Measuring Control Plane Latency in SDN-enabled Switches

Keqiang He†, Junaid Khalid†, Aaron Gember-Jacobson†, Sourav Das†, Chaithan Prakash†,
Aditya Akella†, Li Erran Li*, Marina Thottan*
†University of Wisconsin-Madison, *Bell Labs

ABSTRACT

Timely interaction between an SDN controller and switches is crucial to many SDN applications—e.g., fast rerouting during link failure and fine-grained traffic engineering in data centers. However, it is not well understood how the control plane in SDN switches impacts these applications. To this end, we conduct a comprehensive measurement study using four types of production SDN switches. Our measurements show that control actions, such as rule installation, have surprisingly high latency, due to both software implementation inefficiencies and fundamental traits of switch hardware.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Network**]: General; C.4 [**Performance of Systems**]: Metrics—*performance measures*

Keywords

Software-defined Networking (SDN); Latency; Measurement

1. INTRODUCTION

Software defined networking (SDN) advocates for the separation of control and data planes in network devices, and provides a logically centralized platform to program data plane state [3, 14]. This has opened the door to rich network control applications that can adapt to changes in network topology or traffic patterns more flexibly and more quickly than legacy control planes [2,6,7,9,10,13,16]. However, to optimally satisfy network objectives, many important control applications require the ability to reprogram data plane state at very fine time-scales. For instance, fine-grained data center traffic engineering requires routes to be set up within a few hundred milliseconds to leverage short-term traffic predictability [2]. Similarly, setting up routes in cellular networks (when a device becomes active, or during a handoff) must complete within ∼30-40ms to ensure users can interact with Web services in a timely fashion [10].

Timeliness is determined by: (1) the speed of control programs, (2) the latency to/from the logically central controller, and (3) the responsiveness of network switches in interacting with the controller—specifically, in generating the necessary input messages for control programs, and in modifying forwarding state as dictated by the programs. Robust control software design and advances in distributed controllers [12] have helped overcome the first two issues. However, with the focus in current/upcoming generations of SDN switches being on the flexibility benefits of SDN w.r.t. legacy technology, the third issue has not gained much attention. Thus, it is unknown whether SDN can provide sufficiently responsive control to support the aforementioned applications.

To this end, we present a thorough systematic exploration of latencies in four types of production SDN switches from three different vendors—Broadcom, Intel, and IBM—using a variety of workloads. We investigate the relationship between switch design and observed latencies using greybox probes and feedback from vendors. Key highlights from our measurements are as follows: (1) We find that *inbound latency*, i.e., the latency involved in the switch generating events (e.g., when a flow is seen for the first time) can be high—8 ms per packet on average on Intel. The delay is particularly high whenever the switch is simultaneously processing forwarding rules received from the controller. (2) We find that *outbound latency*, i.e., the latency involved in the switch installing/modifying/deleting forwarding rules provided by control applications, is also high—3ms and 30ms per rule for insertion and modification, respectively, in Broadcom. The latency crucially depends on the priority patterns of both the rules being inserted and those already in a switch's table. (3) We find significant differences in latency trends across switches with different chipsets and firmware, pointing to different internal optimizations.

These observations highlight two important gaps in current switch designs. First, some of our findings show that poor switch software design contributes significantly to observed latencies (affirming [5,8,17]). We believe near term work will address these issues; our measurements with an early release of Broadcom's OpenFlow 1.3 firmware exemplify this. More crucially, our measurements reveal latencies that appear to be fundamentally rooted in hardware design: e.g., rules must be organized in switch hardware tables in priority order, and simultaneous switch control actions must contend for limited bus bandwidth between a switch's CPU and ASIC. Unless the hardware significantly changes—and our first-of-a-kind in-depth measurement study may engender such changes—we believe these latencies will manifest even in next generation switches.

2. BACKGROUND

Instead of running a complex control plane on each switch, SDN delegates network control to external applications running on a logically central controller. Applications determine the routes traffic should take, and they instruct the controller to update switches with the appropriate forwarding state. These decisions may be based on data packets that are received by switches and sent to the con-

SOSR 2015, June 17–18, 2015, Santa Clara, CA, USA.

http://dx.doi.org/10.1145/2774993.2775069.

