Latency in Software Defined Networks: Measurements and Mitigation Techniques

Keqiang He†, Junaid Khalid†, Sourav Das†, Aaron Gember-Jacobson†, 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 such as fast rerouting during link failure and reactive path setup for latency-sensitive flows. However, our measurement study using three types of production SDN switches shows that the interaction latencies such as rule installation time are significant. This is due to both software implementation inefficiencies and fundamental traits of underlying hardware. Simulations based on our measurements show that the high latencies render SDN incapable of supporting key control applications. To systematically overcome the latencies and achieve responsive control, we develop a system called Mazu. Mazu eliminates some components of latency, and mitigates others, via a mixture of clever techniques, including offloading certain tasks to proxies, engineering route computations and reordering rules to minimize per-switch update latencies, and spreading rules across switches to enable network-wide parallel updates. Controlled simulations and testbed experiments show that our techniques can reduce the latency to update network state by almost 5X, making SDN-based control suitably responsive for critical management applications.

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 [6, 17]. 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 [5, 8, 9, 12, 13, 15, 20, 21].

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 [5]. Similarly, setting up routes in cellular networks (when a device becomes active, or during a handoff) must complete within $\sim$30-40ms to ensure users can interact with Web services in a timely fashion [13].

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 [14] 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 three types of production SDN switches from two different vendors—Broadcom and Intel—using a variety of workloads. We investigate the relationship between switch design and observed latencies using both 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). 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 the three switches, 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 [11,23]). 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. To this end, we design a framework called *Mazu* that leverages SDN's central view and global control. To mitigate inbound latency arising from bus bottlenecks in switches, Mazu redirects relevant packets to a fast proxy that generates the necessary messages for the controller. This virtually eliminates inbound latency and saves switch resources for rule updates.

To mitigate outbound latency, which is intrinsically linked to switch design (especially hardware), we propose three techniques: *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 installations 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, and the impact of outbound latency is minimized.

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 in/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 AND MOTIVATION

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 are frequently 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 [17]—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; we articulate these steps below. We then highlight several applications whose efficacy is heavily impacted by the latency of these control operations on switches.

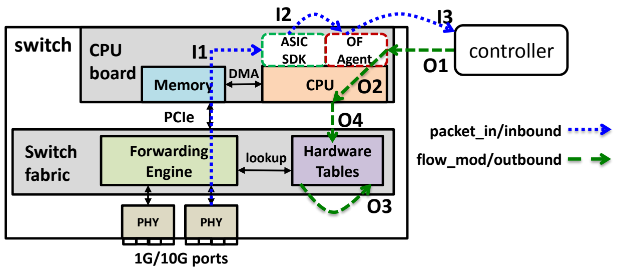


Figure 1: Schematic of an OpenFlow switch. We also show the factors contributing to inbound and outbound latency

2.1 Components of Latency

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 1): (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.

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.

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*.

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

Application Design. Reactive applications enforce default-off forwarding to flow sub-spaces; this causes any flow in that sub-space to generate a *packet_in* message when it reaches an ingress switch for the first time. The application then determines the forwarding action and sends the corresponding *flow_mod* messages to the switch. These applications are impacted by both in- and outbound latency. Proactive applications directly update forwarding state using *flow_mod* messages. They are mainly impacted by outbound latency.

