Planck: Millisecond-scale Monitoring and Control for Commodity Networks

Jeff Rasley[†] Brent Stephens[‡] Colin Dixon[◇] Eric Rozner[*] Wes Felter[*]
Kanak Agarwal[*] John Carter[*] Rodrigo Fonseca[†]

[†]Brown University [‡]Rice University [*]IBM Research–Austin,TX [◇]Brocade

ABSTRACT

Software-defined networking introduces the possibility of building self-tuning networks that constantly monitor network conditions and react rapidly to important events such as congestion. Unfortunately, state-of-the-art monitoring mechanisms for conventional networks require hundreds of milliseconds to seconds to extract global network state, like link utilization or the identity of "elephant" flows. Such latencies are adequate for responding to persistent issues, e.g., link failures or long-lasting congestion, but are inadequate for responding to transient problems, e.g., congestion induced by bursty workloads sharing a link.

In this paper, we present Planck, a novel network measurement architecture that employs oversubscribed port mirroring to extract network information at 280 µs–7 ms timescales on a 1 Gbps commodity switch and 275 µ*s*–4 ms timescales on a 10 Gbps commodity switch, *over 11x and 18x faster than recent approaches*, respectively (and up to 291x if switch firmware allowed buffering to be disabled on some ports). To demonstrate the value of Planck's speed and accuracy, we use it to drive a traffic engineering application that can reroute congested flows in milliseconds. On a 10 Gbps commodity switch, Planck-driven traffic engineering achieves aggregate throughput within 1–4% of optimal for most workloads we evaluated, even with flows as small as 50 MiB, an improvement of up to 53% over previous schemes.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network monitoring*; C.4 [**Performance of Systems**]: Measurement techniques

Keywords

Networking Measurement; Software-Defined Networking; Traffic Engineering

1. INTRODUCTION

Modern data center networks operate at speeds and scales that make it impossible for human operators to respond to transient problems fast enough, e.g., congestion induced by workload dynamics. Gone are the days of monitoring and tuning networks at the granularity of days, hours, and minutes [4]. Even reacting in seconds can cause significant disruption, so data center networks are frequently constructed to have full bisection bandwidth to avoid congestion [1, 13]. This brute force approach adds substantial cost, results in poorly utilized networks, and only reduces the likelihood of issues [14]. Instead, if done quickly enough, detecting congestion and routing traffic to avoid it could both reduce costs and improve performance.

Software-defined networking (SDN) allows for this kind of autonomous, self-tuning network that constantly monitors network conditions and reacts rapidly to problems. Previous work has demonstrated that routes can be installed by an SDN controller in tens of milliseconds [11, 39], but state-of-the-art network measurement systems typically spend hundreds of milliseconds or more collecting statistics [2, 4, 6, 10, 41], which limits the minimum latency of any autonomous measurement-decision-actuation network management control loop. In modern 10 Gbps and 40 Gbps networks, this is too slow to react to any but the largest network events, e.g., link failures, VM migrations, and bulk data movement. Problems induced by transient conditions, e.g., conflicting small-to-medium flows, cannot be identified fast enough to respond before they disappear, resulting in frequent bursts of congestion. To support future autonomous SDNs, a much lower latency network monitoring mechanism is necessary.

This paper introduces Planck, a network measurement architecture that extracts network information at 280 µs–7 ms timescales on a 1 Gbps commodity switch and 275 µ*s*–4 ms timescales on a 10 Gbps commodity switch, *over an order of magnitude (11–18x) faster than state-of-the-art*, see Table 1. Planck achieves this level of performance through a novel use of the *port mirroring* mechanism.

Port mirroring is supported by most modern switches to enable a variety of network monitoring and security applications. When port mirroring is enabled, traffic destined for a *single* port is mirrored to a monitoring port that is connected to a monitoring or intrusion detection system. Planck repurposes this existing port mirroring capability to support an extremely high rate of packet sampling. In Planck, *multiple* (or all) ports are mirrored to a single monitoring port, which introduces a problem: the total traffic flowing through the switch, and thus mirrored to the monitoring port, often will exceed the capacity of the monitoring port. When this happens, the switch mirrors as much as it can and drops the rest, in effect providing a sample of the monitored traffic. In our experimentation, the buffering and drop behaviors of two different commercial switches (IBM RackSwitch G8264 and Pronto 3290) did not persistently fail to sample specific flows and provided samples that allowed for accurate estimates of link utilization and flow rates.

SIGCOMM'14, August 17–22, 2014, Chicago, IL, USA.

ACM 978-1-4503-2836-4/14/08 ...$15.00.
http://dx.doi.org/10.1145/2619239.2626310.

System	Speed	Slowdown vs 10 Gbps Planck
Planck 10 Gbps minbuffer	275–850 µs	1/15–1/5x
Planck 1 Gbps minbuffer	280–1150 µs	1/15–1/4x
Planck 10 Gbps	**< 4.2 ms**	**1x**
Planck 1 Gbps	< 7.2 ms	1.7x
Helios [10]	77.4 ms	18x
sFlow/OpenSample [41]	100 ms	24x
Mahout Polling† [5] (implementing Hedera)	190 ms	45x
DevoFlow Polling† [6]	500 ms–15 s	119–3570x
Hedera [2]	5 s	1190x

Table 1: A comparison of measurement speed and slowdown to gather accurate per-flow throughput information at each link compared to Planck on a 10 Gbps switch. Planck is 11–18x faster than Helios, the next fastest scheme. The "minbuffer" rows show how fast Planck could be on switches that were configured with minimal buffering for mirror ports, a feature our firmware does not expose. A † indicates the value is not the primary implementation proposed in the corresponding work, but is a reported value or estimate. For more details see § 5.5.

Each monitoring port is connected to a *collector* running on a separate server that uses netmap [30] to process traffic sent by a monitoring port at line-rate. A single server hosts many collectors. Planck collectors can record the stream of sampled packets and perform lightweight analysis of the stream to extract information of interest. Collectors export a number of capabilities, including sFlow-style sampling, extremely low latency link utilization and flow rate estimation, and the ability to capture and dump raw packet samples of any traffic in the network. Applications can query the collector for statistics or subscribe and respond to notifications from the collector, e.g., when a specific level of congestion is detected. To support these capabilities, Planck requires as little as one port per switch and one server per fourteen switches to handle samples.

To demonstrate the value of Planck's speed and accuracy, we built a traffic engineering application that (i) uses Planck to monitor the network, (ii) decides when conditions warrant reconfiguring the network, and, (iii) when so, executes the preferred reconfiguration. This Planck-driven traffic engineering achieves aggregate throughput within 1–4% of optimal for most workloads we evaluated on a 10 Gbps commodity switch, even with flows as small as 50 MiB, an improvement of up to 53% over previous schemes.

This paper makes four main contributions:

1. We present a novel measurement platform, Planck, that uses oversubscribed port mirroring and high speed packet processing to provide millisecond-scale network monitoring.
2. We provide a detailed analysis of switch packet drop and buffering policies and their impact on monitoring latency.
3. We develop an algorithm to accurately estimate a flow's throughput within a 200–700 µs timescale, using samples obtained from an unknown sampling function.
4. We demonstrate the feasibility of millisecond timescale traffic engineering on commercial 10 Gbps switches using a combination of Planck and an SDN application that responds to congestion using ARP messages to switch rapidly between pre-installed alternate routes.

The remainder of this paper is organized as follows. In Section 2 we present background on low-latency network measurement. Sections 3 and 4 describe the design and implementation of Planck. Section 5 evaluates Planck including the impact of oversubscribed port mirroring on network traffic and the nature of Planck's samples. Section 6 describes two applications that use Planck, a vantage point monitor and traffic engineering tool. We evaluate our Planck-based traffic engineering application in Section 7 and then discuss related work in Section 8. Finally, we provide a retrospective discussion in Section 9 and conclude in Section 10.

2. BACKGROUND

Network measurement is too broad a field to fully characterize here. We focus on measurement techniques that are useful in discovering either link utilization or the most significant flows crossing each link at fine time granularities, i.e., seconds or faster. We omit a discussion of probe-based measurement techniques. While they can discover network conditions, e.g., congestion, they typically cannot determine the specific traffic causing those conditions.

2.1 Packet Sampling

In packet sampling, switches forward one-in-N packets they receive, along with metadata, such as the packet's input port, to a *collector*. The collector then estimates the traffic on the network by multiplying the packet and byte counts from the samples by N [29].

Sampling forms the core of the sFlow [32] standard that many switches implement. sFlow typically strips off the packet payload and adds metadata such as a switch ID, the sampling rate used when this packet was selected, and the output port(s) selected for this packet. Sampled packets are then sent out via the control plane CPU of the switch. Figure 1(a) shows the path samples take when using sFlow with the dashed line labeled 'sFlow'.