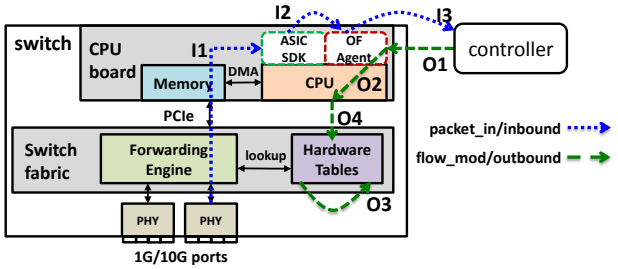


Figure 1: Schematic of an OpenFlow switch. We also show the factors contributing to inbound and outbound latency

troller. Such packet events and state update operations are enabled by OpenFlow [14]—a standard API implemented by switches to facilitate communication with the controller. Although SDN moves control plane logic from switches to a central controller, switches must still perform several steps to generate packet events and update forwarding state. We describe these steps below.

Packet Arrival. When a packet arrives, the switch ASIC first performs a lookup in the switch's hardware forwarding tables. If a match is found, the packet is forwarded at line rate. Otherwise the following steps occur (Figure 1): (I1) The ASIC sends the packet to the switch's CPU via the PCIe bus. (I2) An OS interrupt is raised, at which point the ASIC SDK gets the packet and dispatches it to the switch-side OpenFlow agent. (I3) The agent wakes up, processes the packet, and sends to the controller a *packet_in* message containing metadata and the first 128B of the packet. All three steps, I1–I3, can impact the latency in generating a *packet_in* message. We categorize this as *inbound latency*, since the controller receives the message as input.

Forwarding Table Updates. The controller sends *flow_mod* messages to update a switch's forwarding tables. A switch takes the following steps to handle a *flow_mod* (Figure 1): (O1) The OpenFlow agent running on the CPU parses the message. (O2) The agent schedules the addition (or removal) of the forwarding rule in hardware tables, typically TCAM. (O3) Depending on the nature of the rule, the chip SDK may require existing rules in the tables to be rearranged, e.g., to accommodate high priority rules. (O4) The rule is inserted (or removed) in the hardware table. All four steps, O1–O4, impact the total latency in executing a *flow_mod* action. We categorize this as *outbound latency*, since the controller outputs a *flow_mod* message.

3. LATENCY MEASUREMENTS

In this section, we systematically measure in/outbound latencies to understand what factors contribute to high latencies. We generate a variety of workloads to isolate specific factors, and we use production switches from three vendors, running switch software with support for OpenFlow 1.0 [14] or, if available, OpenFlow 1.3, to highlight the generality of our observations and to understand how software evolution impacts latencies.[1] Henceforth, we refer to the four hardware and software combinations (Table 1) as Intel, BCM-1.0, BCM-1.3, and IBM. To ensure we are experimenting in the optimal regimes for each switch, we take into account factors such as flow table capacity and support for *packet_in*.

3.1 Measurement Methodology

Figure 2 shows our measurement setup. The host has one 1Gbps and two 10Gbps interfaces connected to the switch under test. The eth0 interface is connected to the control port of the switch, and

[1]When using OpenFlow 1.3 firmware, we only leverage features also available in OpenFlow 1.0 for an apples-to-apples comparison.

Model	CPU	RAM	OF Ver.	Flow Table Size	Ifaces
Intel FM6000	2Ghz	2GB	1.0	4096	40x10G + 4x40G
Broadcom 956846K	1Ghz	1GB	1.0	896	14x10G + 4x40G
			1.3	1792 (ACL tbl)	
IBM G8264	?	?	1.0	750	48x10G + 4x40G

Table 1: Switch specifications

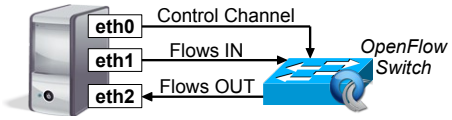


Figure 2: Measurement experiment setup

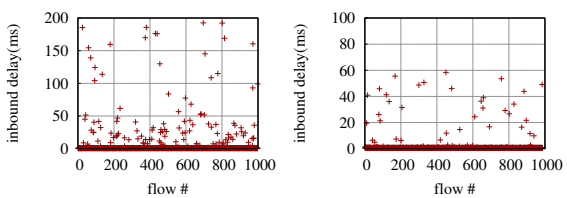
an SDN controller (POX for Intel, BCM-1.0, and IBM; RYU for BCM-1.3) running on the host listens on this interface. The RTT between switch and controller is negligible (≈0.1ms). We use the controller to send a burst of OpenFlow *flow_mod* commands to the switch. For Intel, BCM-1.0, and IBM, we install/modify/delete rules in the single table supported by OpenFlow 1.0; for BCM-1.3, we use the highest numbered table, which supports rules defined over any L2, L3, or L4 header fields. The host's eth1 and eth2 interfaces are connected to data ports on the switch. We run pktgen [15] in kernel space to generate traffic on eth1 at a rate of 600-1000Mbps using minimum Ethernet frame size.