2.2 Motivating Applications

We now provide examples of both reactive and proactive management applications that require fine-grained control over data plane state. We highlight the impact of inbound and outbound latencies on each application's objectives.
Mobility (reactive). Recent work [13] advocates using SDN to simplify path setup and management in cellular networks. When mobile devices want to access the Internet, GPRS Tunneling Protocol (GTP) tunnels need to be set up, and reconfigured during handoff. Crucially, these tunnels must be setup within a small latency bound. For example, when a mobile device in an idle state wants to communicate with an Internet server it needs to first transition from idle to connected; according to recent measurement studies [10], the state transition delay is about 260 ms in LTE. To keep access latency below 300 ms, as recommended by Web service providers, path setup must finish in 40 ms. This can be very challenging especially when paths for multiple devices need to be setup at once, e.g., during a popular event.
Blocking Malicious Web Requests (reactive). Applications like HP Network Protector [1] use SDN to intercept DNS queries from hosts, and drop, forward, or redirect the request according to global policies. Hosts issuing many malicious requests can be quarantined by installing high priority rules that drop traffic from those hosts. As in the mobility scenario, timely interaction between the controller and switches is critical to avoid impacting Web access latencies.
Failover (proactive). It is possible that SDN can help mitigate the network-wide impact of failures in wide-area networks, reducing both downtime and congestion without requiring significant over provisioning. When failures occur, the SDN management application can quickly compute new paths for flows traversing failed nodes or links, while also simultaneously rerouting other high/low priority flows so as to avoid hot-spots [9]. However, this requires significant updates to network state at multiple network switches. The longer these updates take, the longer the effect of failure is felt in the form of congestion and drops. We find that outbound latencies can inflate the time by nearly 20s (§7) putting into question SDN's applicability to this scenario.
Intra-Datacenter Traffic Engineering (proactive). MicroTE [5], Hedera [4], and other recent proposals have argued for using SDN to route traffic subsets at fine time-scales in order to achieve fine-grained traffic engineering in data centers. For instance, MicroTE leverages the fact that a significant fraction of ToR-to-ToR DC traffic (ToR is "top-of-rack" switch) is predictable on short time-scales of 1-2s. It computes and installs at ToR switches routes for such traffic on short time-scales. Thus, latencies in installing routes can significantly undermine MicroTE's effectiveness. Indeed, we find that updating a set of routes at a ToR switch in MicroTE can take as long as 0.5s on some SDN switches (§7).

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)	

Table 1: Switch specifications

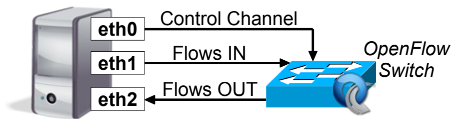


Figure 2: Measurement experiment setup.

3. LATENCY MEASUREMENTS

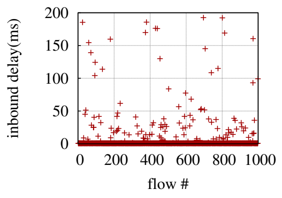
In this section, we systematically measure in/outbound latencies to understand what factors contribute to high latencies. We generate a variety of workloads to isolate specific factors, and we use production switches from two vendors, running switch software with support for OpenFlow 1.0 [17] or, if available, OpenFlow 1.3, both to highlight the generality of our observations and to understand how software evolution impacts latencies.[2] Henceforth, we refer to the three hardware and software combinations (Table 1) as Intel, BCM-1.0, and BCM-1.3. 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

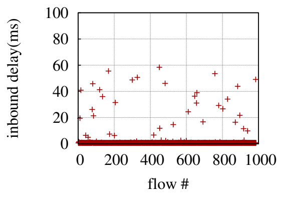
Figure 2 shows our measurement setup. The host has one 1Gbps and two 10Gbps interfaces that are connected to the switch under test. The eth0 interface is connected to the control port of the switch, and an SDN controller (POX for Intel and BCM-1.0, RYU for BCM-1.3) running on the host listens on this interface. The propagation delay between switch and controller is negligible (about 0.03ms). We use the controller to send a burst of OpenFlow *flow_mod* commands to the switch. For Intel and BCM-1.0, 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 [19] 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 ob-

[2]When using the OpenFlow 1.3 firmware, we only leverage features also available in OpenFlow 1.0 to allow 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

served in the data plane [23]. 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* message on eth0. The difference ($t_2 - t_1$) is the inbound latency.[3]

Representative results for these two experiments are shown in Figure 3a and 3b, respectively, for the Intel switch; 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 3a).

[3]This measurement technique differs from the approach used in [11], where the delay was captured from the switch to the POX controller which includes the overhead at the controller.