Recent work [41] has reported that involving the switch control plane CPU and the PCI bus connecting them limits the achievable sampling rate. In the case of the IBM RackSwitch G8264 [16], the maximum rate is about 300 samples per second. This low sample rate results in high estimation error unless samples are aggregated over long periods of time, i.e, a second or more. The error for a throughput estimate from s samples is approximately $196 \cdot \sqrt{1/s}$ [29], so even if all 300 samples in a second come from a single link, that link's estimated load will be off by about 11%. In more realistic scenarios the error will be noticeably worse: the collector will have to wait longer than a second to report information.

2.2 Port Counters

Switches usually maintain counters that track the number of bytes and packets that are sent and received for each port. While these counters provide little direct insight into the flows crossing each link[1], they can be periodically polled to infer link utilization over time. Port counters can be read via most existing switch interfaces including SNMP [38], OpenFlow [26], and sFlow [32].

2.3 Flow Counters

Many switches track the number of packets and bytes handled by individual match-action table entries, which can be used to track individual flows. When available, these counters are exported by querying so-called "ACL rules", OpenFlow rules, and/or NetFlow [3].

Flow-based counters have a number of limitations. First, ACL/OpenFlow tables are typically small, e.g., the IBM Rackswitch G8264 supports only about 1,000 flow-granularity rules. Also, many switches do not support fast flow counter polling—it can take seconds to read all of them [6]. More recent work indicates that it takes 75–200 ms [5, 10] to extract flow counters, as shown in Table 1.

[1]The field of network tomography provides some insights into how one might deduce flows from port counters. These approaches are usually time-intensive and at best give probabilistic information at host-pair granularity [22].

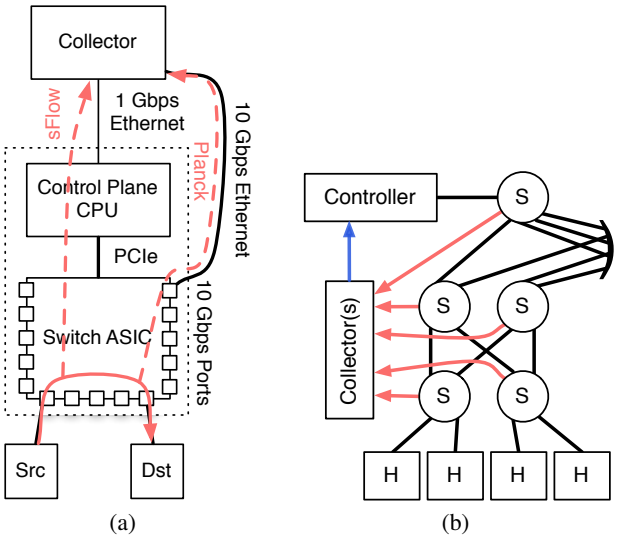

Figure 1: Planck architecture: (a) Switch architecture illustrating the fast (Planck) and slow (sFlow) paths. (b) Network architecture showing one pod of a fat tree consisting of hosts (H), switches (S), collector(s), and the SDN controller. Traffic from hosts is forwarded by the switches to the collector(s), where they are processed. The collector(s) send events to the controller, which can send messages to the switches to reconfigure routes.

NetFlow takes a slightly different approach—it maintains a cache of information on active TCP and UDP flows. Whenever a switch receives a packet, NetFlow checks to see if the packet belongs to a cached flow. If so, it increments the associated flow counters. If not, it creates a new cache entry. If the cache is full, an older entry is evicted and sent to the collector. This approach uses the collector like a backing store for information about layer-4 flows crossing a given device. Cache entries also can be configured to time out periodically, giving the collector a more up-to-date view of the network, but the timeouts are on the order of seconds [3], which provides little or no advantage over counter polling.

3. DESIGN

Figure 1 illustrates Planck's three main components: (i) switches configured to provide samples at high rates, (ii) a set of collectors that process those samples and turn them into events and queryable data, and (iii) a controller that can act on the events and data. The remainder of this section discusses each of those elements in turn.

3.1 Fast Sampling at Switches

As previously discussed, current mechanisms for extracting switch measurements leave much to be desired. Even sFlow [32], which is designed to provide samples in real time, can only generate hundreds of samples per second on a modern 10 Gbps switch. At that rate, it takes seconds to infer network state with accuracy. We overcome this limitation by leveraging the port mirroring feature found in most commodity switches. Port mirroring enables non-disruptive traffic monitoring by *replicating* all—or a subset of—traffic destined for a given output port to a designated *monitor port*.

We repurpose this functionality to support high sampling rates by oversubscribing the monitor port(s), i.e., configuring the switch such that traffic destined to multiple output ports is replicated to each monitor port. When the total traffic flowing through the mirrored ports is light, all traffic is forwarded out the monitor port(s). However, when incoming traffic exceeds the capacity of the monitor port(s), congestion ensues and mirrored packets get buffered. If congestion persists, eventually the buffer fills up, at which time the switch starts dropping mirrored packets. This effect constrains the rate of samples that Planck can capture to the aggregate bandwidth of the monitor port(s). In general, we can subdivide a given switch's N ports in to k monitor ports and $N - k$ normal data ports. In the common case, we expect that $k = 1$ will provide more than enough visibility into the network at the cost of giving up only a single port per switch. In Section 5 we show that this approach is feasible.

One complication of this design is that the sampling rate, the fraction of packets passing through the switch that are mirrored, varies as traffic load changes. When traffic flowing through the switch is light, 100% of packets are mirrored. Once traffic exceeds the capacity of the monitor port(s), Planck is limited to receiving as many packets as the monitor port(s) can carry. The instantaneous sampling rate is unknown because it dynamically scales proportional to the rate of traffic being mirrored divided by the bandwidth of the output port(s). We compensate for this uncertainty by using sequence numbers to compute throughput (see Section 3.2.2) rather than requiring a known sampling rate.

3.2 Collector

The collector has four major goals: (i) process sampled packets at line rate, (ii) infer the input and output ports for each packet, (iii) determine flow rates and link utilization, and (iv) answer relevant queries about the state of the network.

The collector uses netmap [30] for line-rate processing and borrows from a substantial body of work on line-rate packet processing on commodity servers [18, 19, 24, 28], so only the last three goals are discussed in detail here. Without monitor port buffer latency, the collector can determine reasonably stable global network statistics every few hundred microseconds, which is on par with an RTT in our network. While the collector determines utilization statistics on shorter time periods, these statistics are not stable due to the on/off nature of Ethernet and the bursty behavior of TCP.

3.2.1 *Determining Input and Output Ports*

Traditional network sampling techniques append metadata, e.g., input and output ports, to each packet sample. This metadata is important for determining if a given port (link) is congested. Mirrored packets do not include metadata, so the collector must infer input and output ports from the packet alone. To solve this problem, the SDN controller shares the topology of the network and the rules embedded in each switch's forwarding tables with the Planck collector. As long as the network employs deterministic routing, which includes ECMP if the hash function is known, the collector can infer the full path that a packet follows based on the packet header, and thus determine the input and output port the packet traversed through a particular switch. Keeping the forwarding table information consistent between the controller, switches, and collector(s) requires care, but in practice it does not vary quickly. Alternately, the collectors can infer the path any given flow takes, and thus the input and output ports on each switch, based on samples from multiple switches. However, depending on the sampling rate and the length of a flow, the information about paths could be incomplete.

3.2.2 *Determining Flow Rates and Link Utilization*

The Planck collector parses packet headers to maintain a NetFlow-like flow table that tracks information about individual TCP flows, including their throughputs and paths through the network. Traditionally, sampling-based measurements determine flow rates and

link utilization by multiplying the throughput of samples received for a given flow or port by the sampling rate. However, port mirroring does not provide a fixed sampling rate.

To determine the throughput of TCP flows, the collector tracks byte counts over time by using the TCP sequence numbers, which are byte counters in and of themselves. If the collector receives a TCP packet A with sequence number S_A at time t_A, and a TCP packet B with sequence number S_B at time t_B from the same flow, such that $t_A < t_B$, the collector is able to recover an estimate of the throughput of that flow. Upon receiving such a packet B, the collector can infer the flow's throughput information by computing $(S_B - S_A)/(t_B - t_A)$. To compute link utilization, the controller sums the throughput of all flows traversing a given link.

By itself, this approach tends to produce jittery rate estimates due to TCP's bursty behavior, especially during slow start. To reduce jitter, Planck clusters packets into *bursts* (i.e. collections of packets sent in quick succession) separated from other bursts by sizable time gaps[2]. This behavior is common, for example, during TCP slow start. Bandwidth estimates are done on bursts. Once a flow hits steady state, the gaps between packets shrink as TCP settles into a continuous stream of packets clocked by received ACKs. Planck limits a burst to at most 700 µs to get regular rate estimates from flows once they are in steady state.