Prior work notes that accurate execution times for OpenFlow commands on commercial switches can only be observed in the data plane [17]. Thus, we craft our experiments to ensure the latency impact of various factors can be measured directly from the data plane (at eth2 in Figure 2), with the exception of *packet_in* generation latency. We run *libpcap* on our measurement host to accurately timestamp the packet and rule processing events of each flow. We first log the timestamps in memory, and when the experimental run is complete, the results are dumped to disk and processed. We use the timestamp of the first packet associated with a particular flow as the finish time of the corresponding *flow_mod* command; more details are provided later in this section.

3.2 Dissecting Inbound Latency

To measure inbound latency, we empty the table at the switch, and we generate traffic such that *packet_in* events are generated at a certain rate (i.e., we create packets for new flows at a fixed rate). To isolate the impact of *packet_in* processing from other message processing, we perform two kinds of experiments: (1) the *packet_in* will trigger corresponding *flow_mod* (insert simple OpenFlow rules differing just in destination IP) and *packet_out* messages; (2) the *packet_in* message is dropped silently by the controller.

We record the timestamp (t_1) when each packet is transmitted on the measurement host's eth1 interface (Figure 2). We also record the timestamp (t_2) when the host receives the corresponding *packet_in*



(a) with flow_mod/pkt_out (b) w/o flow_mod/pkt_out

Figure 3: **Inbound delay** on **Intel**, flow arrival rate = 200/s

with flow mod/pkt out		
flow rate	100/s	200/s
cpu usage	15.7%	26.5%

w/o flow mod/pkt out		
flow rate	100/s	200/s
cpu usage	9.8%	14.4%

Table 2: CPU usage on Intel

message on eth0. The difference $(t_2 - t_1)$ is the inbound latency.[2]

Representative results for an Intel switch are shown in Figure 3; IBM has similar performance (5ms latency per *packet_in* on average).[3] In the first experiment (Figure 3a), we see the inbound latency is quite variable with a mean of 8.33ms, a median of 0.73ms, and a standard deviation of 31.34ms. In the second experiment (Figure 3b), the inbound delay is lower (mean of 1.72ms, median of 0.67ms) and less variable (standard deviation of 6.09ms). We also observe that inbound latency depends on the *packet_in* rate: e.g. in the first experiment the mean is 3.32 ms for 100 flows/s (not shown) vs. 8.33ms for 200 flows/s (Figure 3a).

The only difference between the two experiments is that in the former case the switch CPU must process *flow_mod* and *packet_out* messages, and send forwarding entries and outbound packets across the PCIe bus to the ASIC, in addition to generating *packet_in* messages. As such, we observe that the CPU usage is higher when the switch is handling concurrent OpenFlow operations and generating more *packet_in* messages (Table 2). However, since the Intel switch features a powerful CPU (Table 1), plenty of CPU capacity remains. Our conversations with the switch vendor suggest that the limited bus bandwidth between the ASIC and switch CPU is the primary factor contributing to inbound latency.

3.3 Dissecting Outbound Delay

We now study the outbound latencies for three different *flow_mod* operations: insertion, modification, and deletion. For each operation, we examine the latency impact of key factors, including table occupancy and rule priority.

Before measuring outbound latency, we install a single default low priority rule which instructs the switch to drop all traffic. We then install a set of non-overlapping OpenFlow rules that output traffic on the port connected to the eth2 interface of our measurement host. For some experiments, we systematically vary the rule priorities.

3.3.1 Insertion Latency

We first examine how different rule workloads impact insertion latency. We insert a burst of B rules: $r_1, \cdots, r_B$. Let $T(r_i)$ be the time we observe the first packet matching r_i emerging from the output port specified in the rule. We define per-rule insertion latency as $T(r_i) - T(r_{i-1})$.

