Latency in Software Defined Networks: Measurements and Mitigation Techniques

Paper #91, 12 pages + 1 page appendix

ABSTRACT

Timely interaction between an SDN controller and switches is crucial to many SDN applications, such as 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. Simulations based on our measurements indicate that these high latencies render SDN incapable of supporting key control applications. Since the hardware changes necessary to lower latencies are likely 5+ years away, we propose three immediately deployable measurement-driven latency mitigation techniques: optimizing route selection, spreading rules across switches, and reordering rule installations. Simulations show that our techniques can reduce the latency to update network state by almost 5X.

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 [**?**, **?**]. 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 [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**].

In order 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 [**?**]. 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 [**?**].

For such applications, timely interaction between the logically central SDN control plane and network switches is crucial. 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 them. Robust control software design and advances in distributed controllers [**?**] 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 both grey-box 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). We find 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 high as well (3ms and 30ms per rule for insertion and modification, respectively, in Broadcom). The latency crucially depends on the priority patterns both in rules being inserted as well as those already in a switch's table. We find that there are significant differences in latency trends across switches with different chipsets/firmware, pointing to different internal optimizations.

Wouldn't improvements in switch designs eliminate these latencies over time? Some of our findings show that poor switch software design contributes significantly to observed latencies (affirming [**?**, **?**]). We believe that near term work will address these issues; our measurements with an early release of Broadcom's OpenFlow 1.3 software exemplify this.

More crucially, our measurements also 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.

Since hardware upgrade cycles are 5+ years long (according to anecdotal evidence), operators need immediately deployable solutions to mitigate these latencies while they wait for switch software and hardware to appropriately evolve. Many SDN applications already avoid (or minimize their dependence on) packet-in messages, thereby mitigating the impact of inbound latency. In contrast, rule installations/updates are an intrinsic part of SDN applications, which makes outbound latency more challenging, and important, to address.

In this paper, we propose three techniques to mitigate the outbound latencies imposed by current switches: *Flow engineering* (FE) leverages our empirical latency models to compute paths such that the latency of installing forwarding state at any switch is minimized. *Rule offloading* (RO) computes strategies for opportunistically offloading installation of some forwarding state to downstream switches. Finally, *rule reordering* (RR) sends rule installation requests in an order that is optimal for the switch in question. By reducing installation latency per switch (FE + RR) and enabling network-wide parallel updates (RO), rule updates can finish much faster.

We evaluate these techniques for fast fail-over and responsive traffic engineering applications under various settings. Depending on the topology and the nature of rules in switches, we find that outbound latencies can render SDN incapable of supporting such applications. In contrast, our techniques can improve the time taken to update network state in these scenarios by factors of 1.6-5X, which we argue makes SDN-based control suitably responsive for these settings.

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 controller. Such packet events and state update operations are enabled by OpenFlow [**?**]—a standard API implemented by switches to facilitate communication with the controller.

Despite moving control plane logic from switches to applications, switches must still perform several steps to generate packet events and update forwarding state.

Inbound Latency. 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 **??**): (I1) The switch ASIC decides to send the packet to the switch's CPU via the PCI/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.

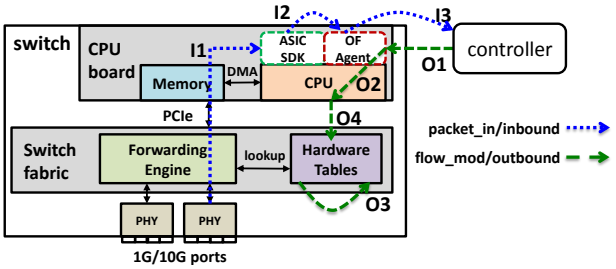


Figure 1: Schematic of an OpenFlow switch. We also show the factors contributing to inbound and outbound latency

Outbound Latency. The application residing on the controller processes the *packet_in*, and upon determining a route for packets belonging to the corresponding flow, sends a *flow_mod* message and a *packet_out* message. *flow_mod* can also be sent by a controller without receiving a *packet_in*.

A *flow_mod* message describes the action the switch should apply to all future packets of the flow: this could be forward according to some rule, update an existing rule, or delete a rule. The format of a "simple" wildcard rule is *in-port=1,dstip=10.0.0.2,action=output:3*.[1] The rule may also specify a priority (to determine which rule to apply when multiple rules match a packet) and a timeout after which the rule is deleted from the table.

A switch takes the following steps to handle a *flow_mod*: (O1) The OpenFlow agent running on the CPU parses the message. (O2) The agent schedules the rule to be applied to hardware tables, typically TCAM. (O3) Depending on the nature of the rule, the chip SDK may require existing rules in the tables to be moved around, e.g., to accommodate high priority rules. (O4) The hardware table is updated with the rule. 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.

A *packet_out* message releases a specific packet buffered at the switch, and the switch forwards it as specified in the message. The steps taken by the switch are the inverse of those for generating *packet_in* messages; the latency for these steps is another form of *outbound latency*.

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,

[1] In OpenFlow 1.0, a rule can match up to 12 common header fields.

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

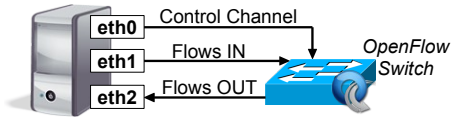


Figure 2: Measurement experiment setup

running switch software with support for OpenFlow 1.0 [**?**] or, if available, OpenFlow 1.3, to highlight the generality of our observations and to understand how software evolution impacts latencies.[2] Henceforth, we refer to the four hardware and software combinations (Table **??**) 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*.

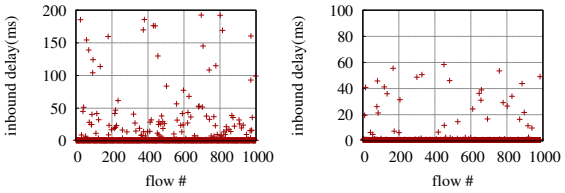
3.1 Measurement Methodology

Figure **??** 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 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 [**?**] in kernel space to generate traffic on eth1 at a rate of 600-1000Mbps.