If the collector sees an out of order packet, i.e., $t_A < t_B$ and $S_A > S_B$, it cannot determine if this is due to reordering or retransmission, and thus ignores the packet when it comes to throughput estimation. In practice, out of order packets are uncommon enough in data center networks that ignoring them does not affect accuracy significantly. A collector could infer the rate of retransmissions based on the number of duplicate TCP sequence numbers it sees, but we leave this to future work.

While the discussion thus far has focused on TCP, many other types of traffic contain sequence numbers, and the previously described method is general enough to apply to any packet type that places sequence numbers in packets. If the sequence numbers represent packets rather than bytes, then they need to be multiplied by the average packet size seen in samples as well, but this shouldn't significantly hurt rate estimation. We leave developing a model for throughput of flows without sequence numbers to future work.

3.3 Controller

The SDN controller performs two Planck-specific functions: (i) install mirroring rules in the switches and (ii) keep the collector(s) informed of the network topology and forwarding rules. It also exports functionality to network applications: (iii) the ability to query for link and flow statistics and (iv) the ability to subscribe to collector events.

The simplest Planck-specific SDN controller extension is to forward statistics requests to the collectors. This acts as a drop-in replacement for most, if not all, SDN controller statistics APIs and typically results in a much lower latency statistics capability than provided by SDN controllers, e.g., using OpenFlow counters.

Perhaps a more interesting new feature of Planck is that it allows SDN applications to subscribe to events generated by the collector(s). This allows SDN controller applications to start reacting to network events within milliseconds of when they are detected by the collector(s). Currently, the only events exported by the collector(s) are when link utilizations cross a specified threshold, but it would be straightforward to add others. Events include context annotations, e.g., the link congestion event includes the flows using the link and their current estimated rates. Annotations are intended to help an application better respond to the event, e.g., reroute an individual flow to deal with the congestion.

[2]At 10 Gbps, we find 200 µs works well as a minimum gap size.

4. IMPLEMENTATION

Planck's implementation consists of a fast packet collector and an extended OpenFlow controller. The collector is built using netmap [30] and the controller is built on Floodlight [12]. We detail the base implementation in this section; in Section 6 we discuss extensions for vantage point monitoring and traffic engineering.

4.1 Base Controller

The Planck controller is a modified and extended Floodlight OpenFlow controller [12]. Its main features are (i) a modified routing module that adds actions to the switch forwarding tables to mirror traffic and (ii) a module that communicates installed and alternate routes to the collector(s) for traffic input-output port inference. The former involved modifying Floodlight's routing module to dynamically add a second OpenFlow output action to each rule that replicates traffic to the appropriate monitor port. The latter involved extending the controller to broadcast updates to the collector(s) whenever it modifies existing routes or installs new ones. The controller also refrains from using new or modified routes for a short period of time after installing them to give the collector(s) time to process the new route information and thus infer input and output ports correctly.

4.2 Collector

In our current design, each monitor port is associated with a separate collector process and is connected directly to the server on which this process executes, i.e., mirrored traffic does not traverse intermediate switches. Numerous collector instances are run on a single physical machine. Each collector instance runs in user space and connects to the netmap [30] Linux kernel module to receive samples. Because the collector knows the current state of its switch's forwarding tables and because we route on MAC addresses, it can uniquely identify an arbitrary packet's output port based solely on the destination MAC address of the packet. Further, it can uniquely identify the input port based only on the source-destination MAC address pair.

Our current implementation supports three queries: (i) link utilization, (ii) rate estimation of flows crossing a given link, and (iii) a raw dump of a configurable number of the last packet samples. Additionally, it allows for subscription to link utilization events. We leave the development and support for more queries to future work.

5. PLANCK EVALUATION

In this section, we describe microbenchmarks we ran to: (i) determine the impact of oversubscribed port mirroring on switch traffic, (ii) characterize sampled traffic, and (iii) evaluate the accuracy of Planck's throughput estimation scheme. All the microbenchmarks are performed on a single switch—testbed details can be found in Section 7.1.

5.1 Impact of Mirroring on Switch Traffic

To accommodate transient output port congestion, modern switches dedicate a modest amount of memory for buffering packets waiting to be forwarded. For example, the Broadcom Trident switch ASIC contains 9 MB of buffer space shared between its 64 ports, of which a small amount is dedicated to each output port while most is allocated dynamically. A single congested port can consume up to 4 MB of buffer space, which for a 10 Gbps switch adds 3.5 ms of queueing latency. Counterintuitively, latency induced by congestion

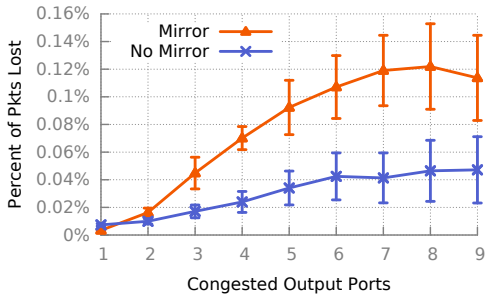

Figure 2: Drops of non-mirrored packets, as logged on the switch, as the number of congested output ports is varied.

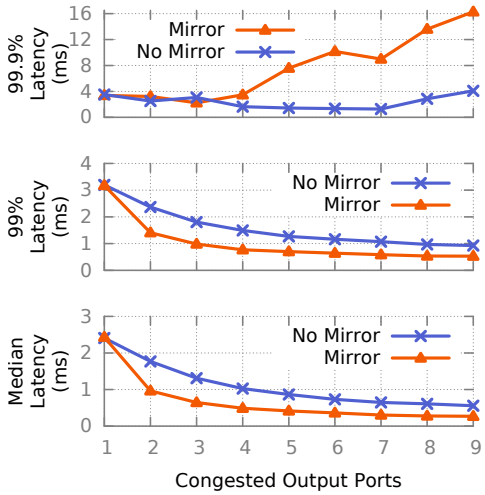

Figure 3: Latency of non-mirrored traffic as the number of congested output ports is varied.

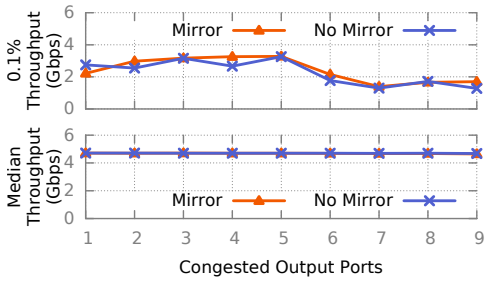

Figure 4: Flow throughput, averaged over 1 second intervals, as the number of congested output ports is varied.

decreases as more ports become congested because each port receives a smaller share of the shared buffer. Since Planck deliberately oversubscribes mirror ports, mirrored traffic is frequently buffered, which has two negative effects: (i) samples are delayed and (ii) any buffer space used to buffer samples is not available to handle bursts on other (non-mirror) ports.

To understand the impact of oversubscribed port mirroring on the non-mirrored (original) traffic passing through the switch, we conduct a series of tests to measure the loss, latency, and throughput of non-mirrored traffic under a variety of network conditions when mirroring is enabled and disabled. We vary the number of *congested ports*, ones where two hosts saturate TCP traffic to the same destination, from one (three hosts) to nine (27 hosts) to stress the shared switch buffers. All graphs are generated over 15 runs.

Port mirroring uses some of the shared buffer space on the switch, and thus there is less buffer space available to the other ports. Figure 2 shows that port mirroring can increase non-mirrored traffic loss due to decreased buffer space, but the absolute drop rates are very small, with average loss less than 0.12%. Figure 3 shows the latency of non-mirrored traffic as we vary the number of congested ports. Decreased buffer space at the switch manifests itself in *lower* latencies for the average (not shown), median, and 99th-percentile cases when mirroring is enabled. Also shown in the figure is the 99.9th-percentile latency, which shows the effect of retransmission delays from extra loss incurred due to buffer space contention. Note that since loss rates are low, we only see latency affected in the 99.9th-percentile. Finally, Figure 4 shows that the median and tail (0.1th-percentile) flow throughput for non-mirrored traffic is unaffected by mirroring. As discussed in Section 9.2, limiting the buffer space made available to sampling ports should mitigate the already small impacts of our scheme.

5.2 Undersubscribed Sample Latency

Ideally, to measure the sample latency, we would measure the time from when the first bit of a packet arrived on the wire toward a switch, i.e., when the packet started consuming network resources, to when the packet arrived at the collector and could be processed. Unfortunately, we were not able to measure this precisely because our testbed did not provide a way to get an accurate timestamp for when a packet was placed on the wire. Instead, we measured the time between when `tcpdump` reported the packet being sent to when the collector received the packet. This includes some delay at the sender as the packet traverses the lower parts of the kernel and the NIC, which is not technically sample delay. As a result our measurements are strict overestimates of true sample latency.