Rule Complexity. To understand the impact of rule complexity (i.e., the number of header fields specified in a rule), we install bursts of rules that specify either 2, 8, or 12 fields. In particular, we specify destination IP and EtherType (others wilcarded) in the 2-field case; input port, EtherType, source and destination IPs, ToS, protocol, and source and destination ports in the 8-field case; and all supported header fields in the 12-field (exact match) case. We use a burst size of 100 and all rules have the same priority.

We find that rule complexity *does not* impact insertion latency. The mean per-rule insertion delay for 2-field, 8-field, and exact match cases is 3.31ms, 3.44ms, and 3.26ms, respectively, for BCM-1.0. Similarly, the mean per-rule insertion delay for Intel, IBM, and BCM-1.3 is ≈ 1 ms irrespective of the number of fields. All experiments that follow use rules with 2 fields.

[2]Our technique differs from [8], where the delay was captured from the switch to the controller, which includes controller overhead.

[3]BCM-1.0 and BCM-1.3 do not support *packet_in* messages.

Table occupancy. To understand the impact of table occupancy, we insert a burst of B rules into a switch that already has S rules installed. All $B + S$ rules have the same priority. We fix B and vary S, ensuring $B+S$ rules can be accommodated in each switch's hardware table.

We find that flow table occupancy *does not* impact insertion delay if all rules have the same priority. Taking $B = 400$ as an example, the mean per-rule insertion delay is 3.14ms, 1.09ms, 1.12ms, and 1.11ms (standard deviation 2.14ms, 1.24ms, 1.53ms, and 0.18ms) for BCM-1.0, BCM-1.3, IBM and Intel, respectively, regardless of the value of S.

Rule priority. To understand the effect of rule priority on the insertion operations, we conducted three different experiments each covering different patterns of priorities. In each, we insert a burst of B rules into an empty table ($S = 0$); we vary B. In the *same priority* experiment, all rules have the same priority. In the *increasing* and *decreasing priority* experiments, each rule has a different priority and the rules are inserted in increasing/decreasing priority order, respectively.

Representative results for same priority rules are shown in Figure 4a and 4b for $B = 100$ and $B = 200$, respectively; the switch is BCM-1.0. For both burst sizes, the per-rule insertion delay is similar, with medians of 3.12ms and 3.02ms, and standard deviations of 1.70ms and 2.60ms for $B = 100$ and $B = 200$, respectively. The same priority insertion delays on BCM-1.3, IBM, and Intel are slightly lower, but still similar: mean per-rule insertion delay is 1.09ms, 1.1ms, and 1.17ms, respectively, for $B = 100$. We conclude that *same priority* rule insertion delay does not vary with burst size.

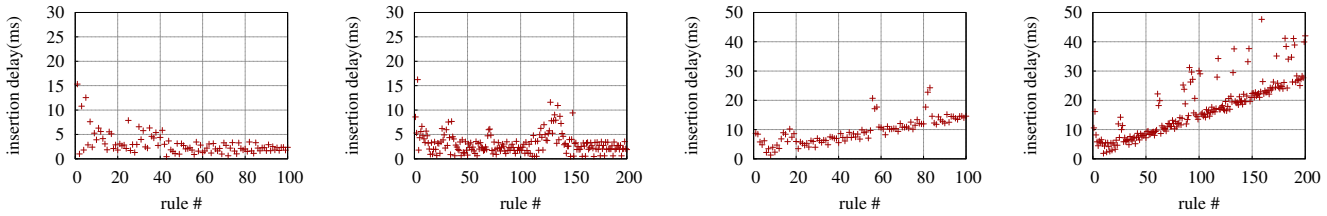
In contrast, the per-rule insertion delay of increasing priority rules *increases linearly* with the number of rules inserted for BCM-1.0, BCM-1.3, and IBM. Figure 4c and 4d shows this effect for $B = 100$ and $B = 200$, respectively, for BCM-1.0. Compared with the same priority experiment, the average per-rule delay is much larger: 9.47ms (17.66ms) vs. 3.12ms (3.02ms), for $B = 100$ (200). The results are similar for BCM-1.3 and IBM: the average per-rule insertion delay is 7.75ms (16.81ms) and 10.14ms (18.63) for $B = 100$ (200), respectively. We also observe the slope of the latency increase is constant—for a given switch—regardless of B.

The increasing latency in BCM-1.0, BCM-1.3, and IBM stems from the TCAM storing high priority rules at low (preferred) memory addresses. Each rule inserted in the *increasing priority* experiments displaces all prior rules!