Prior work notes that accurate execution times for OpenFlow commands on commercial switches can only be observed in the data plane [**?**]. 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 **??**), 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

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



(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

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 **??**). We also record the timestamp (t_2) when the host receives the corresponding *packet_in* message on eth0. The difference ($t_2 - t_1$) is the inbound latency.[3]

Representative results for these two experiments are shown in Figure **??** and **??**, respectively, for the Intel switch; IBM has similar performance (5ms latency per *packet_in* on average). BCM-1.0 and BCM-1.3 do not support *packet_in* messages. For the first experiment, we see that the inbound latency is quite variable with a mean of 8.33ms, a median of 0.73ms, and a standard deviation of 31.34ms. For the second experiment, 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 first experiment the mean is 3.32 ms for 100 flows/s (not shown) vs. 8.33ms for 200 flows/s (Figure **??**).

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 **??**). However, since the Intel switch features a powerful CPU (Table **??**), plenty of CPU capacity remains. Our conversations with the switch vendor suggest

[3]This measurement technique differs from the approach used in [**?**], where the delay was captured from the switch to the POX controller which includes the overhead at the controller.

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.

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.

BCM-1.0, same priority. Representative results for $B = 100$ and $B = 200$ are shown in Figure **??** and **??**, respectively. In both cases, we see that 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. We conclude that same priority rule insertion delay does not vary with burst size on BCM-1.0.

BCM-1.0, increasing priority. Figure **??** shows the result for $B = 100$. We note that the per-rule insertion delay actually *increases linearly* with the number of rules inserted. Figure **??** shows the result for $B = 200$; we see that the slope stays the same as $B = 100$. 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). Results for other values of B are qualitatively similar. The TCAM in this switch stores high priority rules at low (preferred) memory addresses. Thus, each rule inserted in this experiment displaces all prior rules!

BCM-1.0, decreasing priority. We also perform decreasing priority insertion (not shown). The average per-rule insertion delays for $B = 100$ and $B = 200$ are 8.19ms and 15.5ms, respectively. We observe that the burst of B rules is divided into a number of 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.

BCM-1.3, same priority. The mean per-rule insertion delay is 1.09ms (1.08ms) for $B = 100$ (200). Thus, similar to BCM-1.0, the rule insertion time does not vary with burst size when all rules are of the same priority.

BCM-1.3, increasing priority. The average per-rule insertion delay is much larger: 7.75ms (16.81ms) for $B = 100$ (200). This is similar to our findings for BCM-1.0, affirming that TCAM organization requirements, not software implementation issues, are the primary cause.

BCM-1.3, decreasing priority. The per-rule delay is similar to that of same priority insertion: $\approx$ 1ms. This contrasts with BCM-1.0, where decreasing priority insertion increases with the number of rules inserted—average of 8.19ms (15.5ms) for $B = 100$ (200). Hence the BCM-1.3 firmware has been better optimized to handle decreasing priority rule insertions.

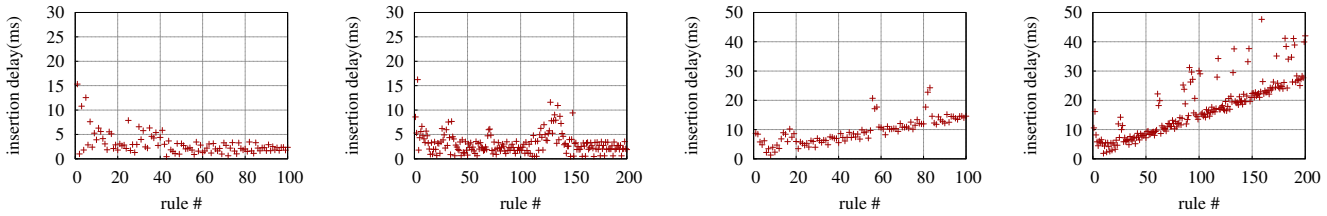
IBM, same priority. The IBM switch's same priority rule insertion performance trend is quite similar to that of Broadcom. When all the inserted rules have the same priority, the per-rule insertion latency is around 1.1ms.

IBM, increasing priority. When it comes to increasing priority, the per-rule insertion latency becomes significantly larger. The average per-rule insertion delays for $B = 100$ and $B = 200$ are 10.14ms and 18.63ms, respectively.

IBM, decreasing priority. IBM switch's per-rule insertion latency in decreasing priority is similar to that of same priority rule insertion, namely, around 1.1ms per insertion.

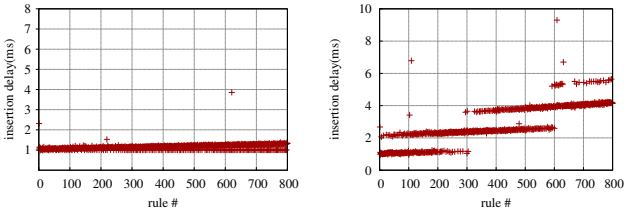
Intel, same priority. For $B = 800$ on Intel[4] we see that the

[4]We present results for a larger value of B because the flow table



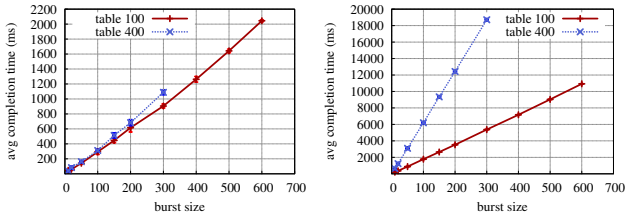
(a) burst size 100, same priority (b) burst size 200, same priority (c) burst size 100, incr. priority (d) burst size 200, incr. priority

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.

per-rule insertion delay is similar across the 800 rules, with a median of 1.17ms and standard deviation of 0.16ms (not shown). The results for other values of B are similar. Thus, similar to BCM-1.0, BCM-1.3, and IBM same priority rule insertion delay does not vary with burst size on Intel.

