High-Fidelity Switch Models for Software-Defined Network Emulation

Danny Yuxing Huang, Kenneth Yocum, and Alex C. Snoeren

Department of Computer Science and Engineering
University of California, San Diego
{dhuang, kyocum, snoeren}@cs.ucsd.edu

ABSTRACT

Software defined networks (SDNs) depart from traditional network architectures by explicitly allowing third-party software access to the network's control plane. Thus, SDN protocols such as OpenFlow give network operators the ability to innovate by authoring or buying network controller software independent of the hardware. However, this split design can make planning and designing large SDNs even more challenging than traditional networks. While existing network emulators allow operators to ascertain the behavior of traditional networks when subjected to a given workload, we find that current approaches fail to account for significant vendor-specific artifacts in the SDN switch control path. We benchmark OpenFlow-enabled switches from three vendors and illustrate how differences in their implementation dramatically impact latency and throughput. We present a measurement methodology and emulator extension to reproduce these control-path performance artifacts, restoring the fidelity of emulation.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: [Modeling Techniques; Measurement Techniques]; C.2 [**Computer Communication Networks**]:

Keywords

SDN; Software-Defined Networking; OpenFlow; Modeling

1. INTRODUCTION

The performance of applications and services running in a network is sensitive to changing configurations, protocols, software, or hardware. Thus network operators and architects often employ a variety of tools to establish the performance and correctness of their proposed software and hardware designs before deployment. Network emulation has been an important tool for this task, allowing network operators to deploy the same end-system applications and protocols on the emulated network as on the real hardware [8, 13, 15]. In these environments, the interior of the network comprises a real-time link emulator that faithfully routes and switches packets according to a given specification, accurately accounting for details such as buffer lengths and AQM settings. In general, these emulators do not attempt to reproduce vendor-specific details of particular pieces of hardware, as end-to-end performance is typically dominated by implementation-agnostic features like routing (choice of links), link contention, and end-system protocols (application or transport-layer).

However, software defined networks (SDNs) challenge this assumption. In traditional switching hardware the control plane rarely affects data plane performance; it is reasonable to assume the switch forwards between ports at line rate for all flows. In contrast, an OpenFlow-enabled switch outsources its control plane functions to an external controller. New flows require a set-up process that typically involves the controller and modifications to the internal hardware flow tables (e.g., ternary content-addressable memory, or TCAM). Moreover, this process depends upon the particular hardware and software architecture of the switch, including its CPU, TCAM type and size, and firmware. Thus, while the maturity of traditional network switch designs have allowed emulators to make simplifying assumptions, the relatively youthful state of SDN hardware implies that network emulators may not be able to make similar generalizations.

We argue that high-fidelity, vendor-specific emulation of SDN (specifically, OpenFlow) switches is required to provide accurate and repeatable experiments. In particular, we show that architectural choices made by switch vendors impact the end-to-end performance of latency-sensitive applications, such as webpage load times, database transactions, and some scientific computations. State-of-the-art SDN emulation depends upon the use of multiple software switches, such as Open vSwitch (OVS), to emulate a network of OpenFlow devices [7, 11]. While these systems allow network operators to test innovative OpenFlow controller software, our results show that, in general, it is impossible for an operator to use these emulators to determine how applications will perform over a network consisting of OpenFlow switches produced by a particular (set of) vendor(s).

For instance, different vendors employ distinct TCAM technologies that affect the size of the hardware flow table (the maximum number of entries may even change based on rule complexity [2]) and the rate at which rules may be installed. Switch firmware may buffer and/or delay flow installations, or even employ a software flow table with various policies regarding how to promote flows to the hardware table. For example, by default OVS allows 65,000 rules with nearly instantaneous rule installation; in contrast, a Fulcrum Monaco switch in our testbed has a 511-entry flow table that can insert at most 42 new rules a second.

To characterize how these differences affect application performance, we perform an empirical investigation of three vendors' OpenFlow switches. We find two general ways in which switch

HotSDN'13, August 16, 2013, Hong Kong, China.