Surprisingly, latency does not increase when increasing priority rules are inserted in Intel. As shown in Figure 5a, the median per-rule insertion delay for Intel is 1.18ms (standard deviation of 1.08ms), even with $B = 800$! Results for other values of B are similar. This shows that the Intel TCAM architecture is fundamentally different from Broadcom and IBM. Rules are ordered in Intel's TCAM such that higher priority rules do not displace existing low priority rules.

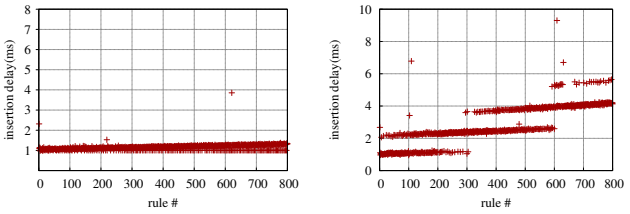
However, displacement does still occur in Intel. Figure 5b shows per-rule insertion latencies for for *decreasing priority* rules for $B = 800$. We see two effects: (1) the latencies alternate between two modes at any given time, and (2) there is a step-function effect after every 300 or so rules.

A likely explanation for the former is bus buffering. Since rule insertion is part of the switch's control path, it is not really optimized for latency. The latter effect can be explained as follows: Examining the Intel switch architecture, we find that it has 24 slices, $A_1 \ldots A_{24}$, and each slice holds 300 flow entries. There exists a consumption order (low-priority first) across all slices. Slice A_i stores the i^{th} lowest priority rule group. If rules are inserted in de-



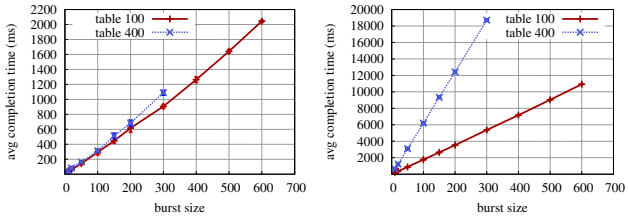
(a) burst size 100, same priority (b) burst size 200, same priority (c) burst size 100, incr. priority (d) burst size 200, incr. priority

Figure 4: **BCM-1.0** priority per-rule **insert** latency



(a) burst size 800, incr. priority (b) burst size 800, decr. priority

Figure 5: **Intel** priority per-rule **insert**



(a) insert low priority rules into a table with high priority rules (b) insert high priority rules into a table with low priority rules

Figure 6: Overall completion time on **BCM-1.0**. Initial table occupancy is S high (low) priority rules; insert a burst of low (high) priority rules. Averaged over 5 runs.

creasing priority, A_1 is consumed first until it becomes full. When the next low priority rule is inserted, this causes one rule to be displaced from A_1 to A_2. This happens for each of the next 300 rules, after which cascaded displacements happen: $A_1 \rightarrow A_2 \rightarrow A_3$, and so on. We confirmed this with Intel.

We observe different trends when inserting *decreasing priority* rules in BCM-1.0, BCM-1.3, and Intel. With BCM-1.0, we find the average per-rule insertion delay increases with burst size: 8.19ms for $B = 100$ vs. 15.5ms for $B = 200$. Furthermore, we observe that the burst of B rules is divided into several groups, and each group is reordered and inserted in the TCAM in order of increasing priority. This indicates that BCM-1.0 firmware reorders the rules and prefers increasing priority insertion. In contrast, BCM-1.3's per-rule insertion delay for decreasing priority rules is similar to same priority rule insertion: $\approx$ 1ms. Hence, the BCM-1.3 firmware has been better optimized to handle decreasing priority rule insertions. The same applies to Intel: per-rule insertion delay for decreasing priority rules is similar to same priority rule insertion: $\approx$ 1.1ms.

Priority and table occupancy combined effects. We now study the combined impact of rule priority and table occupancy. We conduct two experiments: For the first experiment, the table starts with S high priority rules, and we insert B low priority rules. For the second experiment, the priorities are inverted. For both experiments, we measure the total time to install all rules in the burst, $T(r_B) - T(r_1)$.