Intel, increasing priority. Figure **??** shows per-rule latencies for $B = 800$. *Surprisingly*, in contrast with BCM-1.0, BCM-1.3, and IBM, the per rule insertion delay among the rules is more or less the same, with a median of 1.18ms and a standard deviation of 1.08ms. We see similar results for other values of B. This shows that the Intel TCAM architecture is fundamentally different from Broadcom. Rules are ordered in Intel's TCAM such that higher priority rule insertion does not displace existing low priority rules.

Intel, decreasing priority. Figure **??** shows per-rule insertion latencies for $B = 800$. We see two effects: (1) the latencies alternate between two modes at any given time, and (2) 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 decreasing 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.

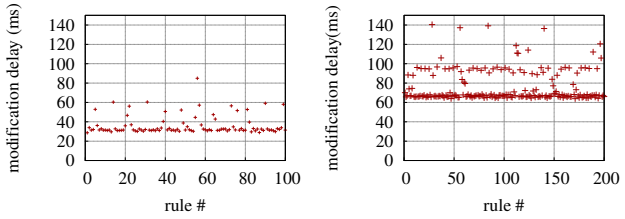
Priority and table occupancy combined effects. We now study the combined impact of rule priority and table occupancy. We conduct two experiments: For the first experiment, the table starts with S high priority rules, and we insert B low priority rules. For the second experiment, the priorities are inverted. 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 **??** (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 **??**, 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.

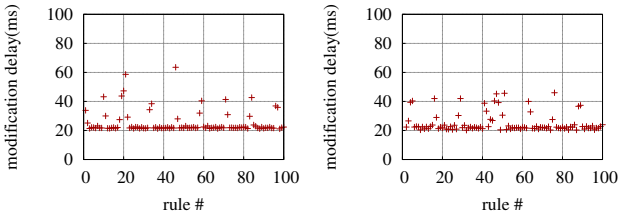
Summary and root causes. We observe that: (1) rule complexity does not affect insertion delay; (2) same priority insertions in BCM-1.0, BCM-1.3, Intel and IBM are fast and not affected by flow table occupancy; and (3) priority insertion patterns can affect insertion delay very differently. For Intel, 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-

size on Intel is larger (Table **??**).



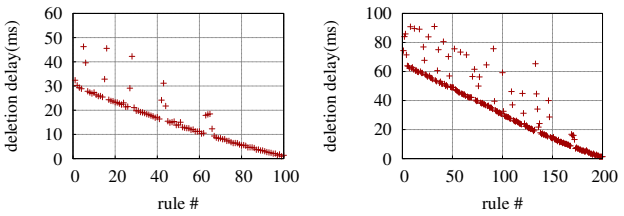
(a) 100 rules in table (b) 200 rules in table

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



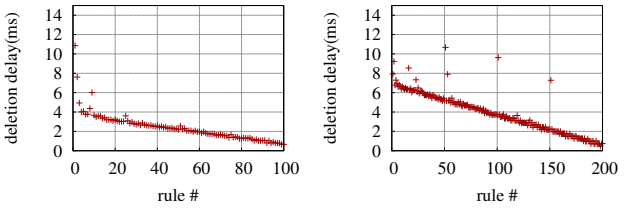
(a) burst size 100, incr. priority (b) burst size 100, decr. priority

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



(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

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 [**?**]). 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 experiment with bursts of rules. Modification latency is defined similar to insertion.

Table occupancy. To study the impact of table occupancy, we pre-insert 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 **??** and **??**, 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) (§**??**). 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.

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 **??** and **??** 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).

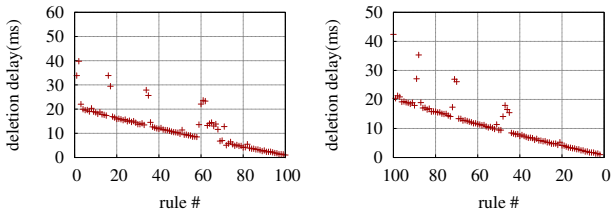
Summary and root causes. We conclude that the per-rule modification latency on BCM-1.0 and IBM is impacted purely by table occupancy, not by rule priority structure. For BCM-1.3 and Intel, the per-rule modification delay is independent of rule priority, table occupancy, and burst size; BCM-1.3's per-rule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

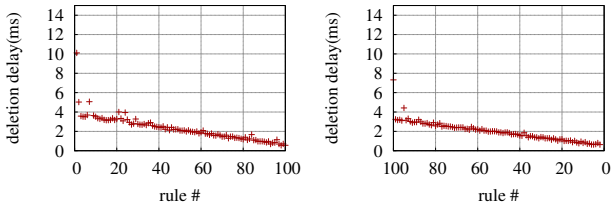
We now estimate the impact of rule deletions. We use bursts of operations as before. Denote $T(r_i)$ as the first time we stopped observing packets matching rule r_i from the intended port of the rule action. We define deletion latency as $T(r_i) - T(r_{i-1})$.

Table Occupancy. We pre-insert S rules into a switch, all with the same priority. We then delete one rule at a time,



(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

sending deletion requests back-to-back. The results for BCM-1.0 at $S = 100$ and $S = 200$ are shown in Figure **??** and **??**, respectively. We see that per rule deletion delay decreases as the table occupancy drops. We see a similar trend for Intel (Figure **??** and **??**) BCM-1.3 and IBM (figure not shown).