Switch	Firmware	CPU	Link	Flow table entries	Flow setup rate	Software table
HP ProCurve J9451A	K.15.06.5008	666 MHz	1 Gbps	1500	40 flows/sec	Yes
Fulcrum Monaco Reference	Open vSwitch 1.0.1	792 MHz	10 Gbps	511	42 flows/sec	No
Quanta LB4G	indigo-1.0-web-lb4g-rc3	825 MHz	1 Gbps	1912	38 flows/sec	Yes
Open vSwitch on Xeon X3210	version 1.7.0	2.13 GHz	1 Gbps	65,000	408 flows/sec	Only

Table 1: The three vendor switches studied in this work, as well as our host-based OVS SDN emulator.

design affects network performance: control path delays and flow table designs. Section 2 describes how these two classes of artifacts impact network performance and quantifies the error introduced by unmodified OVS emulations. Section 3 describes a methodology to "thumbprint" important hardware switch behavior in a black-box fashion and replicate it in an emulator. Even with our simple model, we find that an appropriately calibrated emulation infrastructure can approximate the behavior of the switches we study.

2. MOTIVATION

We focus on two particular switch artifacts: control path delays and flow table characteristics. Our treatment is not exhaustive; there are likely to be other controller/switch interactions that highlight vendor differences beyond those discussed here. For instance, OpenFlow's pull-based flow monitoring is a well-known source of control plane overhead, directly affecting rule installation rates [4].

We study performance differences across switch designs from three vendors listed in Table 1. This particular set of switches represents distinct vendors, chipsets, link rates, OpenFlow implementations, and hardware/software flow table sizes. We also perform our experiments on a host-based switch running an in-kernel Open vSwitch module, a setup similar to that used in Mininet to emulate SDN switches [7]. As a host-based switch, OVS has significant advantages in terms of management CPU and flow table memory (although our experiments switch packets between only two ports).

2.1 Variations in control path delays

We begin by studying the impact of control path delays, i.e. the sequence of events that occurs when a new flow arrives that leads to the installation of a new forwarding rule in the switch. In general, OpenFlow switches have two forwarding modes. For the packets of flows that match existing entries in the hardware flow table, they forward at line rate. Packets from unmatched flows, on the other hand, are sent along the control path, which is often orders-of-magnitude slower.

This control path consists of three steps: First, the switch moves (some of) the packet from the ingress line card to the management CPU and sends an OpenFlow `packet_in` event to the controller. The controller subsequently builds an appropriate packet matching rule and sends a `flow_mod` event back to the switch. The switch processes the `flow_mod` event by installing the new rule into the forwarding table and then either sending the packet out the designated egress port or dropping it, in accordance with the newly installed rule.

The first and last steps are potentially influenced by the hardware and firmware in the switch, independent of the delays introduced by the controller. To test whether or not this is the case with current switches, we compare the performance of Redis, a popular open-source key-value store, when run across single-switch networks consisting of each of our three switches as well as an OVS emulator. Redis allows us to easily vary the number and size of the TCP connections it employs by changing the size of the client pool and requested value sizes, respectively.

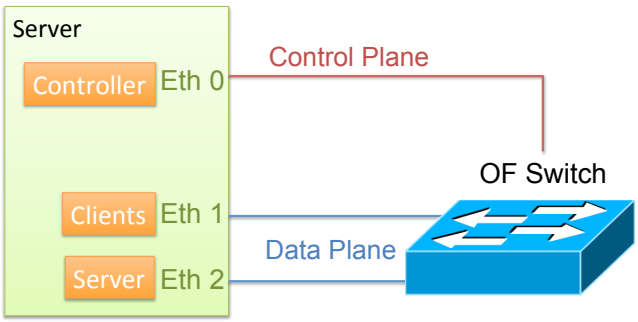

Figure 1: Experimental testbed topology.