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

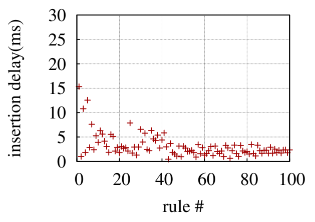
We first examine how different rule workloads impact insertion latency. We insert a burst of B rules: $r_1, \cdots, r_B$. Let $T(r_i)$ be the time we observe the first packet matching r_i emerging from the output port specified in the rule. We define per-rule insertion latency as $T(r_i) - T(r_{i-1})$.

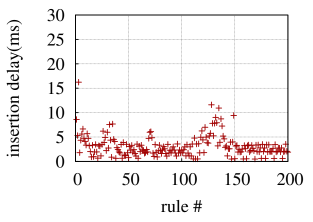
Rule Complexity. To understand the impact of rule complexity (i.e., the number of header fields specified in a rule), we install bursts of rules that specify either 2, 8, or 12 fields. In particular, we specify destination IP and EtherType (others wilcarded) in the 2-field case; input port, EtherType, 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.

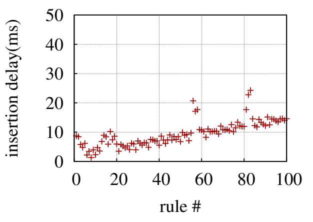
We find that rule complexity *does not* impact insertion latency. The mean per-rule insertion delay for 2-field, 8-field, and exact match cases is 3.31ms, 3.44ms, and 3.26ms, respectively, for BCM-1.0. Similarly, the mean per-rule insertion delay for BCM-1.3 is $\approx$ 1 ms irrespective of the number 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 the flow table occupancy *does not* impact the insertion operation if all rules have the same priority. Taking $B = 400$ as an example, the mean per-rule insertion delay is 3.14ms, 1.09ms, and 1.11ms, and the standard deviation


(a) burst size 100, same priority


(b) burst size 200, same priority


(c) burst size 100, incr. priority

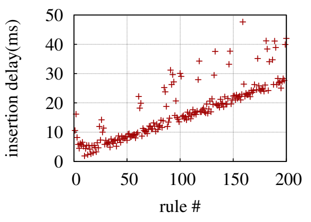

(d) burst size 200, incr. priority

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

is 2.14ms, 1.24ms, and 0.18ms for BCM-1.0, BCM-1.3, 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.

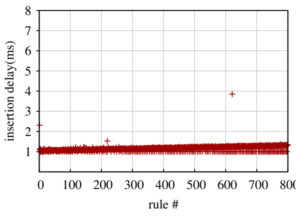
BCM-1.0, same priority. Representative results for $B = 100$ and $B = 200$ are shown in Figure 4a and 4b, respectively. In both cases, we see that the per-rule insertion delay is similar: with medians of 3.12ms and 3.02ms, and standard deviations of 1.70ms and 2.60ms, for $B = 100$ and $B = 200$, respectively. We conclude that same priority rule insertion delay does not vary with burst size on BCM-1.0.

BCM-1.0, increasing priority. Figure 4c shows the result for $B = 100$. We note that the per-rule insertion delay actually *increases linearly* with the number of rules inserted. Figure 4d 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.


(a) burst size 800, incr. priority

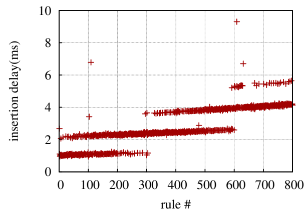

(b) burst size 800, decr. priority

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

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.

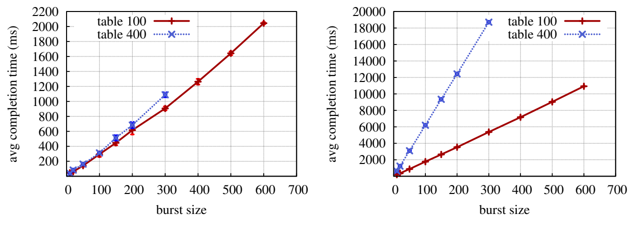
Intel, same priority. For $B = 800$ on Intel[4] we see that the per-rule insertion delay is similar across the 800 rules, with a median of 1.17ms and standard deviation of 0.16ms (not shown). The results for other values of B are similar. Thus, similar to BCM-1.0 and BCM-1.3, same priority rule insertion delay does not vary with burst size on Intel.