To eliminate clock skew, we used separate NICs on the same physical server as both the sender and the collector. In an otherwise idle network, we observed sample latencies of 75–150 µs on our 10 Gbps network and 80–450 µs on our 1 Gbps network.

5.3 Analysis of Sampled Data

We then investigated the characteristics of sample data by varying the number of max-rate flows with unique source-destination pairs mirrored to the same monitor port on the switch. All flows have dedicated ports on the switches, so in the absence of congestion, each flow saturates its links. We analyze two metrics on the sampled data: *burst length*, the number of packets from a given flow received consecutively by the collector, and *inter-arrival length*, the number of packets from other flows received between two bursts of a given flow. In an idealized setting where each flow has identical offered (max) load and the switch performs perfect round-robin scheduling, burst length would be one and inter-arrival length would be NUMFLOWS − 1.

Figure 5 shows a CDF of burst length, in MTUs (i.e., 1500 bytes), for 13 concurrent flows on the switch. Over 96% of the time, burst length is less than or equal to one MTU, which indicates the scheduler on the monitor port typically samples one packet at a time from each flow under saturated conditions. Figure 6 shows average inter-arrival length, in MTUs, for a varying number of flows. For more than four flows in the network, inter-arrival length increases linearly, as predicted. In this case we omitted error bars because they are large, e.g., the standard deviation for 13 flows is roughly 28. The red (top) line in Figure 7 shows the CDF of inter-arrival lengths, in MTUs, for 13 flows. Over 85% of the time the inter-arrival time

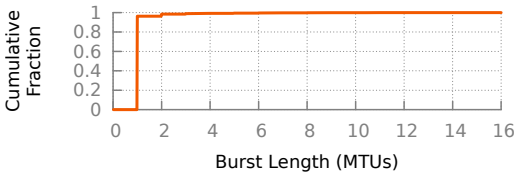

Figure 5: CDF of burst lengths, in MTUs, for 13 concurrent flows on sampled data.

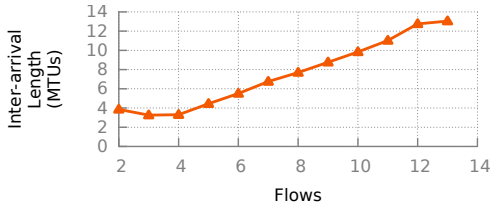

Figure 6: Inter-arrival lengths, in MTUs, for varying number of flows on sampled data.

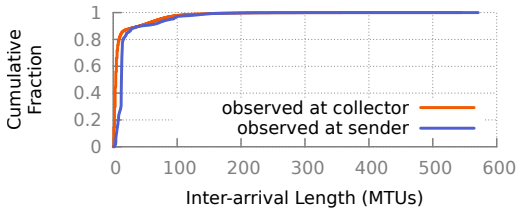

Figure 7: CDF of inter-arrival lengths, in MTUs, for 13 concurrent flows. The red line shows the inter-arrival lengths on the sampled data, and the blue line shows the length of bursts that could fit between a sender's non-transmitting periods.

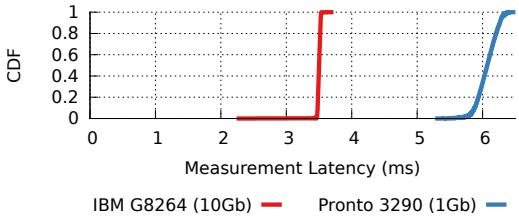

Figure 8: An experiment showing the latency between when a packet is sent and received by the collector on 10 Gbps and 1 Gbps switches during high congestion.

is less than or equal to 13 MTUs, but there is a long tail. Sender burstiness is a known problem [23] and the blue (bottom) line in Figure 7 quantifies this burstiness by showing the number of MTUs that could be sent in the gaps between packet departure times at a flow source in our testbed. The fact that the gaps observed at the sender closely match the gaps observed at the collector indicates, that the large inter-arrival times are an artifact of TCP and not Planck.

Finally, we examine the latency induced on mirrored traffic due to oversubscription. Figure 8 presents a CDF of the latency between when a packet is sent and the collector receiving it. Three hosts send saturated TCP traffic to a unique destination to oversubscribe the monitor port. We measure the latency imposed on mirrored data for an IBM G8264 10 Gbps switch and a Pronto 3290 1 Gbps switch, and observe a median latency of roughly 3.5 ms on the 10 Gbps network and just over 6 ms on the 1 Gbps network. Figure 9 shows the average sample latency as we vary the *oversubscription factor* on the 10 Gbps switch. The oversubscription factor indicates how many more times traffic we are sending to the monitor port than its capacity. The roughly constant observed latency indicates that the IBM switch likely allocates a fixed amount of buffer space to the mirrored port.

5.4 Throughput Estimation

Figure 10(a) shows the collector's throughput estimate of a single TCP flow as it starts, using a 200 µs rolling average. These results illustrate the perils of estimating throughput at microsecond scales. During slow start, TCP sends short bursts of packets at line rate followed by periods of nearly an RTT of no packets. As the RTT on our network varies from about 180 µs to 250 µs during the run, the rolling window usually captures one burst, but sometimes includes either zero or two bursts, which causes substantial variation in the instantaneous throughput estimates.

As pointed out in prior work [23], bursty behavior is common in 10 Gbps environments. To get accurate measurements, Planck uses the window-based rate estimation mechanism described in Section 3.2.2 using 200 µs gaps to separate bursts. This results in more stable estimates, as seen in Figure 10(b).

We further investigate the performance of Planck rate estimation in Figure 11. As the throughput across the entire switch increases, the sampling rate of our measurements decreases, which could reduce accuracy. In this experiment we increase the oversubscription rate and measure the mean relative error of the rate estimates. We obtained ground truth sending rates by using the rate estimation described in Section 3.2.2 on the full `tcpdump` traces from the sender and compared them with Planck's throughput estimate which resulted in a roughly constant error rate of 3%.

5.5 Latency Breakdown

In order to better understand all of the events that contribute to measurement latency we broke down the events a sample packet sees into a timeline, seen in Figure 12.

This timeline shows the events we were able to directly measure, such as when a packet is sent (seen via `tcpdump`), when our collector sees it (via netmap) and when our collector has an accurate estimate of the flow's throughput (via our rate estimator).

On a 10 Gbps network with minimum buffering on the monitoring port we see a total measurement delay of 275–850 µs, which comes from 75–150 µs (§5.2) until our collector sees the sample and 200–700 µs (§5.4) until we have a stable rate estimate of the flow. We see similar results on a 1 Gbps network, where the total measurement latency is 280–1150 µs coming from a sample delay of 80–450 µs (§5.2) and the same rate estimation delay.

In terms of prior measurement work presented in Table 1, we see a 291x speed-up when comparing this minimum-buffering case against the reported measurement latency of Helios [10]. As our switch's firmware did not allow us to minimize the monitor port's buffering, we report our worst-cast measurement latency of 4.2 ms at 10 Gbps, or an 18x speed-up.

6. APPLICATIONS

In addition to the core Planck architecture, we have built two applications that exemplify ways that Planck can be extended.

6.1 Vantage Point Monitoring

While there are exceedingly good tools for capturing traffic from end hosts, most notably `tcpdump`, there are far fewer tools for capturing high-fidelity traffic traces from switches short of port-

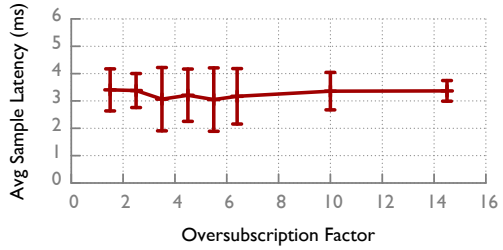

Figure 9: Planck's latency between when a packet is sent and received by the collector on 10 Gbps with various oversubscription factors. As an example, an oversubscription factor of 1.5 means we sent 15 Gbps worth of traffic through a 10 Gbps monitor port.

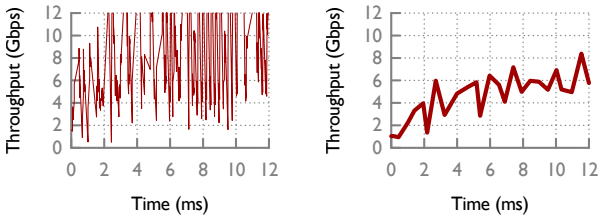

Figure 10: Planck's estimate of throughput as a TCP flows starts using (a) a 200 µs rolling average and (b) Planck's smoothed rate estimator.