Figure 1 illustrates the testbed topology used in each of our experiments. The server has three 1-Gbps ports that are all connected to the OpenFlow switch under test. The first, *eth0*, is connected to the switch's control port. A POX [12] controller running the default L2-learning module listens on this Ethernet port. By default, it installs an exact-match rule for every new incoming flow. The next two server ports are connected to data ports on the switch. To prevent the data traffic from bypassing the switch, we place *eth1* and *eth2* in separate Linux network name spaces. Experiments with the OVS-based emulator run on the same machine; the emulated links are gated to 1 Gbps using Linux Traffic Control (TC). In no experiment (even those using OVS) was the host CPU utilization greater than 80%.

Our first experiment studies control-path delays by starting Redis clients on distinct source ports every 50 ms. Each client requests a single 64-byte value and we record the query completion time. Each request requires the switch to invoke the control path described above. The experiment lasts two minutes; we report results from the last minute to capture steady-state behavior. The switches under test belong to researchers from a number of research institutions. Each switch is connected to a local test server according to the topology described above, but the physical switch and server are located at the respective institution; we conduct experiments remotely via SSH. All of the servers are similarly provisioned; we verify that all Redis queries complete well within 1 ms when the required flow rules have been installed in advance.

Figure 2 shows the CDF of query completion times for each switch. The first observation is that all of the switches exhibit distinct delay distributions that are almost entirely due to variations in control path delays. Here OVS has an obvious advantage in terms of management CPU and flow table size and its Redis queries complete considerably faster than any of the hardware switches (including the 10-Gbps Monaco). Thus, latency-sensitive applications might achieve good performance using an OVS-based emulator but suffer delays when deployed on actual hardware switches. Moreover, significant differences exist even between the two 1-Gbps switches.

Hence, we argue that for workloads (and/or controller software) that necessitate frequent flow set up, OVS-based emulations must

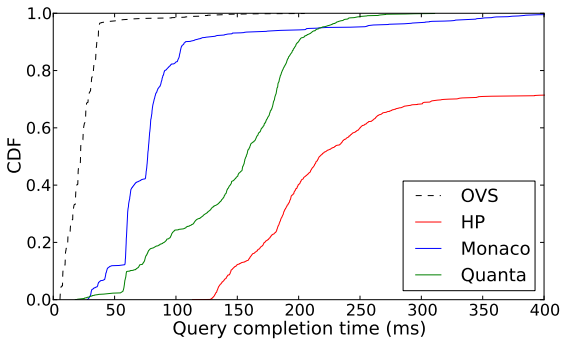

Figure 2: Query completion times for Redis clients.

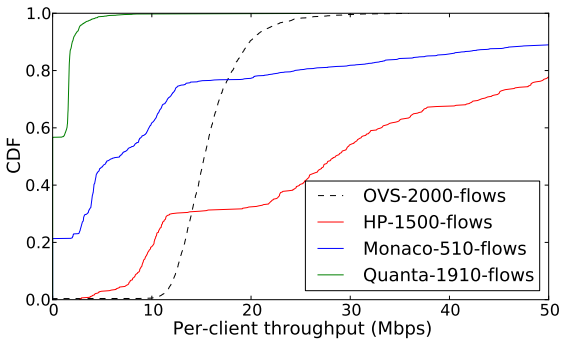

Figure 3: Throughput of concurrent Redis requests. Active flow counts fit within each switch's flow table.

account for individual switch artifacts to accurately predict application performance. Section 3 discusses our methodology for quantifying this effect and reproducing it in an OVS-based emulator.

2.2 Impact of flow table design

This section explores how differences in flow table hardware and management software affect the forwarding performance of OpenFlow switches. The flow table not only defines the maximum working set of flows that can be concurrently forwarded at line rate, but it also determines the ability of an OpenFlow controller to modify and update those rules. Most often, they are implemented in TCAMs, which combine the speed of exact-match tables with the flexibility of wild-card rules that match many header fields. However, TCAMs are relatively complicated, expensive, and power hungry; economics often limits the physical number of entries available. Moreover, TCAM rule insertion algorithms often consider the current set of rules, and may rewrite them entirely, which can gate the maximum allowed rate of hardware rule insertions.