Intel, increasing priority. Figure 5a shows per-rule latencies for $B = 800$. *Surprisingly*, in contrast with BCM-1.0 and BCM-1.3, 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 5b 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

[4]We present results for a larger value of B because the flowtable size on Intel is larger (Table 1).



(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.

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: $A_1 \to A_2 \to A_3$, and so on. We confirmed this with Intel.

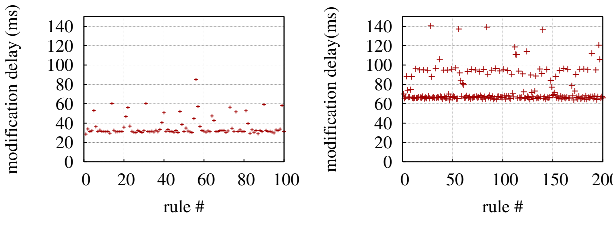
Priority and table occupancy combined effects. We now study the combined impact of rule priority and table occupancy. We conduct two experiments: For the first experiment, the table starts with S high priority rules, and we insert B low priority rules. For the second experiment, the priorities are inverted. 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 and BCM-1.3, 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 Intel, the results are similar to same priority rule insertion. This indicates that Intel optimizes for rule priority better than BCM-1.0.

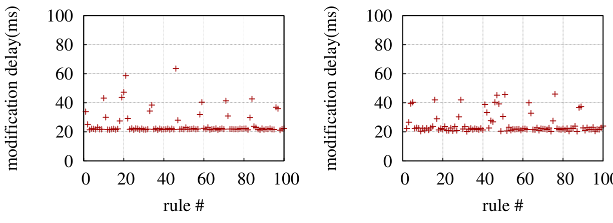
Summary and root causes. We observe that: (1) rule complexity does not affect insertion delay; (2) same priority insertions in BCM-1.0, BCM-1.3, and Intel are fast and not affected by flow table occupancy; and (3) priority insertion patterns can affect insertion delay very differently. For Intel, increasing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



(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

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 [2]). 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

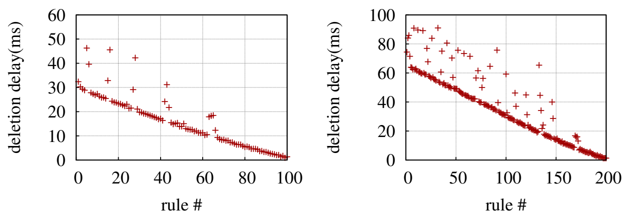
We now study modification operations. As before, we experiment with bursts of rules. Modification latency is defined similar to insertion.

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 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).

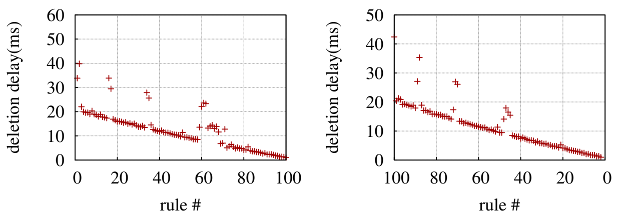
In contrast, Intel and BCM-1.3 have lower modification delay, and it does not vary with table occupancy. For Intel (BCM-1.3) the per-rule modification delay for both $S = 100$ and $S = 200$ is around 1 ms (2ms) for all modified rules, similar to (2X more than) same priority insertion delay.

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.