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 BCM-1.0 (Figure **??**), BCM-1.3 (figure not shown) Intel (Figure **??**) and IBM, 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

Prior work [**?**, **?**] has highlighted problems with the software implementation on OpenFlow switches. A subset of our findings confirm this: e.g., rule insertion latencies are 3ms with BCM-1.0, which is significantly higher than the update rate that TCAM hardware natively supports [**?**]. We believe near term work will reduce such issues, as indicated by improved latencies in BCM-1.3. However, given that software "glue" will continue to exist between 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.

A central contribution of this paper is to highlight these latencies and point out to application designers, chip hardware and SDK developers, and switch vendors the steps they need to take to curb these latencies.

However, given that organizations already have vast beds of switches and hardware upgrade cycles are 5+ years, operators need solutions to cope with these latencies until hardware and software sufficiently evolve to match their requirements. Many SDN applications already avoid (or minimize their dependence on) packet-in messages, thereby mitigating the impact of inbound latency. In contrast, rule installations/updates are an intrinsic part of SDN applications, which makes outbound latency more challenging, and important, to address. In the next section, we present three immediately deployable techniques to mitigate outbound latency.

4. MITIGATING THE LATENCY OF PROGRAMMING NETWORK STATE

The success of many SDN applications hinges on their ability to quickly program network state. For example:

WAN Failover. When WAN failures occur, an SDN application may want to quickly compute new paths for flows traversing failed nodes or links, while simultaneously rerouting other high/low priority flows to avoid hot-spots [**?**]. However, this requires significant updates to network state at multiple switches. The longer these updates take, the longer traffic is subject to congestion and loss. We find that outbound latencies can inflate failure response time by $\approx$ 20s (§**??**).

Intra-Datacenter Traffic Engineering. To eliminate hotspots and maximize data center performance, an SDN application may want to reroute traffic subsets at fine timescales [**?**]. For example, MicroTE [**?**] leverages the fact that a significant fraction of ToR-to-ToR traffic (ToR is "top-of-rack" switch) is predictable on short time-scales of 1-2s. Thus, MicroTE frequently computes and updates routes at ToR switches. Unfortunately, outbound latencies can cause these updates to take up to 0.5s (§**??**), so traffic that is predictable on 1s timescales is optimally routed only half the time.

To mitigate the impact of outbound latency, and support the needs of these and other SDN apps, we propose three immediately deployable techniques: flow engineering (FE), rule offload (RO), and rule reordering (RR). This section discusses these techniques in detail; we evaluate them in §**??**.

4.1 Flow Engineering

SDN applications that perform failure recovery, traffic engineering, or other forms of routing must quickly compute and setup network paths in order to satisfy reachability and performance objectives. These applications must often select one of many possible paths based on congestion, delay, flow table occupancy, or other metrics. Unfortunately, a (slightly) more optimal path according to these metrics may take significantly longer to setup due to outbound latency.

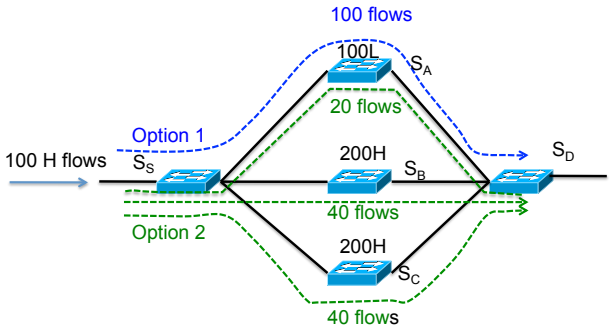


Figure 13: Flow engineering example, assuming BCM-1.0. $nL = n$ low priority rules; $nH = n$ high priority rules; capacity = 1000 rules. Ingress and egress tables are empty

For example, consider an SDN application that seeks to minimize the imbalance in flow table occupancy [**?**, **?**, **?**]. If such an application needed to setup routes for 100 flows of high priority H in the topology shown in Figure **??**, it would select the first path through switch S_A for all flows, thereby equalizing the number of flow table entries across all switches. However, assuming the switches are BCM-1.0, this would require displacing *each* of the 100 existing rules of low priority L in S_A for *each* of the 100 new flows, resulting in a total flow installation time of ≈1.5s. In contrast, by routing only 20 new flows through S_A, and dividing the remaining flows evenly between the paths through S_B and S_C, we can keep the same level of imbalance in flow table occupancy (S_B and S_C each have twice as many entries as S_A), while reducing the total flow installation time to ≈0.3s (assuming rules are installed in S_A, S_B, and S_C in parallel).

The goal of *flow engineering* (FE) is to select paths that minimize installation delay while still satisfying an SDN application's primary path selection criteria (e.g., flow table occupancy or congestion). Assuming there are many possible sets of paths $\{\mathcal{P}^i_{obj}\}_i$ that (closely) satisfy an application's objectives, FE selects the set $\mathcal{P}^{displace}_{obj,tbl\_sz}$ that minimizes the aggregate latency impact of rule installations, and any associated rule displacements, while still obeying flow table space constraints. $\mathcal{P}^{displace}_{obj,tbl\_sz}$ can be computed using a two step optimization: (*i*) identify sets of paths that satisfy the SDN application's objective function, but do not select the actual paths to use; (*ii*) select paths that minimize aggregate flow installation time. Unfortunately, the time required to solve the integer linear program (presented in Appendix **??**) would outweigh the latency benefits it seeks provide, so we formulate an efficient heuristic.

Flow Engineering Heuristic. Our goal is to satisfy a bound C on the time required to install/modify rules across all switches.

We represent the network as a graph $G = (V, E)$, where each node is a switch (or PoP) and each edge is a link (or tunnel). Given a traffic matrix M, the SDN application computes K candidate equal cost paths for each $(u, v) \in V$, where cost is defined in terms of the application's objective (e.g., average link utilization). We assume the application also assigns a priority $Pri(u, v)$ to each flow (u, v).