In general, emulating the correct flow table size is important since an OpenFlow network design should ensure that the flow working set (concurrent set of flows) at any switch fits within its flow table. Flows that remain in the hardware flow table will see line rate forwarding; otherwise, they experience zero effective capacity. Given this zero/one property, it is easy to emulate the performance impacts of table size. One could develop a controller module that keeps track of the current size of the flow table. It behaves normally when the flow table size is within some threshold; otherwise, the controller stops issuing new `flow_mod` events to mimic a full flow table.

Even if the flow working set is sufficiently small, however, the presence of shorter flows can still create flow table churn. This operating regime (by one account 90% of flows in a data center send less than 10 KB [1]) is affected by other characteristics of flow table management. Four that we observe are buffering flow installation (`flow_mod`) events, the use of a software flow table, automatic rule propagation between the software and hardware tables, and hardware rule timeouts. For example the HP and Quanta switches both use software tables, but manage them differently.

While the Quanta switch only uses the software table once the hardware table is full, the HP switch's management is more complicated. If `flow_mod` events arrive more rapidly than 8 per second, rules might be placed into the software table, which, like the Quanta switch, forwards flows at only a few megabits per second. In fact, the firmware buffers new rules that may have arrived too fast for the hardware table to handle. Whether a rule enters the hardware or software table is a function of the event inter-arrival rate and the

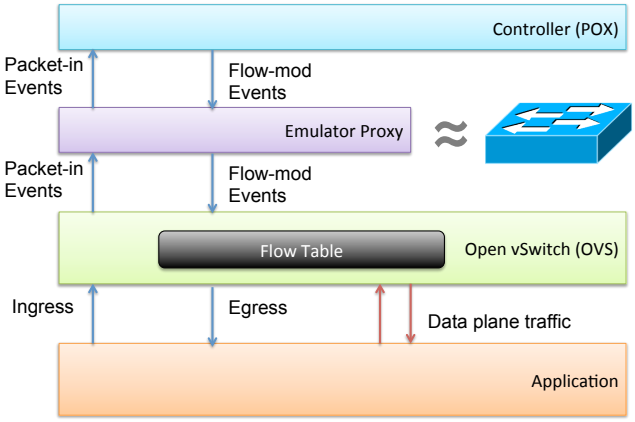

Figure 4: The emulator manipulates OpenFlow events between the controller and OVS in order to approximate the performance of a given hardware switch.

current capacity of the hardware table. A further complicating factor is a rule-promotion engine that migrates rules from the software to the hardware table.

While a full exploration of all the interactions (and attempts to emulate them) is beyond the scope of this work, we use a flow-churn experiment to illustrate the impact of these differences. Using the same set-up as shown in Figure 1, we start Redis clients, asynchronously, every 10 ms, each requesting a 5-MB value from the server. Throughout the experiment, we ensure that the maximum number of concurrent flows does not exceed the size of the hardware flow table for each switch (2000, 1910, 1500, and 510 for OVS, Quanta, HP, and Monaco, respectively). We set the idle flow timeout to ten seconds for each switch.

Figure 3 plots the distribution of per-client throughputs. Even though the number of active flows does not exceed the flow table size, each switch behaves in a markedly different way. The Monaco and Quanta switches drop flows while waiting for existing entries to time out. In contrast, the HP switch never rejects flows as rules are spread across hardware and software tables; the HP's hardware table is never full during the experiment. Although the Quanta switch also has a software table, it is not used to buffer up excess rules like the HP switch. Instead, the controller receives a failure whenever the hardware table is full.