For BCM-1.0, BCM-1.3, and IBM, we expect that as long as the same number of rules are displaced, the completion time for different values of S should be the same. Indeed, from Figure 6a (for BCM-1.0), we see that even with 400 high priority rules in the table, the insertion delay for the first experiment is no different from the setting with only 100 high priority rules in the table. In contrast, in Figure 6b, newly inserted high priority rules will displace low priority rules in the table, so when $S = 400$ the completion time is about 3x higher than $S = 100$. For IBM (not shown), inserting 300 high priority rules into a table with 400 low priority rules takes more than 20 seconds.

For Intel, the results are similar to same priority rule insertion. This indicates that Intel uses different TCAM organization schemes than the Broadcom and IBM switches.

Summary and root causes. We observe that: (1) rule complexity does not affect insertion delay; (2) same priority insertions in BCM-1.0, BCM-1.3, Intel and IBM are fast and not affected by flow table occupancy; and (3) priority insertion patterns can affect insertion delay very differently. For Intel, increasing priority insertion is similar to same priority insertion, but decreasing priority incurs much higher delay. For BCM-1.3 and IBM the behavior is inverted: decreasing priority insertion is similar to same priority insertion and increasing priority insertion incurs higher delay. For BCM-1.0, insertions with different priority patterns are all much higher than insertions with same priority.

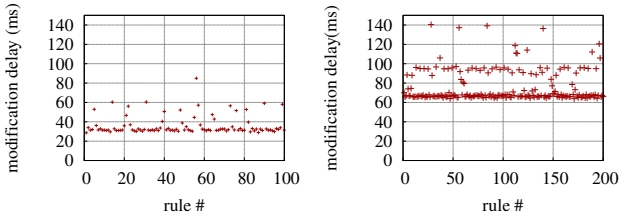
Key root causes for observed latencies are: (1) how rules are organized in the TCAM, and (2) the number of slices. *Both of these are intrinsically tied to switch hardware.* Even in the best case (Intel), per-rule insertion latency of 1ms is higher than what native TCAM hardware can support (100M updates/s [1]). Thus, in addition to the above two causes, there appears to be an *intrinsic switch software overhead* contributing to all latencies.

3.3.2 *Modification Latency*

We now study modification operations. As before, we use bursts of rules and a similar definition of latency.

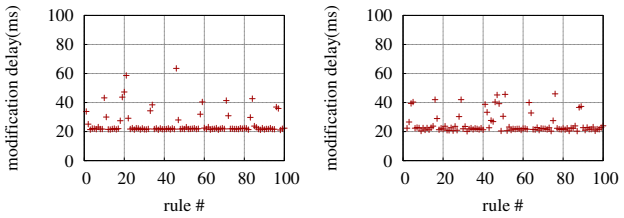
Table occupancy. To study the impact of table occupancy, we preinsert S rules into a switch, all with the same priority. We then modify one rule at a time by changing the rule's output port, sending modification requests back to back.

Per-rule modification delay for BCM-1.0 when $S = 100$ and $S = 200$ are shown in Figure 7a and 7b, respectively. We see that the per-rule delay is more than 30 ms for $S = 100$. When we double the number of rules, $S = 200$, latency doubles as well. It grows linearly with S (not shown). Note that this latency is much higher than the corresponding insertion latency (3.12ms per rule) (§3.3.1). IBM's per-rule modification latency is also affected significantly by the table occupancy— the per-rule modification latencies for $S = 100$ and $S = 200$ are 18.77ms and 37.13ms, respectively.



(a) 100 rules in table (b) 200 rules in table

Figure 7: **BCM-1.0** per-rule **mod.** latency, same priority



(a) burst size 100, incr. priority (b) burst size 100, decr. priority

Figure 8: **BCM-1.0** priority per-rule **modification** latency

In contrast, Intel and BCM-1.3 have lower modification delay, and it does not vary with table occupancy. For Intel (BCM-1.3) the per-rule modification delay for both $S = 100$ and $S = 200$ is around 1 ms (2ms) for all modified rules, similar to (2X more than) same priority insertion delay.

Rule Priority. We conduct two experiments on each switch to study the impact of rule priority. In each experiment, we insert B rules into an empty table ($S = 0$). In the *increasing* priority experiments, the rules in the table each have a unique priority, and we send back-to-back modification requests for rules in increasing priority order. We do the opposite in the *decreasing priority* experiment. We vary B.