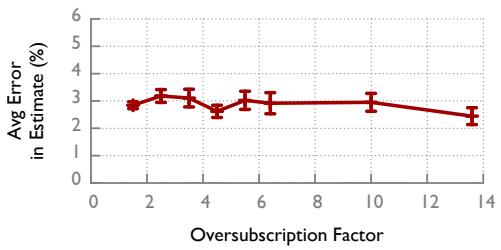

Figure 11: Planck's error in estimating throughput with various oversubscription factors stays around 3%.

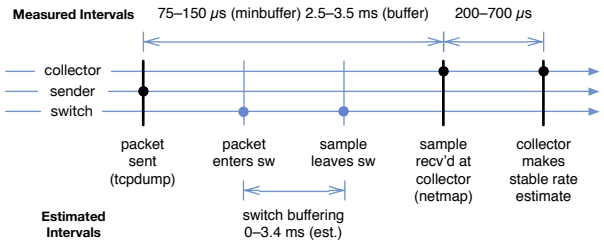

Figure 12: Timeline of measured and estimated sample latency events on a 10 Gbps network. Black vertical bars indicate accurately measured events. Blue vertical bars indicate estimates.

mirroring one port at a time to a monitoring port and running `tcpdump` on an attached machine.

Instead, our vantage point monitoring application runs inside the collector and stores as many recently received samples from each switch as is possible and writes them to a `tcpdump`-compatible pcap [27] file when asked. While this doesn't provide a full trace due to sampling, it provides a high fidelity view of what the switch is actually seeing while only giving up a single output port to monitor the full switch. Planck's input and output port information can also be included by creating a pcap file for each port.

These pcap files can then be used with normal network monitoring and measurement tools like wireshark [43] and `tcpdump`. We also note that this application provides the data for much of our evaluation as it provides data at finer time scales and with more consistent timestamps than other techniques we tried.

As future work we intend to provide options to infer missed packets for TCP to provide more complete traces as well as explore how to use this application to create network traces at the switch level for use in creating realistic workloads and other research efforts.

6.2 Traffic Engineering

To show Planck's value when running realistic workloads, we built a traffic engineering application on top of our base controller (see Section 4.1) which uses link congestion events produced by the collector, combined with the other features of the platform, to quickly reroute flows from congested paths to more lightly-loaded ones. They key benefit of using Planck for this application is its very fast control loop, which, as we show in Section 7, greatly improves the achieved aggregate throughput.

With network events being triggered in hundreds of microseconds, for a controller to keep pace, its decisions about how to alter forwarding behavior must be based on simple calculations. Our approach is to pre-install alternate paths between each source and destination. Thus, the controller only needs to decide which of the alternate paths to use, if any, when reacting to an event. In effect this splits path selection into two parts: first selecting a good set of possible paths offline and second selecting among them in an online setting. This drastically reduces the online computation required.

Our application has three main components that extend the base Planck infrastructure, and which we describe below: (i) the ability to find and pre-install multiple paths in the network, (ii) an algorithm for selecting new paths in the event of congestion, and (iii) a mechanism to quickly install route changes in the network.

Alternate Forwarding Paths Our controller application, implemented as an additional module on our Floodlight-based controller, uses layer-2 Per-Address Spanning Trees (PAST) for multipath routing [39]. PAST provides similar functionality and performance as Equal-Cost Multipath (ECMP) routing does at layer-3. Lastly, we chose PAST for its scalability and because, unlike ECMP, our testbed supports it.

In addition to installing standard PAST routes, a destination-oriented spanning tree for every reachable MAC address, the controller installs three additional spanning trees per host. Each alternate path is provisioned with its own unused, unique destination MAC address, which we term a *shadow MAC address*. We install four paths in total, but this number is not fundamental. However, four carefully chosen alternate paths are sufficient for traffic engineering on our topology.

An example of shadow MAC addresses can bee seen in Figure 13. Using destination MAC addresses that differ from host MAC address can be problematic for the destination host because, by default, hosts do not accept packets not destined for them. However, this problem is easily resolved by installing MAC rewrite rules at the destination's egress switch. Even in physical switches, these extra rules only require TCAM state proportional to the number of alternate routes and hosts per switch.

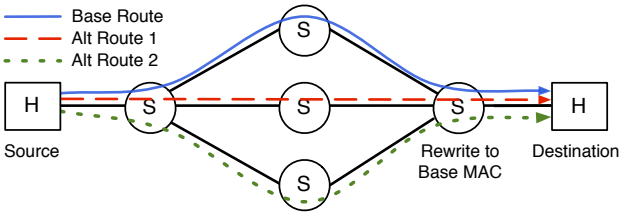

Figure 13: Example of alternate routes between a source and destination using shadow MAC addresses.

Algorithm 1 – Traffic Engineering Application

Input: A flow congestion notification (congn), the application's view of the network (net), and a flow timeout (ftimeout)
Output: Route control notifications

```
1  process_cong_ntfy(congn, net, ftimeout):
2    flows = get_congn_flows(congn, net)
3    paths = get_flow_paths(congn, net, flows)
4    net_update_state(congn, net, flows, paths)
5    remove_old_flows(net, ftimeout)
6    for flow in flows:
7      greedy_route_flow(net, flow)
8
9  greedy_route_flow(net, flow):
10   bestpath = net_rem_flow_path(net, flow)
11   bestbtlneck = find_path_btlneck(net, bestpath)
12   for path in flow.altpaths:
13     if find_path_btlneck(net, path) > bestbtlneck:
14       bestpath = path
15       bestbltneck = find_path_btlneck(net, path)
16   net_add_flow_path(net, flow, bestpath)
17   send_route_control(flow, bestpath)
```

Choosing alternate paths with sufficient path diversity is necessary for traffic engineering. Consequently, choosing paths with many common links can lead to an inability to avoid congestion. Currently, our routing implementation uses the fact that finding edge-disjoint spanning trees in fat trees is trivial as each core switch defines a unique spanning tree. However, this computation is done offline, and more complex algorithms for different topologies or applications are possible. We leave optimizing the set of pre-installed paths for other metrics and determining how many alternate paths are required to future work.

Reacting to Congestion The controller subscribes to link utilization events from the collector(s). The core design for traffic engineering is to, for every congestion notification, greedily route each flow in the notification to a less congested path, if possible, by quickly changing routing labels. The greedy routing of flows, presented in Algorithm 1, which uses Algorithm 1 from DevoFlow [6] to implement `find_path_btlneck`, considers each flow independently and finds the alternate path of a flow with the largest expected bottleneck capacity, which, for a constant number of alternate paths of length P, is only expected to be an $O(P)$ operation.

When selecting alternate paths, flow rerouting must consider flows on other links that it has heard of in the past. In order to avoid using stale information, flow entries in the controller are expunged after a specified timeout and thus traffic engineering does not consider them when calculating available bandwidth along routes.

Fast Rerouting In addition to implementing Algorithm 1 and to leverage shadow-MAC-address-based alternate routes, a mechanism for dynamically changing a flow's destination MAC address is required. We implemented two different mechanisms to switch a flow's destination MAC address, both requiring a single message: (i) using an OpenFlow rule to rewrite the destination MAC address at the source host's ingress switch and (ii) using spoofed ARP messages to update the ARP cache of the source host.

The OpenFlow-based rerouting mechanism is straightforward, but the TCAM rule state requirements of this method at each switch can be proportional to the number of hosts in the network. To address this problem, we developed MAC address rewriting through ARP spoofing, which requires no switch state, and thus no OpenFlow rule installation to reroute a flow. If the controller sends an ARP message pretending to be from the destination IP but using the alternate shadow MAC address, the source will update its ARP cache to the new address and almost immediately send packets using the new route. Although this may seem unsafe at first, since this is essentially ARP poisoning, in many SDN environments the controller intercepts all ARP messages from hosts, so the controller is the only entity capable of sending ARP messages.

Two caveats to the ARP-based rerouting mechanism are that some operating systems, e.g., Linux, ignore spurious ARP replies and in addition lock an ARP cache entry for a period of time after changing. The first problem is solved by sending unicast ARP requests which, on Linux, still performs MAC learning for the request and thus updates its ARP cache. The second method requires setting a `sysctl` to enable faster ARP cache updates.

7. APPLICATION EVALUATION

We evaluate our Planck-based traffic engineering application with a series of synthetic and realistic workloads based on similar tests done in previous related work [2, 6, 10].

7.1 Methodology

Testbed All experiments were conducted on a physical testbed consisting of 16 workload generator servers, five collector servers and five 64-port, 10 Gbps IBM RackSwitch G8264 top-of-rack switches [16]. The workload generators are IBM x3620 M3s with six-core Intel Westmere-EP 2.53 GHz CPUs and Mellanox ConnectX 10 Gbps NICs. The collector machines are IBM x3650 M4s with two eight-core Intel Sandy Bridge-EP 2.4 GHz CPUs and seven two-port Intel 82599 10 Gbps NICs. We used Linux 3.5. Note that our experiments use only a fraction of the resources of these servers.