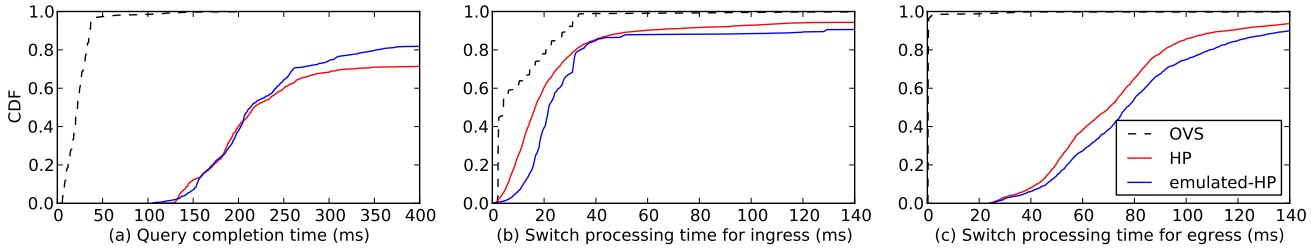

Figure 5: Performance of the HP switch (in red) as compared to an unmodified OVS (dashed) and our HP switch emulator (blue). (a) Distribution of query completion times for Redis clients. (b) Distribution of switch processing time for ingress packets along the control path. (c) Distribution of switch processing time for egress packets.

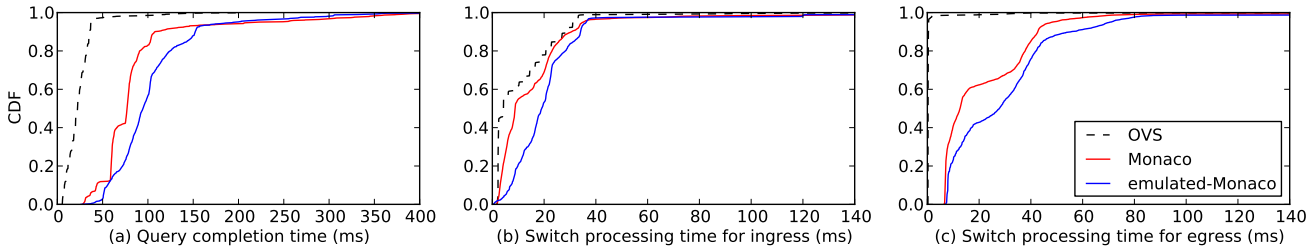

Figure 6: Performance of the Monaco switch (red) and our Monaco switch emulator (blue); OVS repeated from Figure 5 for reference.

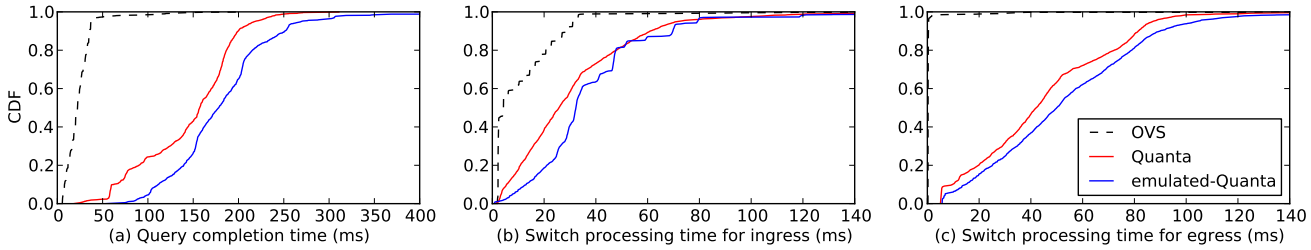

Figure 7: Performance of the Quanta switch (red) and our Quanta switch emulator (blue); OVS repeated from Figure 5 for reference.