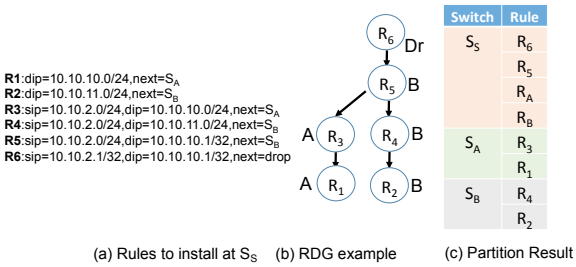


Figure 14: Rule offloading example

We sort the flows in decreasing order of resource demand (e.g., bandwidth) and iterate through them. For each flow (u, v) in the sorted order, we consider the corresponding K equal cost paths in decreasing order of resource availability; let $P^{1\ldots K}_{(u,v)}$ be the sorted order. If the resource demand d_{uv} can be satisfied by the path $P^1_{(u,v)}$, then we compute whether installing/updating rules for flow (u, v) along this path violates the latency bound C.

Given our measurement results, for every switch $s \in P^1_{(u,v)}$, we can model the latency at s due to installing rules for (u, v) as $L_s = \max(a, (b + c * Disp_s(Pri(u, v))))$. Here, $Disp_s(Pri(u, v))$ is the number of rules at s that will be displaced by the rule for (u, v). a, b and c are constants derived from switch measurements. This model essentially says that if the current rule does not displace any rules from s's existing table, then it incurs a fixed cost of a; otherwise, it incurs the cost given by $b + c * Disp_s(Pri(u, v))$. The fixed cost a is the insertion delay without any TCAM reordering.

Now, $\forall s \in P^1_{(u,v)}$, we check if $L_s + CurrentL_s \leq C$, where $CurrentL_s$ is the current running total cost of installing the rules at s, accumulated from vertex pairs considered prior to (u, v) in our iterative approach. If this inequality is satisfied, we assign (u, v) to the path $P^1_{(u,v)}$ and move to the next vertex pair. If not, we move to the next candidate path for (u, v), i.e., $P^2_{(u,v)}$ and repeat the process.

If after iterating through all flows once, we have not selected a feasible path for each flow, then we increase C and start from the beginning. Alternately, we could do a simple binary search on C.

Limitations. Because paths are computed by the SDN application, FE must be integrated with the application. FE does not apply to scenarios where route updates are confined to a single location, presenting no opportunity to spread update load laterally. One such example is MicroTE [**?**], where changes in traffic demands are accommodated by altering rules at the source ToR to reallocate ToR-to-ToR flows across different tunnels.

4.2 Rule Offload

When FE selects a path for a set of flows, the same number of rules are installed/modified in every switch along the path. Thus, a path's update latency is determined by the switch with the largest number of existing low priority rules (assuming BCM-1.0), or the largest number of pending rule installations/modifications. For example, since switch S_S in

Figure **??** is part of all three paths to reach switch S_D, a rule must be installed in S_S for every new flow, regardless of whether the flow traverses the path through S_A, S_B, or S_C. Assuming BCM-1.0 switches, it will take at least 0.3s to install rules for 100 flows of high priority H in S_S—the same time required to install 20 rules in S_A and longer than the ≈0.12s required to install 40 rules in both S_B and S_C.

The goal of *rule offloading* (RO) is to partition rules into subsets that can be installed at downstream switches, with the appropriate *default rules* installed at upstream switches. If the original number of rules is N and no partition (together with default rules) has more than H rules, then, by updating the partitions in parallel, we can reduce rule installation latency by a factor of $\frac{N}{H}$.

Our algorithm consists of two phases. First, given a set of rules slated for installation in a switch S (called the *root* switch), we partition the rules based on their next hop, taking into account any overlap between rules. Second, we compute default rules for each partition; these rules are installed in the root switch to direct packets to the next hop switch where the original rules are actually installed.

Rule Partitioning Algorithm. We represent the rules to be installed at a switch as a rule dependency graph (RDG). In an RDG, a node denotes a rule, an edge represents a dependency between two rules, and a node label indicates a rule's action (or next hop switch). For example, Figure **??**(b) depicts the RDG for the six rules in Figure **??**(a), which are slated for installation in switch S_S in the topology in Figure **??**. Note that when rules send packets over a tunnel, we specify the next hop in the tunnel path, not the tunnel destination, as the node label.

```
foreach R in reverseBFS(G) do
    i = label(R)
    if isLeaf(R) then
        if ruleCount(P_i) > H_down or
        ruleCount(S_i) > H then
            P_root += R
        else
            P_i += R
        end
    else
        if ∀ children(R) ∈ P_i and ruleCount(P_i)
        > H_down and ruleCount(S_i) > H then
            P_i += R
        else
            P_root += R
        end
    end
end
```

Algorithm 1: Rule Partition

The algorithm (Algorithm **??**) performs a reverse breadth-first traversal of the RDG. If a node R is a leaf node, then it is eligible to be placed in partition P_i, where i is R's label (or next hop). A leaf node R is placed in P_i if the current number of rules in P_i is less than H_{down}, otherwise R is placed in P_{root}. H_{down} controls the maximum number of rules we can offload to a switch downstream from the root switch. If a node R is not a leaf node, then it is only placed in partition P_i if all of its children are also in partition P_i, and the current number of rules in P_i is less than H_{down}. Otherwise, R is placed in P_{root}. The result is an allocation of rules $R \in RDG$ to the root switch (S) and its next hops.

In the example (Figure **??**), R_1 and R_3 are assigned to partition P_A, R_2 and R_4 are assigned to P_B, and R_5 and R_6 are assigned to P_{root}.

Computing Default Rules. Given the partitions for the next hop switches, we must compute a set of default rules that *cover* the rules in each partition. These default rules are added to P_{root} to forward packets to the appropriate next hop switch, where the rules in each partition (excluding P_{root}) are installed. The main challenge is dealing with the fact that the intersection of default rules may include rules from multiple partitions. Splitting the default rules into smaller rules can address this, but we must be careful not to introduce too may default rules and undo the benefits of RO.