Topology To evaluate Planck and its applications, we wanted a network topology that provided high path diversity. We chose to build a three-tier fat-tree [1] with 16 hosts.

We built the 16-host fat-tree topology by subdividing four 64-port switches into 20 five-port logical switches using OpenFlow. Due to limited TCAM size we place only five logical switches on each physical switch, leaving many ports unused. Four ports of the subswitch were wired up to build the fat-tree, the final port was used for sampling and connected directly to a collector interface.

Workloads We evaluate the traffic engineering schemes on our testbed with a set of synthetic and realistic workloads, similar to previous related work [2, 6, 10]. A brief description of the workloads follows. As in the prior work, host indices used in the descriptions are contiguous within pods.

Stride(8): The node with index x sends a flow to the node with index $(x + 8) \mod (\text{num\_hosts})$. Stride(8) is a taxing workload because all flows traverse the core.

Shuffle: Each node sends a large amount of data to every other node in the network in random order. This workload mimics Hadoop/MadReduce workloads commonly encountered in real data center networks. In our evaluation, each node sends to two other nodes at a time, and the shuffle completes when all nodes finish.

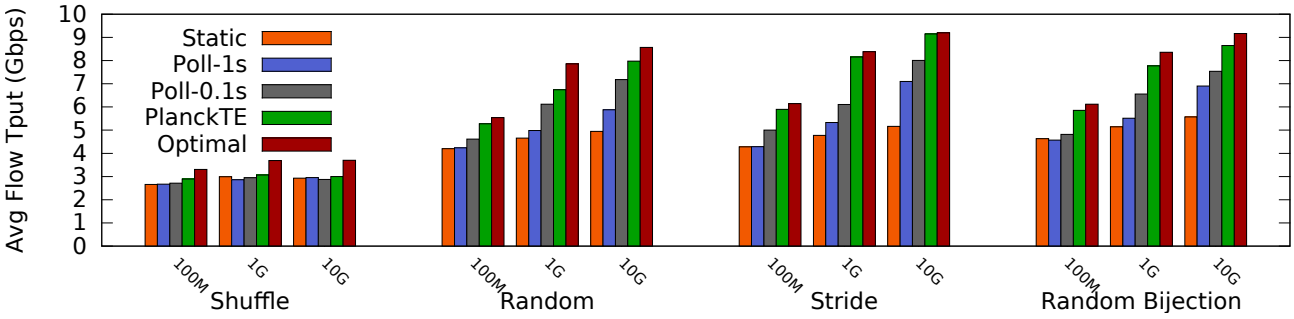

Figure 14: Average flow throughput in each workload.

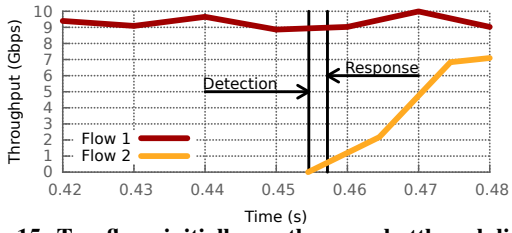

Figure 15: Two flows initially use the same bottleneck link and are then rerouted, demonstrating the latency to detect congestion and reroute one of the flows. Flow 1 does not reduce its sending rate because the detection and rerouting occurs *faster than the switch's buffer fills,* so it sees no loss.

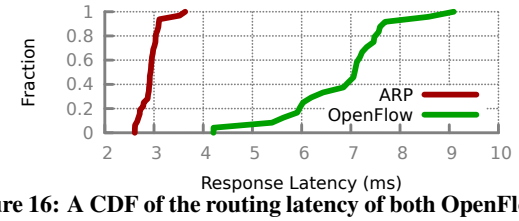

Figure 16: A CDF of the routing latency of both OpenFlow rule control and ARP-based control.

Random Bijection: Every node is exactly the source of one flow and the destination of another flow. Each run in our experiment represents a different random bijection mapping.

Random: Every node picks a destination not equal to itself from a uniform distribution. Each run is a different mapping. These runs allow for hotspots to form in the network.

For each of the above workloads, we perform experiments with 100 MiB, 1 GiB, and 10 GiB flow sizes, unless otherwise noted. A flow size in the shuffle workload represents the amount of data each host needs to send to another host, so a 1 GiB workload represents a 240 GiB shuffle. We tried different configurations of shuffle and stride, and also other workloads such as Staggered Prob (as in [2]), but we found the trends to be consistent, so we omit those results for brevity. All experiments are run using TCP and we run all <workload, flow size> combinations over 15 runs.

We run four different routing algorithms for each of the workloads. As an upper bound, all 16 hosts connect to one of our 64-port 10 Gbps Ethernet switches. This topology is used to represent an optimal non-blocking network, referenced by the name *Optimal*. To provide a baseline, we use PAST [39], a static multipath routing algorithm with performance comparable to ECMP, which is referenced by the name *Static*. To emulate previous traffic engineering projects that rely on polling [2, 6], we implement a global first fit routing algorithm that greedily reroutes every flow either once a second, which is referenced by the name *Poll-1s*, or once every 100 ms, which is called *Poll-0.1s*. Lastly, the traffic engineering algorithm described in Section 6.2 uses a 3 ms flow timeout, approximately the latency of rerouting a flow, and is referenced by the name *PlanckTE*.

7.2 Planck Control Loop Latency

Figure 15 demonstrates the full control loop of our system when ARP messages are used for fast control. In this experiment, we show the throughput over time of two flows, *Flow 1* and *Flow 2*. The flows use initial routes that collide, and *Flow 2* is started after *Flow 1* has reached steady state. The labels *Detection* and *Response* mark the times when congestion is detected and the flow is rerouted, respectively. The latency between detecting the first packet that causes congestion and sending a congestion notification was between 25–240 µs across runs, and the difference between detection and response is 2.6 ms in this figure. Because the throughput of *Flow 1* never decreases, we can see that the switch had sufficient buffer capacity to avoid dropping a packet during the time in which both flows were active but before a flow was rerouted.

Figure 16 further characterizes the control loop of our system by presenting a CDF of the response latency of both OpenFlow- and ARP-based control, where response latency is defined as the time from when the congestion notification was sent to the time at which a collector sees a packet with the updated MAC address. We see that ARP-based control takes around 2.5 ms to 3.5 ms, while the latency for OpenFlow-based control varies from about 4 ms to 9 ms, with the median control time taking over 7 ms. As in Section 5.3, the majority of this latency can be attributed to switch buffering.

7.3 Traffic Engineering

In this section, we evaluate the effectiveness of a traffic engineering scheme designed to work within our monitoring framework. We analyze the performance of this scheme by comparing it against other traffic engineering schemes under various workloads.

Varying Flow Sizes We first investigate the performance of each traffic engineering scheme under a variety of flow sizes. If PlanckTE can operate on fine-grained timescales, then its performance should track the performance of Optimal for smaller flow sizes. We vary the flow sizes in a stride(8) workload from 50 MiB up to 100 GiB and plot the average throughput achieved by each flow in the network in Figure 17. We use average flow throughput as a metric because fine-grained traffic engineering can impact the initial stages of a flow's throughput and capturing these effects are important.

We note the following trends in the figure. First, PlanckTE can effectively route on small time scales, given its performance relative to Optimal. PlanckTE and Optimal have similar performance for flows as small as 50 MiB, which theoretically can take as little as

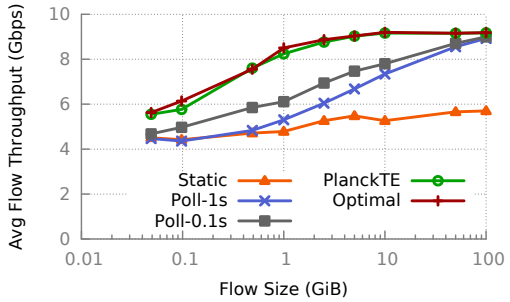

Figure 17: Average flow throughput for varying flow sizes in stride(8), shown at log-scale, for flow sizes ranging from 50 MiB to 100 GiB.

4.2 ms to transfer. Poll-1s can only engineer flows larger than 1 GiB because these are the first flow sizes whose stride(8) workload takes longer than one second to complete, whereas Poll-0.1s can optimize 100 MiB flows. The performance of Poll-1s and Poll-0.1s eventually approach Optimal's performance as the flow size increases. With 100 GiB flows, all schemes but static provide similar performance.