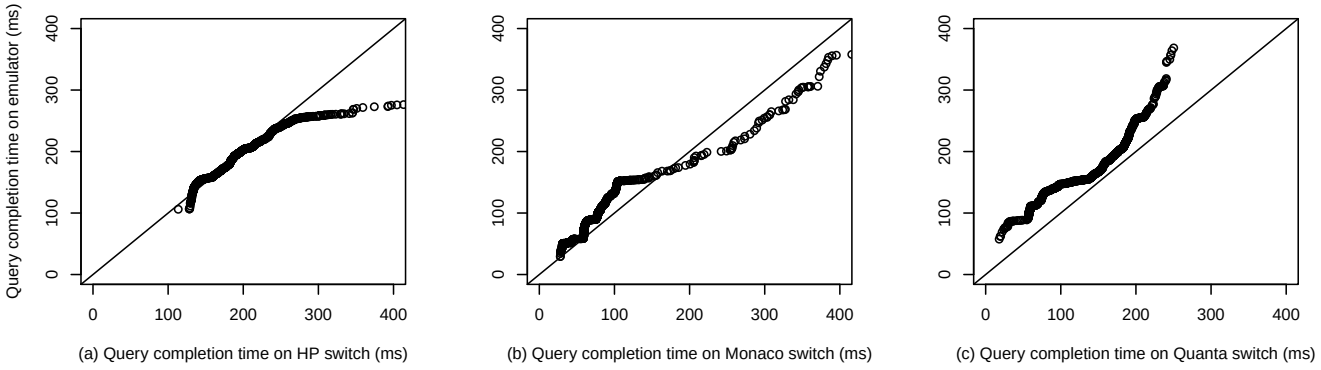

Figure 8: QQ-plots compare the emulator and hardware switch distributions of Redis query completion times.

Like control path delays, OVS emulation is a poor approximation for any of the emergent behaviors that arise from vendors' flow table management schemes. We leave measuring and reproducing these effects as future work.

3. EMULATING CONTROL PATH DELAY

This section examines how to measure and reproduce specific switch control path delays in an OVS-based emulator. Section 3.1 introduces a methodology for measuring the components of control path delay affected by switch design. We then design an emulator (Section 3.2) that incorporates these statistics. Section 3.3 shows that even this basic "thumbprint" and replicate strategy allows us to closely approximate control path delays for our target switches.

3.1 Measuring control path delays

As discussed in Section 2.1, OpenFlow switches introduce delays along the control path in two places:

Ingress packet processing: This delay occurs between the arrival of the first packet of a new flow and the `packet_in` event sent to the controller.

Egress packet processing: This delay occurs between the arrival of a `flow_mod` event and sending the buffered packet through the egress port.

We find these delays to be markedly different across vendors, as ingress and egress events must leverage the hardware flow table and switch CPU. For each switch we capture its specific ingress and egress delay distributions using the set-up shown in Figure 1. This process uses the same experimental procedure as described in Section 2.1 but with two additions. First, we modify the L2-learning POX module, such that it records the times at which `packet_in` events are received and `packet_out` events are sent. Second, `tcpdump` captures time stamps of all Redis packets on both *eth1* and *eth2*. Since there is one frame of reference for all time stamps, it is straightforward to calculate the ingress and egress delay distributions.

Figure 5(a) shows the distribution of 64-byte Redis query completion times for the HP ProCurve switch (in red) and OVS (dashed black), both reproduced from Figure 2. The other two subfigures, 5(b) and 5(c), show the distributions of the component ingress and egress delays observed during the experiment. Compared with OVS, the HP switch introduces significantly more latency and loss (not shown) along the control path. As a result, Redis clients complete faster on OVS than on HP. Figures 6 and 7 plot the results for the remaining two hardware switches. The differences between the ingress and egress delay distributions account for the cross-vendor variations in Redis performance shown in Figure 2.

3.2 Control path emulation with OVS

Like other OpenFlow emulation architectures [11], we use OVS to emulate individual OpenFlow switches. However to reproduce the control path delays observed in the prior section we design a *proxy-assisted* OVS emulator. As shown in Figure 4, an emulation proxy intercepts OpenFlow control traffic between the controller and OVS. Since OVS delays control packets differently from hardware switches, the emulator must shape the intercepted packets in a way that would resemble the ingress and egress delay distribution of the target hardware switch.