Our heuristic from computing default rules is shown in Algorithm **??**. Given the rules in a pair of partitions P_i and P_j, we create a *covering rectangle* for the rules in each partition, denoted as C_i and C_j. A covering rectangle is one whose source IP range covers the entire source IP range specified in the rules in a partition; likewise for destination IPs. This can easily be extended to higher dimensions if rules are based on more than just source and destination IPs.

If the number of rules from either P_i or P_j in $C_i \cap C_j$ is below a threshold Θ, then all such rules are "promoted" to the root partition P_{root}. We also create two default rules, one each for C_i and C_j, and install them in the root switch S.

If, however, the number of rules in $C_i \cap C_j$ exceeds Θ, then we further divide both C_i and C_j into two sub-rectangles, and we repeat the process above for pairs of sub-rectangles, one from each partition. We recursively repeat the process for a small number of steps. If at the end of these steps, the combined number of default rules and rules in P_{root} is significant ($> \Omega$), then we merge P_i and P_j and simply install all of their rules at the root switch S.

```
foreach distinct partition pair (P_i, P_j) in P do
    get covering rectangles C_i and C_j for P_i and P_j
    checkOverlap(P, C_i, C_j)
end
myprocProcedure checkOverlap(P, C_i, C_j) if
ruleCount(C_i ∩ C_j) ≥ Θ then
    divide C_i and C_j into two sub-rectangles each
    foreach sub-rectangle pair (C'_i, C'_j) do
        checkOverlap(P, C'_i, C'_j)
    end
else
    move C_i ∩ C_j to P_root
    P_root += defaultRule(P_i, C_i)
    P_root += defaultRule(P_j, C_j)
end
```

Algorithm 2: Computing default rules

We recursively apply RO to the switches in a set of paths, starting at the ingress switch for the paths (e.g., S_S in Figure **??**), followed by the second switch in each path, and so on. The termination condition is that a set number of hops in each path are explored. If at termination, the number of rules accommodated at every switch, except the ingress switch, is $< H$, then we lower H by a factor $\gamma < 1$ and repeat again. If H^* is the value of H at the last of such iterations, then we achieve a speedup of $\frac{N}{H^*}$ from installing the offloaded rules in parallel. When running RO for an entire network, we sort ingress switches in decreasing order of the number of rules to be installed and apply RO in this order. For simplicity, we have assumed that all the switches have the same latency model; to accommodate switch diversity, we can assign different cost for the rules offloaded to different core switches.

4.3 Rule Reordering

Our measurements show that given rules of different priorities to be inserted at a switch, the "optimal" order of rule insertion varies with switch platform because of the difference in architecture and the workload the hardware is optimized for. For example, for Intel, the optimal order is to insert rules in *increasing* order of priority, whereas the *opposite* is true for Broadcom chip switches. Given this observation, rule reordering controls the actual rule insertion using the pattern that is optimal for the switch.

We assume one-shot consistent updates [**?**] are in use. In this case, new rules will not take effect unless all of them are installed. Therefore, RR can optimize the ordering without causing temporal policy violations. This technique can also be adapted for other update schemes [**?**].

5. EVALUATION

In this section, we use large-scale simulations with various topologies and workloads to study our techniques' effectiveness in meeting the needs of management applications that require low-latency control of data plane state. We consider three applications: failover in a tunneled WAN, two-level responsive traffic engineering, and MicroTE [**?**]. Our simulations leverage switch latency models derived from our measurements (§**??**).

5.1 Failover in a Tunneled WAN

We first evaluate the effectiveness of flow engineering (FE) and rule offload (RO) in the context of a control application that performs failover when a link fails in a tunneled WAN.

Topology. We use a simple full mesh (overlay) network of 25 nodes. The tunnels between these nodes share the same physical network. Each tunnel has between 5 and 10 intermediate switches. Per link capacity lies in $[100, 1000]$.

Workloads. We consider six workloads (Table **??**). For each workload, we assign a popularity index (random number within an internal) to each node. The number of flows between a pair of nodes is proportional to the product of their popularities. Each flow imposes a unit demand. At the start of our simulation, the traffic is routed such that the maximum load on any link is minimized.

Workload	Popularity index	Popularity index for high prio. traffic	# of flows between any pair of nodes	# of low prio. rules in flowtable
s1	1-10	1-5	Avg: 50 Max: 100	0-50
s2				100-200
s3				300-500
s4	1-20	1-7	Avg: 200 Max: 400	0-50
s5				100-200
s6				300-500

Table 3: Workloads used in simulation

Table occupancy. We assume that the new rules being installed upon failure (some of these could be updates to existing rules) all have the same priority P. Further, we assume that the tunnel end-points already have some lower priority rules, a subset of which are displaced by the new rules. We randomly pick the number of such displaced rules within some interval (defined for each workload in Table **??**). For simplicity, we assume that there are no dependencies across rules; we consider dependencies in subsequent sections.

To simulate failures we randomly select a tunnel in the mesh and fail it. On a link failure, about 70 flows are rerouted for low traffic workloads (*s1-s3*) and 220 for high traffic workloads (*s4-s6*). We assume that there is enough spare capacity in the network to reroute the affected flows. All rerouted flows are treated as new flows.

We consider three techniques for rerouting: (1) *Base case*, which reroutes the affected flows while minimizing the maximum link load, ignoring setup latencies. (2) Flow engineering (*FE*), which selects paths for affected flows such that flow installation latency is minimized (§**??**). (3) Flow engineering plus rule offloading (*FE +RO*), which applies FE and then offloads a set of rules from the tunnel end-nodes to at most $k = 3$ next hop switches per tunnel (§**??**).