Figure 8a and 8b show the results for the increasing and decreasing priority experiments, respectively, for $B = 100$ on BCM-1.0. In both cases, we see: (1) the per-rule modification delay is similar across the rules, with a median of 25.10ms and a standard deviation of 6.74ms, and (2) the latencies are identical across the experiments. We similarly observe that priority does not affect modification delay in BCM-1.3, Intel and IBM (not shown).

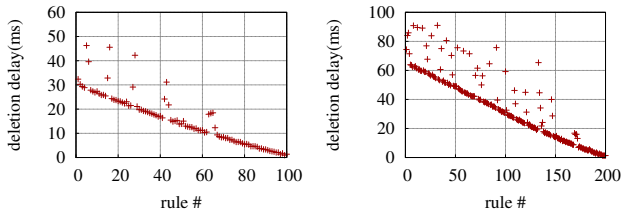
Summary and root causes. We conclude that the per-rule modification latency on BCM-1.0 and IBM is impacted purely by table occupancy, not by rule priority structure. For BCM-1.3 and Intel, the per-rule modification delay is independent of rule priority, table occupancy, and burst size; BCM-1.3's per-rule modification delay is 2X higher than insertion.

Conversations with Broadcom indicated that TCAM modification should ideally be fast and independent of table size, so the underlying cause appears to be less optimized switch software in BCM-1.0. Indeed, our measurements with BCM-1.3 show that this issue has (at least partly) been fixed.

3.3.3 *Deletion Latency*

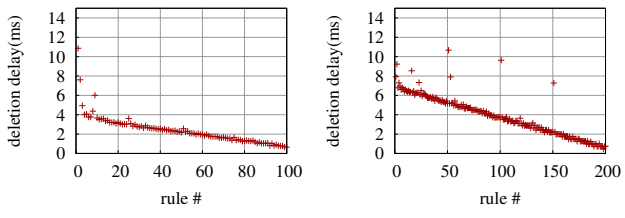
We now study the latency of rule deletions. We again use bursts of operations. $T(r_i)$ denotes the time we stop observing packets matching rule r_i from the intended port of the rule action. We define deletion latency as $T(r_i) - T(r_{i-1})$.

Table Occupancy. We pre-insert S rules into a switch, all with the same priority. We then delete one rule at a time, sending deletion requests back-to-back. The results for BCM-1.0 at $S = 100$ and $S = 200$ are shown in Figure 9a and 9b, respectively. We see that per rule deletion delay decreases as the table occupancy drops. We see a similar trend for Intel (Figure 10a and 10b) BCM-1.3 and IBM (not shown).



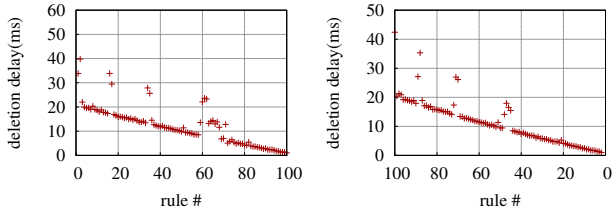
(a) 100 rules in table (b) 200 rules in table

Figure 9: **BCM-1.0** per-rule **del.** latency, same priority



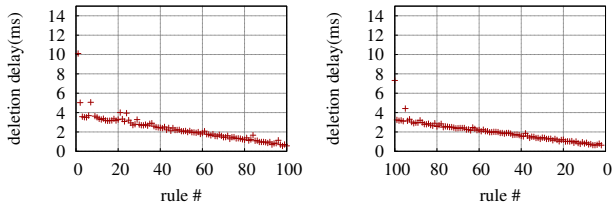
(a) 100 rules in table (b) 200 rules in table

Figure 10: **Intel** per-rule **del.** latency, same priority



(a) increasing priority (b) decreasing priority

Figure 11: **BCM-1.0** priority per-rule **del.** latency, B=100



(a) increasing priority (b) decreasing priority

Figure 12: **Intel** priority per-rule **del.** latency, B=100

Rule Priorities. We start with B existing rules in the switch, and delete one rule at a time in increasing and decreasing priority order. For all switches (BCM-1.0 and Intel shown in Figure 11 and 12, respectively) deletion is not affected by the priorities of rules in the table or the order of deletion.

Root cause. Since deletion delay decreases with rule number in all cases, we conclude that deletion is incurring TCAM reordering. We also observe that processing rule timeouts at the switch does not noticeably impact *flow_mod* operations. Given these two observations, we recommend allowing rules to time out rather than explicitly deleting them, if possible.