However, like the hardware switches, OVS exhibits its own control path delays. Thus, we must adjust the emulation in real-time to account for varying OVS delays. As shown in Figure 5(b), half of the `packet_in` events experience 5–40 ms of delay in OVS. If t_{OVS} is the observed OVS ingress delay and t_{TS} is the a desired delay taken from the target switch's ingress distribution, then the emulator needs to introduce an additional delay of $\max(0, t_{TS} - t_{OVS})$.

We use inverse transform sampling [5] to compute the desired delay. The OVS emulator measures t_{OVS} as the time elapsed between the ingress packet and the packet-in event. We compute the quantile, q, of t_{OVS} by looking up the value in the previously measured ingress delay distribution of OVS (i.e., Figure 5(b)). This process has the effect of uniformly sampling the quantile, q. The emulator then uses q to index the ingress distribution of the target switch to obtain a uniformly sampled ingress delay: t_{TS}. This process minimizes the chance that $t_{TS} < t_{OVS}$. The egress delay sample is calculated in a similar fashion from the respective egress distributions.

3.3 Evaluation

Having established that different OpenFlow switches exhibit different behaviors, here our goal is to determine if our proxy emulator can reproduce those performance artifacts successfully. To do so, we attempt to reproduce the Redis query performance distributions found in Figure 2 using our proxy-assisted OVS emulator.

Figure 8 shows three QQ plots that compare the distributions of Redis performance on our proxy-assisted emulation versus the actual hardware. If the distributions are identical, all points will line up on the $y = x$ diagonal. While far from perfect, in each case the emulator is able to reproduce the general shape of the distribution—the general "slope" of each QQ plot is close to 1. The means differ by the following amounts with 95% confidence intervals: 202 ± 72 ms for HP, 17 ± 5 ms for Monaco, and 63 ± 18 ms for Quanta.

Our current emulator is just a proof of concept; there are still systematic gaps between emulation and reality. For example, the QQ plot is above the $y = x$ diagonal in Figure 8(c). Although the emulated distribution resembles the original one, the emulator may have failed to account for system- or OVS-specific overhead that caused the emulation to run slower. As shown in Figure 7, the emulated distribution (blue line) is almost always below the the actual distribution (red line). Similar differences can be observed in Figure 6, where our emulated distribution is mostly slower than that on the Monaco switch. Because the gap between emulation and reality is relatively consistent, it is trivial to subtract a constant delay from the emulated distribution. This adjustment shifts the QQ plots in Figures 8(b) and (c) downwards to better align with the diagonal (not shown).

In contrast to the Monaco (8(b)) and Quanta (8(c)) experiments, the HP emulation (8(a)) accurately reproduces only the lower 65% of the distribution; it is less accurate for the upper quantiles. The blue curve in Figure 5(a) plots the CDF of the distribution. Although proxy emulation does better than plain OVS, it remains slightly faster than the true HP hardware. Unfortunately, examining the control path's delay profile does not provide many clues. As illustrated in Figures 5(b) and (c), the emulated ingress and egress delay distributions along the control path (blue) match the shape of the original distributions (red).

Evidently, there are additional artifacts that the emulator fails to capture. As `flow_mod` events arrive at the switch faster than the hardware table can handle, we suspect that some flows are matched in the software table, which causes the poor performance on the hardware switch. We are working to extend the emulator to capture more of these flow table artifacts. Eventually, by combining with the current emulator, we hope to be able to emulate the HP switch more accurately.

In summary, we are able to emulate hardware OpenFlow switches significantly better than OVS, although substantial dif-

ferences remain. For the HP switch, replicating the software table properties will reduce this gap. For the Monaco and Quanta switches, we can further optimize the emulator to lower the overhead. Moreover, we can develop a tool that automatically analyzes and tunes the distributions of control path delays—for instance, by subtracting some constant—to improve accuracy.