In all cases, we assume that one-shot consistent updates [**?**] are employed to install routes. Thus, our metric of interest is the *worst case latency incurred at any switch to install all new/modified routes at the switch.*

We simulate with both BCM-1.0 and Intel, assuming all switches in the network are from the same vendor. Figure **??** shows the latencies with BCM-1.0 switches for the three techniques. For the lowest volume workload, the base case incurs a latency of 720ms, whereas FE improves this to 259ms and FE +RO to 133ms. These improvements are crucial, especially for latency sensitive interactive applications.

For the remaining workloads, base case latency varies between 2 and 14s. Using FE offers 22-35% improvement, but using FE together with RO leads to nearly a *factor of 3* improvement in all cases. Note that the gains can be improved further by: (1) leveraging more core switches for offload, and (2) providing a modest amount of reserved capacity for highly critical traffic, so that during failures the number of flows whose routes have to be recomputed is small and the rerouted non-critical flows can tolerate modest amounts of downtime or congestion. In other words, FE and RO pro-

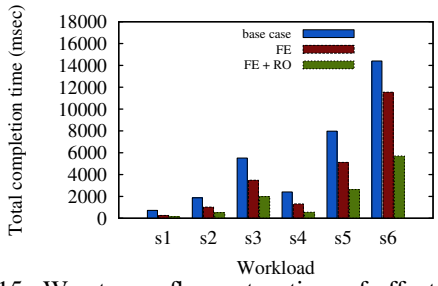


Figure 15: Worst case flow setup time of affected flows in the failover scenario with BCM-1.0 switches

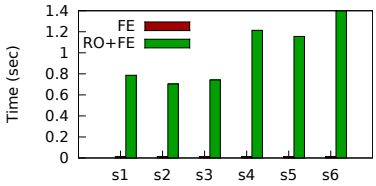


Figure 16: Running time for FE and RO

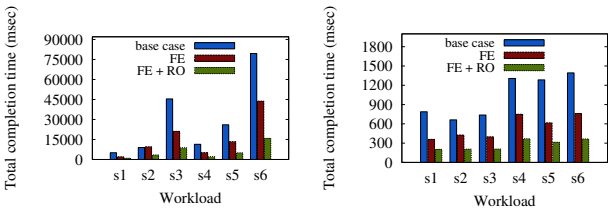
vide operators additional flexibility in designing schemes to better meet failover requirements in their networks.

Figure **??** shows the runtime overhead of FE and RO. FE takes $< 13ms$ for all workloads, while FE+RO takes up to 1.4s. However, even after taking into account this overhead, the net latency benefit of FE+RO for BCM-1.0 switches is still ≈0.3s to 6.6s, depending on workload. Furthermore, we focused on correctness, not efficiency, in designing our simulator, so there is still ample opportunity for improvement.

We also run our simulation with the Intel model. Since all rules we insert have the same priority, and the Intel switch does not impose rule displacement in such situations, the latency is purely driven by the maximum number of rules inserted at any switch. In our simulations, this is almost always at source end-point on a failed tunnel. Since both base case and FE are equally impacted by this, we don't see any improvement from using FE. However, RO still applies, as rules can be offloaded to core switches—we see an improvement of *over 2X* (324ms to 129ms).

5.2 Two-level Responsive Traffic Engineering

Next, we evaluate the effectiveness of FE and RO in the context of a control application that performs two-level responsive traffic engineering. This application simultaneously routes two classes of traffic—high and low priority—over the same network according to different objectives. For low priority traffic, the objective is to minimize the overall link utilization of the network due to this traffic; we install coarse grained (wildcard) rules to route this traffic. The objective for high priority traffic is to minimize the overall link latency; we install fine grained high priority rules to route this traffic. We first route the low priority traffic, and then route the high priority traffic using the remaining network capacity. We assume both categories of traffic can be accommodated without causing any congestion.



(a) Latency on BCM-1.0 switch (b) Latency on Intel switch

Figure 17: Worst case flow set up time in the two level traffic engineering scenario

We use the same topology and workloads described in §**??**. However, for high volume workloads (*s4-s6*) the volume of high priority and low priority traffic between any two overlay nodes is about 17 and 200 flows, respectively, and for low volume workloads (*s1-s3*) the volume is about 12 and 50 flows, respectively.

A network's ability to meet SLAs for each traffic class depends on how quickly the network can establish routes when requests for both classes arrive close in time. Figure **??** and **??** show the total completion time using BCM-1.0 and Intel switches, respectively, with and without our techniques. The base case has a significantly high flow set up time when the number of low priority rules in the table are high: as high as 80s for BCM-1.0. This implies that ignoring flow setup latency can cost traffic engineering dearly in terms of being responsive. For low volume workloads (*s1-s3*) the factor of improvement from just FE is about 2.5X for BCM-1.0 and 1.8X for Intel, and with FE +RO it's about 5X for BCM-1.0 and 4X for Intel. We observe similar speedups for high volume workloads (*s4-s6*).

5.3 MicroTE

MicroTE [**?**] leverages the partial and short term predictability of a data center traffic matrix to perform traffic engineering at small time-scales. As noted in §**??**, FE does not apply to MicroTE since routes span a single tunnel and route changes all happen at a single switch. Thus, MicroTE can only benefit from RO, the extent of which we now study.

We consider the following simple data center topology: we use a three-level FatTree topology [**?**] with degree 8, containing 128 servers connected by 32 edge, 32 aggregate, and 16 core switches. We assume that the traffic rate between a pair of servers is derived from a Zipfian distribution. Figure **??** shows the rule installation completion time. We see that RO provides a 2X improvement (400ms to 200ms) assuming the BCM-1.0 switch. Given the time-scales of predictability considered, this can help MicroTE leverage traffic predictability longer, thereby achieving more optimal routing. The improvement with Intel is 1.6X (80ms to 48ms).