Varying Workloads Figure 14 presents the performance of different traffic engineering schemes for each of the workloads detailed in Section 7.1. As in the previous section, the average throughput of an individual flow is used as the metric. For each workload, the performance of three different flow sizes (100 MiB, 1 GiB and 10 GiB) are presented for the traffic engineering schemes.

We notice the following trends in the figure. First, PlanckTE can closely track the performance of Optimal for all flow sizes. Even under the smallest flow size of 100 MiB, PlanckTE typically comes within 1–4% of Optimal's performance (with the exception of the shuffle workload, where PlanckTE comes within a worst case of 12.3% to Optimal). Second, for both Poll schemes, the performance increases as the flow size increases with Poll-0.1s performing better as expected. Finally, PlanckTE provides benefits over both Poll schemes. The improvement is small for the shuffle workloads where traffic is distributed across many links, and we saw similar trends for Staggered Prob (not shown) where traffic is typically isolated near the network edges. However, in the other workloads, the improvement of PlanckTE over Poll-1s ranges from 24.4% (random) to 53% (stride), and the improvement over Poll-0.1s ranges from 11% (random) to 33% (random bijection).

Lastly, we examine the performance of two different workloads with 100 MiB flow sizes in more detail. Figure 18(a) presents a CDF comparing shuffle completion times for each host in each engineering scheme. The median host completion times are 3.31 seconds for Poll-1s, 3.01 seconds for Poll-0.1s, 2.86 seconds for PlanckTE, and 2.52 seconds for Optimal.

Figure 18(b) contains a CDF of the individual flow throughputs for the stride(8) workload. PlanckTE and Poll-0.1s obtain median flow throughputs of 5.9 Gbps and 4.9 Gbps respectively, and we can see that the performance of PlanckTE closely tracks that of Optimal.

8. RELATED WORK

The work most related to Planck consists of low-latency network measurement techniques and dynamic routing mechanisms for traffic engineering. Traffic engineering is a broad field, but we focus on the recent work predominantly aimed at fine-grained traffic engineering for the data center supported by SDN.

A few recent efforts have focused on how to implement counters and measurement in switches. OpenSketch [44] proposes adding reconfigurable measurement logic to switches and exposing an interface to program it in much the same way that OpenFlow allows for programming forwarding behavior. Other work [25] looks at software-defined counters with a hybrid implementation across the switching ASIC and control plane CPU that might provide better trade-offs in terms of how quickly counters can be read. Additionally, future switching ASICs could provide the ability for fast data-plane only sampling without involving the control plane CPU. Planck provides similar benefits, i.e., comprehensive low-latency measurements, today and this paper uses it to demonstrate how such measurements could be used if and when such new hardware arrives.

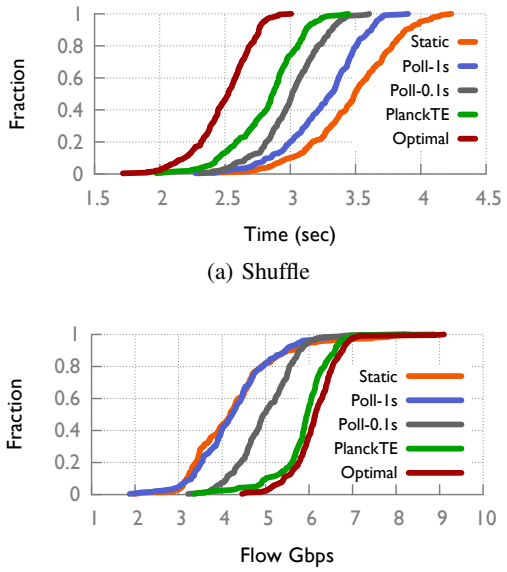

Figure 18: CDFs for 100 MiB flow workloads. The first figure shows host completion times for their shuffle and the second figure shows the flow throughputs for a stride(8) workload.

A separate related line of work looks at improving network measurements in terms of accuracy and coverage [7, 8, 9]. Unlike our work, which focuses on obtaining flow-level throughput measurements at very tight time-scales, they focus on accurately monitoring transit networks at coarser time-scales.

Our traffic engineering application draws inspiration from Hedera [2] and MicroTE [4] and takes a similar approach of measuring the network and rerouting flows based on measurements. However, Hedera and MicroTE use five second and one second control loops, respectively. Further, they both use 1 Gbps links and would need to be noticeably faster to get similar results on 10 Gbps links.

Mahout [5] gets low-latency measurements by modifying end-hosts to detect elephant flows in 1.5–5.5 ms. We expect that using these detections to infer global network state would take notably longer. Planck detects more than just elephant flows without end-host modifications at the same or smaller time-scales.

LocalFlow [31] shows the benefit of a faster traffic engineering control loop in simulation, but does not consider implementing it.

B4 [20] and SWAN [15] monitor and shape traffic at the edge of the network, but these projects operate on the order of seconds or minutes. Similarly, Seawall [35] and EyeQ [21] monitor and shape traffic at the edge, but do not monitor switches.

Additional work has looked at integrating optical links in data centers [10, 40, 42]. This work typically leverages optical networks by building hybrid networks that contain both optical circuit switches and traditional electrical packet switches and needs fast measurements to quickly predict a traffic matrix and then schedule optical

circuits to carry the largest components. Despite this, they can only measure the network every 75–100 ms at minimum. Planck could be used to inform such schemes at much smaller time-scales.

SideCar [36], like Planck, directly attaches servers, called SideCars, to switches with the goal of observing and modifying network behavior. While Planck's collectors are similar, we attach multiple switches to each and use them to monitor and change the behavior of a network of unmodified end-hosts. By contrast, SideCar attaches one server per switch and uses end-host (hypervisor) modifications to invoke them to assist in more general tasks including multicast.

Closely related to Planck is the work of OpenSample [41] and sFlow-RT [33]. OpenSample leverages traditional sFlow [32] samples to detect elephant flows and reroute them to avoid congestion. They also use TCP sequence numbers to improve the accuracy of their samples but do not use them in the presence of a dynamic sampling rate. Additionally, OpenSample operates with a 100 ms control loop and is only evaluated on 10 Mbps links and in simulation.

InMon sFlow-RT [33] also uses sampling to measure the network with the intent of using SDN controllers, but anecdotally takes hundreds of milliseconds to seconds to detect large flows [34].

9. DISCUSSION

9.1 Scalability

While our testbed is insufficient to evaluate Planck at scale, we use measurements and calculations to estimate scalability as switch port counts and the resources required to monitor a network increase.

First, as switch port counts increase, Planck's sampling rate will decrease if only one port is used for samples. However, monitor ports can be added at additional cost to increase the sampling rate.

Second, while collectors are completely independent and thus scale out well, the number of collector instances that can fit on a single server and thus the resources required to monitor a network are worth considering. This depends primarily on the packet processing system used, the number of NICs that can fit in a server, and the number of cores on the server. Our implementation uses netmap [30], and a given collector instance stays below 90% utilization of a single core when processing 10 Gbps at line rate. Recent results from Intel show that it is possible to route 50-60 Gbps per socket [17], so 100 Gbps should be possible on a 2-socket server. We were able to build fourteen 10 Gbps Ethernet ports on a single 2U, 16-core server, so such a server should be able to collect samples from 10-14 switches depending on memory and PCIe bandwidth constrains. In our experiments, we only ran eight collector instances per server due to an unresolved memory bug in netmap for Linux.

Assuming 14 collector instances per server, Planck results in a modest addition to the overall cost for real networks. If 64-port switches are used, with one port being dedicated to monitoring, a full-bisection-bandwidth $k = 62$ three-level fat-tree can be built to support 59,582 hosts from 4,805 switches, which would require 344 collectors, resulting in about 0.58% additional machines. Other topologies that use fewer switches per host, e.g., Jellyfish [37], would require many fewer collectors. For example, a full-bisection-bandwidth Jellyfish with the same number of hosts requires only 3,505 switches and thus only 251 collectors, representing 0.42% additional machines. Using a monitor port also causes the network to support a smaller number of hosts for a given number of switches. For the same number of switches, a fat-tree with monitor ports only supports 1.4% fewer hosts than without monitor ports, and a Jellyfish supports 5.5% fewer hosts than without monitor ports.

Given the performance gains that Planck can offer when coupled with traffic engineering, its possible that networks with lower bisection bandwidths could still see better performance. By tolerating a lower bisection bandwidth, networks could recover extra ports to add more hosts.

Because the volume of network events is far smaller than the volume of samples, we expect that a reasonable network controller will be able to handle the notifications produced by this number of collectors, but we leave that evaluation to future work.

9.2 Implications for Future Switch Design

In building Planck, we have encountered several challenges which we feel point to opportunities to improve the design and implementation of switches and their firmware in the future.