4. RELATED WORK

Mininet is a popular option for emulating SDN-based networks [7, 11]. To emulate an OpenFlow network, it houses OVS instances on a single server on a virtual network topology that runs real application traffic. As we have shown, however, this approach lacks fidelity. The authors of OFLOPS recognize vendor-specific variations as we do [14]. Their benchmark framework injects various workloads of OpenFlow commands into switches. Based on the output, the authors have observed that there are significant differences in how various hardware switches process OpenFlow events, further underscoring the need to capture such variations in emulation.

We focus primarily on vendor-specific variations in the control plane. Data planes can also be different across vendors. To help predict the application performance, commercially available tools, such as those provided by Ixia and Spirent, test the data plane with various workloads. Other work analyzes the output packet traces and constructs fine-grained queuing models of switches [9, 10]. This work is complimentary to and would combine well with ours.

A related effort in modeling OpenFlow switches is the NICE project [3], which claims that OVS contains too much state and incurs unnecessary non-determinism. As a result, they implement only the key features of an OpenFlow switch, including such essential tasks as `flow_mod` and `packet_out`. We similarly simplify our abstraction of OpenFlow switches and model parameters that vary across vendors.

5. CONCLUSION

In this work, we have shown OVS is a poor approximation of real OpenFlow hardware, and that simple steps can dramatically increase the accuracy of emulation. At the moment, we are still trying to address two challenges:

1. Changing the experimental workload may affect the software models of switches. By varying the workload, we hope to construct more accurate models that can predict emergent behaviors of the switch under complex network environments.

2. CPU poses a major performance bottleneck for OVS. In cases where the load on OVS is high or where the hardware switch is fast, the OVS-based emulator is likely to be slower than the hardware. We are exploring the use of time dilation [6] to slow down the end-hosts' traffic, thus preventing the CPU from throttling OVS.

When both of these challenges are solved, we will be able to further shrink the gap between the emulated and actual performance. Eventually, we hope to deploy an emulator that is able to replicate tens or hundreds of OpenFlow switches with high fidelity at scale.

6. ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation through grant CNS-1018808. We would like to thank Marco Canini and Dan Levin for granting us remote access to their Quanta switch, and George Porter for the use of a Fulcrum Monaco switch.

7. REFERENCES

[1] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, pages 267–280, 2010.

[2] Broadcom. Personal communications, March 2013.

[3] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE way to test OpenFlow applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, pages 10–10, 2012.

[4] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM Conference*, pages 254–265, 2011.

[5] L. Devroye. *Non-Uniform Random Variate Generation*, chapter 2.1, pages 27–36. Springer-Verlag, 1986.

[6] D. Gupta, K. V. Vishwanath, M. McNett, A. Vahdat, K. Yocum, A. C. Snoeren, and G. M. Voelker. DieCast: Testing distributed systems with an accurate scale model. *ACM Transactions on Computer Systems*, 29(2):4:1–4:48, May 2011.

[7] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, pages 253–264, 2012.

[8] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale virtualization in the emulab network testbed. In *Proceedings of the USENIX Annual Technical Conference*, pages 113–128, 2008.

[9] N. Hohn, D. Veitch, K. Papagiannaki, and C. Diot. Bridging router performance and queuing theory. In *Proceedings of the ACM SIGMETRICS Conference*, pages 355–366, 2004.

[10] D. Jin, D. Nicol, and M. Caesar. Efficient gigabit Ethernet switch models for large-scale simulation. In *IEEE Workshop on Principles of Advanced and Distributed Simulation*, pages 1 –10, May 2010.

[11] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 19:1–19:6, 2010.

[12] NOXRepo.org. About POX. `http://www.noxrepo.org/pox/about-pox/`.

[13] R. Ricci, J. Duerig, P. Sanaga, D. Gebhardt, M. Hibler, K. Atkinson, J. Zhang, S. Kasera, and J. Lepreau. The FlexLab approach to realistic evaluation of networked systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, pages 15–15, 2007.

[14] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. OFLOPS: An open framework for OpenFlow switch evaluation. In *Proceedings of the 13th International Conference on Passive and Active Measurement*, pages 85–95, 2012.

[15] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, pages 271–284, 2002.