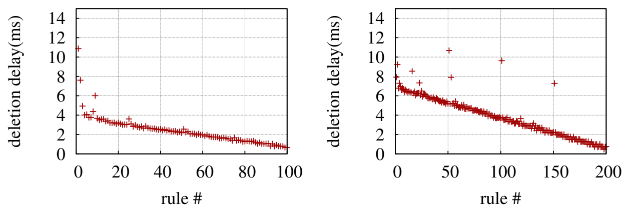
(a) 100 rules in table (b) 200 rules in table

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



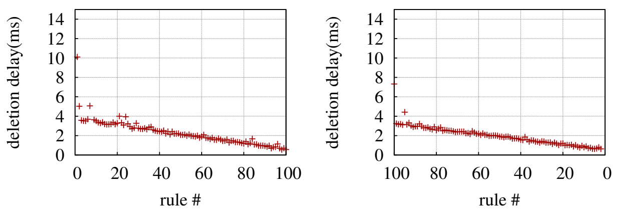
(a) increasing priority (b) decreasing priority

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



(a) 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 12: **Intel** priority per-rule **del.** latency, B=100

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 and Intel (not shown).

Summary and root causes. We conclude that the per-rule modification latency on BCM-1.0 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 poorly 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, 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) and BCM-1.3 (figure not shown).

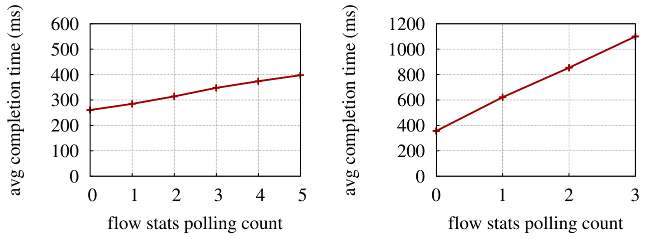
Rule Priorities. We start with B existing rules in the switch, and delete one rule at a time in increasing and decreasing priority order. For BCM-1.0 (Figure 11), BCM-1.3 (figure not shown) and Intel (Figure 12), deletion is not affected by the priorities of rules in the table or the order of deletion.

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.

3.3.4 Impact of concurrent switch CPU jobs

We previously showed the impact of *flow_mod* processing on inbound latency (§3.2). We now study the impact of statistics polling on outbound latency. Figure 13 shows that concurrent activities, such as polling statistics, can have a big impact on insertion delay, especially when table occupancy is high: e.g., with BCM-1.0, the total time to insert a burst of 100 same priority rules into a table with 500 rules is 853ms when two polling events occur during the insertion process, compared to 356ms without polling events.

3.4 Implications



(a) init. table occupancy 0 (b) init. table occupancy 500

Figure 13: Impact of polling on **BCM-1.0**, B=100

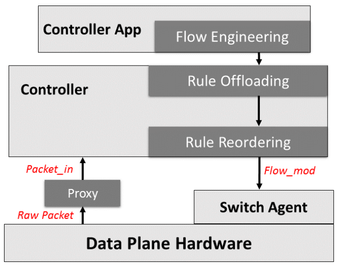


Figure 14: Mazu framework

Prior work [11,23] 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 supported natively by TCAM hardware [2]. We believe that near term work will chip away at such issues, as indicated by improved latencies in BCM-1.3. That said, given that software "glue" will continue to exist between control and data planes in SDN switches, we remain skeptical whether the 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 that these fundamental 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.

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. In the next section, we present our solutions that meet such immediate needs.

4. MAZU OVERVIEW

Our goal is to develop a general set of techniques that an SDN network can employ to overcome the impact of the latencies described above on key management applications. Ideally, the techniques must work across all applications, switches and deployment settings. To this end, we present a new controller framework called Mazu (Figure 14).

To eliminate all inbound delays, and some outbound delays (namely, *packet_out* procesing), we introduce a proxy that generates *packet_in* and processes *packet_out* messages on behalf of one or more switches.

Because the underlying causes of outbound delays are tightly linked with switch software and hardware, we can hope at best to mitigate these latencies. The remaining modules achieve this. The key insight underlying them all is to organize the *flow_mod* input provided to switches such that the aggregate rule installation latency experienced by the application is minimized, given the underlying latency causes.