3.4 Implications

A subset of our findings highlight problems with the firmware on OpenFlow switches: e.g., rule insertion latencies are 3ms with BCM-1.0, which is significantly higher than the update rate that TCAM hardware natively supports [1]. We believe near term work will reduce such issues, as indicated by improved latencies in BCM-1.3. However, given that software will continue to bridge control and data planes in SDN switches, we remain skeptical whether latencies will ever reach what hardware can natively support.

Our measurements also reveal root causes of latency that appear to be fundamentally entrenched in hardware design: e.g., rules must be organized in the TCAM in a priority order for correct and efficient matching; also, *packet_in*, *flow_mod*, and *packet_out* messages must contend for limited bus bandwidth between a switch's CPU and ASIC. Unless the hardware significantly changes, we believe the latencies we identify will continue to manifest in next generation switches.

4. RELATED WORK

A few prior studies have considered SDN switch performance. However, they either focus on narrow issues, do not offer sufficient in-depth explanations for observed performance issues, or do not explore implications on applications that require tight control of latency. Devoflow [4] showed that the rate of statistics gathering is limited by the size of the flow table and that statistics gathering negatively impacts flow setup rate. More recently, two studies [8, 17] provided a more in-depth look into switch performance across various vendors. In [17], the authors evaluate 3 commercial switches and observed that switching performance is vendor specific and depends on applied operations and firmware. In [8], the authors also studied 3 commercial switches (HP Procurve, Fulcrum, Quanta) and found that delay distributions were distinct, mainly due to variable control delays. Our work is complementary with, and more general, than these results. Some studies have considered approaches to mitigate the overhead of SDN rule matching and processing. DevoFlow [4] presents a rule cloning solution which reduces the number of controller requests being made by the switch by having the controller set up rules on aggregate or elephant flows. DIFANE [18] reduces flow set up latency by splitting pre-installed wild card rules among multiple switches and therefore all decisions are still made in the data plane. Lastly, Dionysus [11] optimally schedules a set of rule updates while maintaining desirable consistency properties (e.g., no loops and no blackholes).

5. CONCLUSION

Critical SDN applications such as fast failover and fine-grained traffic engineering demand fast interaction between switch control and data planes. However, our measurements of four SDN switches show the latencies underlying the generation of control messages and the execution of control operations can be quite high, and variable. We find that the underlying causes are linked to software inefficiencies, as well as pathological interactions between switch hardware properties (shared resources and how forwarding rules are organized) and the control operation workload (the order of operations, and concurrent switch activities). These findings highlight the need for careful design of future switch silicon and software in order to fully utilize the power of SDN.

Acknowledgement

We would like to thank Cristian Estan, Frank Feather, Young-Jin Kim, Simon Knee, Syd Logan, Rob Sherwood for helping collect data and understand switch internals. We would also like to thank the anonymous reviewers for their insightful feedback. This work is supported in part by NSF grants CNS-1302041, CNS-1314363 and CNS-1040757.

6. REFERENCES

[1] Private communication with Cristian Estan.

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

[3] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker. Rethinking Enterprise Network Control. *IEEE/ACM Trans. Netw.*, 2009.

[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 *SIGCOMM*, 2011.

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

[6] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: Saving Energy in Data Center Networks. In *NSDI*, 2010.

[7] 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.

[8] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity Switch Models for Software-defined Network Emulation. In *HotSDN*, 2013.

[9] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with A Globally-deployed Software Defined WAN. In *SIGCOMM*, 2013.

[10] X. Jin, L. E. Li, L. Vanbever, and J. Rexford. SoftCell: Scalable and Flexible Cellular Core Network Architecture. In *CoNEXT*, 2013.

[11] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic Scheduling of Network Updates. In *SIGCOMM*, 2014.

[12] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, et al. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.

[13] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. zUpdate: Updating Data Center Networks with Zero Loss. In *SIGCOMM*, 2013.

[14] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 2008.

[15] R. Olsson. Pktgen the Linux Packet Generator. In *Ottawa Linux Symposium*, 2005.

[16] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, et al. Ananta: Cloud Scale Load Balancing. In *SIGCOMM*, 2013.

[17] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. Oflops: An Open Framework for Openflow Switch Evaluation. In *PAM*, 2012.

[18] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable Flow-based Networking with DIFANE. In *SIGCOMM*, 2010.