rerouting.

Mahout [2] has similar goals to OpenSample, but seeks to provide low-latency elephant flow detection by using queue depth at end-hosts rather than using network-based measurements. When using switches with support for packet sampling, OpenSample offers a more resource-efficient measurement platform and is potentially more deployable, since changing

switch configuration is likely easier than installing new software on all end-hosts.

Multipath TCP [33] (MPTCP) allows a single TCP connection to be split into multiple *subflows* that take different paths. Each subflow uses TCP congestion control to monitor and respond to congestion and MPTCP dynamically shunts data to less-congested paths. This can be viewed as a monitoring and traffic engineering system that is implemented entirely on end hosts using local knowledge.

OpenSketch [4] proposes configurable network measurement hardware that can calculate approximate "sketches" of common statistics, such as heavy hitters and flow size distribution. If implemented in switches, OpenSketch could enable even faster traffic engineering by efficiently and rapidly detecting elephant flows directly in switches. Although OpenSketch enables software-defined measurement in the same way that OpenFlow enabled software-defined forwarding, it is based on a clean-slate redesign of portions of the switch hardware.

InMon sFlow-RT [34] is similar to the network analyzer component of OpenSample. Although it is designed to integrate with an OpenFlow controller, its traffic engineering capabilities are not clearly documented.

VI. Conclusions and Future Work

We have presented OpenSample, a working prototype of a low-latency, sampling-based measurement platform, and a data center traffic engineering application based on OpenSample. Our primary contribution is lowering the latency to gather accurate measurements of network load and elephant flows from 1–5 seconds to 100 milliseconds. Faster detection of elephant flows allows traffic engineering to better schedule the network, yielding increased throughput—up to 150% in some cases. In general, workloads that cause more congestion benefit more from our work and our improvement over prior efforts is more significant for smaller flows. OpenSample works with unmodified Ethernet switches, making it deployable without waiting for new hardware or modifying end-host software.

We believe that there is significant opportunity for future work in this space as well. We hope to explore dynamically adapting per-port sampling rates based on observations and combining both maximum likelihood and sequence-number–based estimations for improved accuracy. Lastly, merchant silicon vendors may introduce ASICs that support sampling entirely in the data plane allowing for very high sampling rates, i.e., N=64, even with 10 Gbps links. This opens the possibility of control loops that operate as fast as 100 μs.

Acknowledgements: We thank Andreas Kind and Chi "Harold" Liu from IBM Research—China for their work on network monitoring that inspired this project.

References

[1] M. Al-fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *NSDI*, 2010.

[2] A. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *INFOCOM*, 2011.

[3] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *ICFP*, 2011.

[4] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with OpenSketch," in *NSDI*, 2013.

[5] "Openflow-switch," https://www.opennetworking.org/sdn-resources/onf-specifications/openflow.

[6] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter, "PAST: scalable ethernet for data centers," in *CoNEXT*, 2012.

[7] A. R. Curtis, J. C. Mogul, J. Tourrilhes, and P. Yalagandula, "DevoFlow: Scaling flow management for high-performance networks," in *SIGCOMM*, 2011.

[8] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: fine grained traffic engineering for data centers," in *CoNEXT*, 2011.

[9] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall, "Augmenting data center networks with multi-gigabit wireless links," in *SIGCOMM*, 2011.

[10] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *IMC*, 2010.

[11] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM*, 2008.

[12] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber, "HyperX: topology, routing, and packaging of efficient large-scale networks," *SC Conference*, 2009.

[13] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking data centers randomly," in *NSDI*, 2012.

[14] B. Claise, Ed., "Cisco Systems NetFlow Services Export Version 9," RFC 3954. http://www.ietf.org/rfc/rfc3954.txt, October 2004.

[15] "sFlow," http://sflow.org/about/index.php.

[16] V. Mann, A. Vishnoi, and S. Bidkar, "Living on the edge: Monitoring network flows at the edge in cloud data centers," in *COMSNETS*, 2013.

[17] "Sampled NetFlow [Cisco IOS Software Releases 12.0 S]," http://www.cisco.com/en/US/docs/ios/12_0s/feature/guide/12s_sanf.html.

[18] P. Phaal and S. Panchen, "Packet sampling basics," http://www.sflow.org/packetSamplingBasics/index.htm.

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

[20] P. Phaal and M. Lavine, "sFlow Version 5," http://www.sflow.org/sflow_version_5.txt.

[21] "Open vSwitch," http://openvswitch.org.

[22] "Erlang distribution," http://en.wikipedia.org/wiki/Erlang_distribution.

[23] "Floodlight openflow controller," http://www.projectfloodlight.org/floodlight/.

[24] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *CoNEXT*, 2012.

[25] R. Ozdag, "Intel Ethernet Switch FM6000 Series - Software Defined Networking," http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf.

[26] Sivasankar Radhakrishnan, "cluster_loadgen," https://bitbucket.org/nikhilh/mininet_tests/src/9f051450a32a8411f03b7f6bbb99cb436c5a4a73/hedera/hedera.

[27] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An open framework for openflow switch evaluation," in *PAM*, 2012.

[28] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, "Helios: a hybrid electrical/optical switch architecture for modular data centers," in *SIGCOMM*, 2010.

[29] A. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *INFOCOM*, 2011.

[30] "Oflops," http://www.openflow.org/wk/index.php/Oflops.

[31] D. Erickson, "The Beacon OpenFlow Controller," in *HotSDN*, 2013.

[32] S. Miura, T. Boku, T. Okamoto, and T. Hanawa, "A dynamic routing control system for high-performance PC cluster with multi-path Ethernet connection," in *IPDPS*, 2008.

[33] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving Datacenter Performance and Robustness with Multipath TCP," in *SIGCOMM*, 2011.

[34] "sFlow-RT," http://inmon.com/products/sFlow-RT.php.