Data Plane Sampling Support for sampling in the data plane to remove the control plane bottleneck is the most obvious addition that future switches could make to enable Planck-like functionality. It would enable much higher sampling rates while also maintaining the metadata which Planck is forced to recover including input port, output port, and sampling rate.

Sampling Rate vs. Rate of Samples Traditional sampling-based network monitoring [32] allows a user (or monitoring system) to set a *sampling rate* where statistically one in N packets are sampled. While this is easy to configure and understand, we have found that it causes suboptimal trade-offs. The sampling rate can cause a switch to exceed the rate of samples it can actually send, making the sampling rate inaccurate. To avoid this problem, sampling rates must be set conservatively so that even with high traffic volumes, the switches do not exceed their sampling rate. The result is that when there are low traffic volumes, very few samples are gathered.

Instead, we propose that future sampling-based network monitoring center around a desired *rate of samples* and switches should vary their sampling rates to approximate this rate. This is useful not just to avoid overrunning switch capabilities, but also to match the capacity of the system processing the samples. Planck does this by constraining samples to the link speed of the monitoring port, but future switches should provide ways to do this for arbitrary rates.

Minimized sample buffering One of the biggest challenges we found in Planck was that the oversubscribed monitor ports became congested and samples were buffered. This buffering both increases the latency to receive samples and uses buffer space that the rest of the switch could be using for burst tolerance. Reducing the buffering for samples to a minimum would eliminate both of these issues.

Reducing the buffer space allocated to a monitor port should be possible with simple firmware changes to existing switches.

Clear Sampling Model Our experiments have shown that oversubscribed monitor ports don't exhibit an obvious model that explains the sampling rate across flows or ports. This means that we can only infer the sampling rate for traffic with sequence numbers. Having a clear model for what traffic will be sampled at what rate under what circumstances, especially given that we believe sampling rates should be dynamic, will be essential.

Preferential Sampling of Special Traffic Certain packets are more important than others. For instance, packets with TCP SYN, FIN, and RST flags mark the beginning and end of flows. Sampling these packets at a higher rate, perhaps even sampling all of them, would aid in providing accurate measurements and faster knowledge of these network events.

We had hoped to achieve this effect by matching on these flags and using OpenFlow to put these packets in a higher priority queue on the monitor ports, but OpenFlow does not currently allow for matching on TCP flags. However, types of traffic OpenFlow can match on could be given higher priority sampling with this method.

Further, it is important to limit what fraction of the total samples are allowed to be sampled from higher priority packet classes to avoid allowing an attacker to suppress all normal samples by sending a high rate of traffic in the priority classes, e.g., a SYN flood.

In-switch Collectors Lastly, while we believe there will always be places in which the flexibility of sampling-based measurement will be useful, there may be common information and events that the switch could produce based on it's own samples without needing to commit network resources to the samples themselves. In effect, this would be like running the collector for a switch on the switch itself.

9.3 Future Work

We believe that there is significant future work to be done with Planck. We have shown many ideas about how we build networks, certainly software-defined networks need to be rethought if we want to capitalize on millisecond-scale (or faster) measurements. For example, with careful engineering Planck can allow a network to react to congestion faster than switch buffers fill meaning TCP would not see losses from congestion in the common case.

Going forward, we would also like to turn Planck into a more extensible measurement platform and define the relevant APIs to plug modules into key places. For example, many of the techniques described in OpenSketch [44] could be implemented as streaming operators on the samples Planck receives.

10. CONCLUSION

This paper presented Planck, a novel network measurement system that uses oversubscribed port mirroring to provide measurements every 4.2 ms–7.2 ms—*more than an order of magnitude (11–18x) improvement* over the current state-of-the-art (and up to 291x if switch firmware allowed buffering to be disabled on mirror ports). To demonstrate that this increased speed translates to improved performance, we built a traffic engineering application designed to operate at similar speeds which is able to detect congestion and reroute flows in about 3 milliseconds. Doing so provides near-optimal throughput even for small flow sizes, e.g., 50 MiB, and even at 10 Gbps link speeds. Further, this granularity of measurement radically changes how we should be thinking about network control loops and how we build SDN controllers.

Acknowledgements: We thank our shepherd Amin Vahdat and the anonymous reviewers for their helpful comments. We also thank Andrew Ferguson for his feedback and discussions. Brent Stephens is supported by an IBM Fellowship. Jeff Rasley is supported by an NSF Graduate Research Fellowship (DGE-1058262).

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[2] M. Al-fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

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

[4] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *CoNEXT*, 2011.

[5] A. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-Overhead Datacenter Traffic Management using End-Host-Based Elephant Detection. In *INFOCOM*, 2011.

[6] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling Flow Management for High-Performance Networks. In *SIGCOMM*, 2011.

[7] N. Duffield, C. Lund, and M. Thorup. Learn More, Sample Less: Control of Volume and Variance in Network Measurement. *IEEE Transactions on Information Theory*, 51(5):1756–1775, 2005.

[8] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a Better NetFlow. In *SIGCOMM*, 2004.

[9] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting. In *SIGCOMM*, 2002.

[10] 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.

[11] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *SIGCOMM*, 2013.

[12] Floodlight OpenFlow Controller. http://floodlight.openflowhub.org/.

[13] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *SIGCOMM*, 2009.

[14] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall. Augmenting Data Center Networks with Multi-Gigabit Wireless Links. In *SIGCOMM*, 2011.

[15] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving High Utilization with Software-driven WAN. In *SIGCOMM*, 2013.

[16] IBM BNT RackSwitch G8264. http://www.redbooks.ibm.com/abstracts/tips0815.html.

[17] Intel® Data Plane Development Kit (Intel® DPDK) Overview Packet Processing on Intel® Architecture. http://goo.gl/qdg3rZ, December 2012.

[18] Intel® DPDK: Data Plane Development Kit. http://www.dpdk.org.

[19] V. Jacobson. Van Jacobson's Network Channels. http://lwn.net/Articles/169961/, January 2006.

[20] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-deployed Software Defined WAN. In *SIGCOMM*, 2013.

[21] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*, 2013.

[22] S. Kandula, S. Sengupta, A. Greenberg, and P. Patel. The Nature of Datacenter Traffic: Measurements and Analysis. In *IMC*, 2009.

[23] R. Kapoor, A. C. Snoeren, G. M. Voelker, and G. Porter. Bullet Trains: A Study of NIC Burst Behavior at Microsecond Timescales. In *CoNEXT*, 2013.

[24] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ToCS*, 18(3):263–297, August 2000.

[25] J. C. Mogul and P. Congdon. Hey, You Darned Counters! Get Off My ASIC! In *HotSDN*, 2012.

[26] Openflow-switch. https://www.opennetworking.org/standards/openflow-switch.

[27] Libpcap File Format. http://wiki.wireshark.org/Development/LibpcapFileFormat.

[28] PF_RING: High-speed Packet Capture, Filtering and Analysis. http://www.ntop.org/products/pf_ring/.

[29] P. Phaal and S. Panchen. Packet Sampling Basics. http://www.sflow.org/packetSamplingBasics/index.htm.

[30] L. Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC*, 2012.

[31] S. Sen, D. Shue, S. Ihm, and M. J. Freedman. Scalable, Optimal Flow Routing in Datacenters via Local Link Balancing. In *CoNEXT*, 2013.

[32] sFlow. http://sflow.org/about/index.php.

[33] sFlow-RT. http://inmon.com/products/sFlow-RT.php.

[34] Large Flow Detection Script. http://blog.sflow.com/2013/06/large-flow-detection-script.html, June 2013.

[35] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the Data Center Network. In *NSDI*, 2011.

[36] A. Shieh, S. Kandula, and E. G. Sirer. SideCar: Building Programmable Datacenter Networks without Programmable Switches. In *HotNets*, 2010.

[37] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking Data Centers Randomly. In *NSDI*, 2012.

[38] Version 2 of the Protocol Operations for the Simple Network Management Protocol (SNMP). RFC 3416. http://www.ietf.org/rfc/rfc3416.txt.

[39] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter. PAST: Scalable Ethernet for Data Centers. In *CoNEXT*, 2012.

[40] G. P. R. Strong, N. Farrington, A. Forencich, P. Chen-Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat. Integrating Microsecond Circuit Switching into the Data Center. In *SIGCOMM*, 2013.

[41] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter. OpenSample: A Low-latency, Sampling-based Measurement Platform for SDN. In *ICDCS*, 2014.

[42] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. S. E. Ng, M. Kozuch, and M. Ryan. c-Through: Part-time Optics in Data Centers. In *SIGCOMM*, 2010.

[43] Wireshark. http://www.wireshark.org/.

[44] M. Yu, L. Jose, and R. Miao. Software Defined Traffic Measurement with OpenSketch. In *NSDI*, 2013.