Flow engineering is an application-dependent module that computes routes that spread flows across paths in a network, so as to minimize rule installation latency by controlling rule displacement at any switch, while adhering to network objectives (§6.1). *Rule offloading* takes the set of rules to be installed at any ingress switch as an input (these could be rules computed by the flow engineering step above), and carefully offloads/spreads subsets of these rules to downstream switches/routers having sufficient capacity to hold the rules (§6.2). By virtue of reducing the installation latency per switch and enabling parallel execution of updates, these techniques ensure rule update tasks finish much faster.

Rule Reordering module reorders the rules to be installed at a switch (e.g., those computed by rule offload scheme above) into a sequence that is optimal with respect to the switch's hardware table management scheme. This helps further control rule installation latency (§6.3).

5. HANDLING INBOUND DELAY

Inbound latency can have a pronounced impact on reactive applications such as those described in §2.2. To overcome the primary factor contributing to this latency—limited bus bandwidth between the switch ASIC and CPU (§3.2)—we introduce physically decoupled processing units, i.e., custom proxies, that generate *packet_in* and process *packet_out* messages.

Proxies are co-located with the ingress (edge) switches in the network, and we establish a (short) label-switched path between each switch and its corresponding proxy. The switches continue to have a control channel to the controller. We then insert OpenFlow rules in the switches that forward to their proxy, at line rates, all packets that would have otherwise required them to generate *packet_in* messages. The switches stamp the incoming port ID in the ToS field of the packets before forwarding them to the proxy; this can be achieved by installing one rule for each switch port which matches based on in-port and sets the packet's ToS field accordingly. The proxy then generates the necessary *packet_in* messages reflecting the switches' incoming port ID, and forwards them on its control channel to the controller. Similar to a regular SDN switch, the proxy also locally buffers the packets. The controller sends any *packet_out* messages to the proxy, which processes them and forwards the corresponding buffered packets back to the switch to be routed to their eventual destination. All *flow_mod* messages are sent directly to the switch by the controller.

A single proxy, as we will show in §7.1 is capable of serving multiple switches and is thus cost-effective. We also show that it drastically reduces inbound latency. As an added

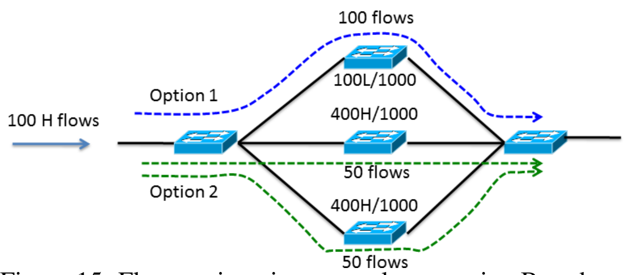


Figure 15: Flow engineering example, assuming Broadcom. NL = N low priority rules; NH = H high priority rules; capacity = 1000 rules. Ingress and egress tables are empty

benefit, more switch CPU resources are available for processing *flow_mod*s, polling statistics, etc.

6. MINIMIZING OUTBOUND DELAY

We describe three Mazu modules for overcoming outbound latencies. Our approaches deal mainly with rule insertions. To handle rule deletions and modifications, we leverage the following key ideas based on our measurement results:

1. *Avoid deleting rules:* Rule deletions are expensive across all the platforms we measured. Thus, *we try to avoid rule deletion*. Instead, we simply let them time out (and insert higher priority rules to supersede them as needed).

2. *Avoid modifying rules for Broadcom*: Our measurements with both Broadcom switches showed that modifying a rule is more expensive than inserting a new rule. Therefore, we *always insert* a new rule R' for a flow at a switch instead of modifying the existing R to R'. We ensure R' is of higher priority than R but lower priority than any R'' that overlaps with R' and is higher priority than R. We simply let R expire (similar to above). A nice side-effect of this is that rule priorities generally "stay high", resulting in lower rule displacements from future insertions compared to modifying rules (as only higher priority rules can cause displacements in Broadcom). For the Intel switch, modification latency is small and independent of rule priorities in the flow table, so no such provision needs to be made.