6. RELATED WORK

A few prior studies have considered SDN switch perfor-

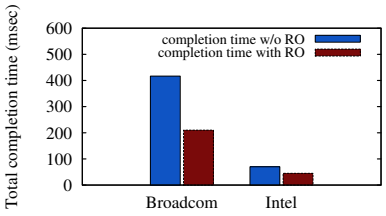


Figure 18: Flow setup time of MicroTE with and without RO in a Fat Tree (k=8) topology

mance. 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 [**?**] 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 [**?**, **?**] provided a more in-depth look into switch performance across various vendors. In [**?**], the authors evaluate 3 commercial switches and observed that switching performance is vendor specific and depends on applied operations, forwarding table management, and firmware. In [**?**], 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. We provide in-depth characterization of the impact of different factors (e.g., rule priority, complexity and table occupancy). We also provide low-level explanations of the latency causes.

Some studies have consider approaches to mitigate the overhead of SDN rule matching and processing. DevoFlow [**?**] 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. Our techniques are largely complementary to this. DIFANE [**?**] 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. However this approach does not apply for the kind of applications we are targeting that need to make fast, frequent updates/modifications to data plane state. vCrib [**?**] automatically partitions and places rules at both hypervisors and switches, but their goal is to reduce computational load on the host hypervisor, while we wish to reduce path setup latency by enabling fast parallel execution of updates. Lastly, Dionysus [**?**] optimally schedules a set of rule updates while maintaining desirable consistency properties (e.g., no loops and no blackholes).

7. CONCLUSION

Critical SDN applications such as fast failover and fine-grained traffic engineering demand tight interaction between switch control and data planes. However, our measurements across four OpenFlow-based switches show that the latencies underlying the generation of control messages (*packet_in*'s) and execution of control operations (*flow_mod*'s) can be quite high, and variable. We find that the underlying causes are linked both 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 issues, and concurrent switch activities). Our measurement study highlights the need of careful design of next generation switch silicon and software in order to fully utilize the power of SDN. Finally, to mitigate the challenges these latencies create for SDN in supporting critical management applications, we present three measurement-driven techniques. Our evaluation shows that these mechanisms can tame flow setup latencies effectively, thereby enabling SDN-based control of critical applications.

1. FLOW ENGINEERING FORMULATION

In this appendix we present an integer linear program for flow engineering (§**??**). We assume the SDN application performs traffic engineering in a data center with the goal of minimizing the maximum link utilization. Table **??** lists the notations used in the formulation.

S	Set of all switches
S_{ToR}	Set of all top-of-rack (ToR) switches
$\tau_u, \forall u \in S$	Maximum number of flow entries in the switch u
E	Set of all physical links (between adjacent switches)
$C_e, \forall e \in E$	Capacity of individual links
F_{uv}	Set of all flows from u to v where $u, v \in S_{ToR}$
P_{uv}	Set of paths from switch u to switch v
K_{uv}	Number of paths from switch u to v where $u, v \in S_{ToR}$
P_{uv}^k	Set of links of k^{th} path from device u to v where $u, v \in S_{ToR}$
T_{uv}^f	Traffic volume from u to v of flow f where $u, v \in S_{ToR}$
$util$	Maximum link utilization
$D_u \forall u \in S$	Number of low priority rules in switch u
I_{uv}^{fk}	Indicator variable denoting that flow f from u to v takes the k^{th} path

Table 4: Notations used in flow engineering formulation

1.1 Step 1 - Minimize "utilization"

In the first step, we minimize the maximum link utilization. We represent the network as a graph *G = (V,E)*, where each vertex is a switch and each edge is a physical link. Let T_{uv}^f be the traffic volume between (u, v) of flow *f*, P_{uv}^k be the set of links on the k^{th} path between (*u,v*).

We have two constraints: First, the total traffic volume from an edge should not exceed its capacity, C_e, times the link utilization limit, $util$. We use a binary indicator variable I_{uv}^{fk} to show whether or not the flow f is on path k.

$$\sum_{u,v \in S_{ToR},\ f \in F_{uv},\ k \in P_{uv}\ s.t.\ e \in P_{uv}^k} T_{uv}^f * I_{uv}^{fk} \leq util \times C_e \quad (1)$$

Second, each flow f should be routed from a single path.

$$\sum_{k \in P_{uv}} I_{uv}^{fk} = 1, \qquad \forall u, v \in S_{ToR},\ f \in F_{uv} \quad (2)$$

The objective is to minimize *util*, the maximum link utilization, under the above two constraints.

1.2 Step 2 - Minimize "rule setup latency"

Our measurements show that flow installation latency has a linear relation with the burst size of the rules (i.e., the number of rules to be installed), and, if rules are of a higher (or lower, depending on switch vendor) priority, installation latency increases with the number of rules that need to be displaced. Therefore, the second step is minimize the maximum number of rules in any switch. Flow installation latency can be defined as

$$latency_u = (a_u + c_u * D_u) * \tau_u$$

where a_u and c_u are constants for switch u derived from measurements.

There are three constraints: First, traffic traversing an edge e does not exceed the C_e multiplied by $1 + \delta$ times the $util^*$. $util^*$ is the maximum link utilization we got from the step 1, and δ defines an upper limit on maximum link utilization.

$$\sum_{u,v \in S_{ToR},\ f \in F_{uv},\ k \in P_{uv}\ s.t.\ e \in P_{uv}^k} T_{uv}^f * I_{uv}^{fk} \leq (1+\delta) \times util^* \times C_e \quad (3)$$

Second, constraint is same as step 1.

$$\sum_{k \in P_{uv}} I_{uv}^{fk} = 1, \qquad \forall u, v \in S_{ToR},\ f \in F_{uv} \quad (4)$$

Third, the number of flows passing through a switch s is less than the number of maximum rules in any switch in the network.

$$\sum_{u,v \in S_{ToR},\ f \in F_{uv},\ k \in P_{uv}\ s.t.\ s \in P_{uv}^k} I_{uv}^{fk} \leq \tau_u \qquad \forall s \in S \quad (5)$$

The objective is to minimize the $latency_u$, the maximum flow installation latency, under above three constraints.