6.1 Flow Engineering

SDN applications typically compute paths between various network locations that meet some global objective pertaining to performance or security. A common issue considered in most prior works on such applications is to deal with limited switch table sizes, by picking routes that obey or optimize table space constraints [9,18,21]. Unfortunately, these techniques do not provide sufficient control over outbound delay.

Minimize maximum flow table occupancy is very suboptimal. For example, consider a simple setting where there are three candidate paths between a pair of nodes as shown in Figure 15. Each path has one Broadcom switch. The switch on the first path has 100 rules of low priority L, whereas the switches on the second and third paths each have 400 rules of high priority H. Suppose that a hypothetical traffic engineering application has 100 flows of priority H to allocate to these paths, and each path is equally preferable for a flow. Existing techniques for table space management would assign all flows to the first path to minimize maximum flow table occupancy; but our measurements for Broadcom show that *each* of these 100 rules will displace all the 100 low priority rules in the TCAM, resulting in high latencies! Allocating 50 flows each on the latter two paths instead results in no rule displacement, and the number of rules installed per path will be smaller. Thus, when the flows are installed in parallel across the latter two paths, this results in significant reduction in installation latency. Based on Figure 6, it is about 200 ms vs 2 seconds, *a 10X difference!* Intel switches have similar issues, but for other priority patterns.

The goal of flow engineering is to select paths across the network such that installation delay is minimized. The key insight we use is the following: in general, there are many possible sets of paths $\{\mathcal{P}^i_{obj}\}_i$ in a network that optimize an SDN application's objectives, e.g., optimal capacity and latency. From this, flow engineering selects the set $\mathcal{P}^{displace}_{obj,tbl\_sz}$ that minimizes the aggregate impact of both rule displacement in TCAM as well as the number of rules installed at any switch, while obeying table space constraints. $\mathcal{P}^{displace}_{obj,tbl\_sz}$ can be computed by running a two step optimization, where the first step computes the value of the network's objective function, but not the actual routes to use, and the second step computes routes that minimize the aggregate effect of rule displacement and the number of rules to be inserted at any switch. The detailed optimization formulation is a large integer linear program (omitted for brevity) and hence inefficient to solve. Below, we discuss a simplifying heuristic in the context of a traffic engineering application.

We represent the network as a graph *G = (V,E)*, where each node is a switch (or a PoP) and each edge is a physical link (or virtual tunnel). Given a traffic matrix M, the application attempts to route it such that the average link utilization is within some bound; the heuristic can be easily extended to accommodate other objectives. For each source-destination pair we see, whether a path can accommodate both its demand, and the path setup latency is within some bound. If either is violated, we try the next candidate path.

More precisely, suppose we want to bound the maximum cost of installing rules at any switch by some C. We start by selecting some low value for C. We assume that we have computed K candidate equal cost paths for each $(u, v) \in V$. Suppose the priority of the (u, v) flow is $Pri(u, v)$ at every switch in the network (this is typically set by the operator).

We sort the traffic demands in decreasing order and iterate through them. For each (u, v) in the sorted order, we consider the corresponding K equal cost paths in decreasing order of available capacity; let $P^{1...K}_{(u,v)}$ be the sorted order.

If the demand d_{uv} can be satisfied by the path $P^1_{(u,v)}$ within the utilization bound, then we compute whether installing the (u, v) path violates the rule installation latency bound or not. We do this by modeling the per-switch latency, as well as maximum latency on the path:

Per-switch latency. Given our measurement results, for every switch $s \in P^1_{(u,v)}$, we can model the latency at s due to routing (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). For the Broadcom switch, this is the number of rules of priority *lower* than $Pri(u, v)$, whereas for the Intel switch $Disp_s(Pri(u, v))$ is the number of rules of priority *higher* than $Pri(u, v)$ divided by 300. This is a conservative estimate assuming all rules are packed in increasing priority of slices (§3.3.1). $Disp_s(Pri(u, v))$ can be easily tracked by the SDN controller. In the above, a, b ad c are constants derived from our 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 ordering. a is the same whether it is modification or insertion for Intel. For Broadcom, since we avoided modification, it represents insertion delay without TCAM displacement.

Maximum installation latency. 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 source-destination pairs considered prior to (u, v) in our iterative approach.

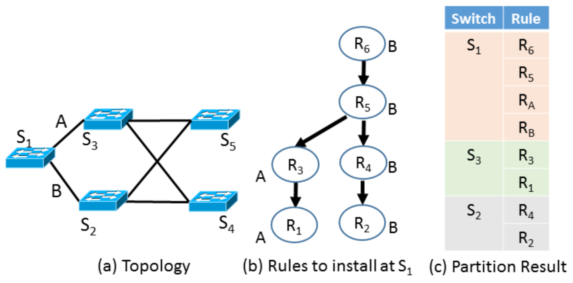
If this inequality is satisfied, we assign (u, v) to the path $P^1_{(u,v)}$ and move to the next source-destination pair. If not, meaning that installing the (u, v) route on this path violates the maximum cost bound C for some switch on the path, then we move to the next candidate path for (u, v), i.e., $P^2_{(u,v)}$ and repeat the same as above.

If after iterating through all (u, v) pairs once, the traffic matrix cannot be allocated, then we increase C and start over again. Alternately, we could do a simple binary search on C.

Comments. Because the paths are computed by the SDN application, flow engineering will necessarily have to be implemented within the application. Flow engineering does not apply to scenarios where route updates are confined to just one location, presenting no opportunity to spread update load laterally. One such example is MicroTE [5] (§2), where changes in traffic demands are accommodated by altering rules at the source ToR to reallocate ToR-to-ToR flows across different tunnels.

6.2 Rule Offload

Rule offloading applies particularly to networks where tunnels are used, e.g., cellular networks (§2), carrier networks that rely on label-switching, data centers using VXLAN and inter-DC WAN networks such as those consider in [9, 12]. In such networks, SDN applications control the tunnel endpoints to setup overlay paths. Compared to the rate of changes at these tunnel end-points, the underlay, which may also be run using an SDN, maintains much smaller forwarding state, and observes much less churn in forwarding state. *Our approach leverages these attributes of switches in the underlay to offload to them rules that would otherwise be installed at the tunnel end points.*



Switch	Rule
S_1	R_6
	R_5
	R_A
	R_B
S_3	R_3
	R_1
S_2	R_4
	R_2

Figure 16: Rule offloading example

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

The main idea in our algorithm is to recursively partition the rules into a number of child partitions. Since we offload to next hop switches, each partition has an associated next hop. Rules in the same partition all go to the same next hop. The partition algorithm ensures that there is no dependency among child partitions. For each partition, we also compute default rules to direct packets to that partition. The objective is to maximize the number of rules that can be offloaded minus the number of default rules introduced.

Different from [18]'s goal of reducing computational load of host hypervisor, our goal in rule offloading is to reduce path setup latency by enabling fast parallel execution of updates. Also, per source rule offloading is considered in [18]. In contrast, we offload by grouping the next hop of rule actions to increase offloading opportunities.

We illustrate this using an example in Figure 16. Figure 16(a) shows the topology. Suppose we need to install six rules R_1, R_2,$\cdots$,R_6 to switch S_1. The rule dependency graph is shown in Figure 16(b). If there are rule entries in the flow table, the dependency graph will include those rules. For example, there is an edge from R_3 to R_1. This means that the two rules overlap. When a packet matches both rules, R_3 takes precedence. The labels A and B denote the next hop of the rules' action. If a rule's action is to send through a tunnel, the label will be the next hop of the tunnel path, not the tunnel destination. If a rule's action is deny, for simplicity, it will not be offloaded.

The algorithm starts from the leaf nodes (rules R such that there is no R' with $R \rightarrow R'$). All of them with the same next hop are placed in one partition. In the example, we have two next hops S_3 and S_2 through port A and B respectively. We have two leaf rules R_1 and R_2. R_1's next hop is S_3 and R_2's next hop is S_2. R_1 will be in partition 1 and R_2 will be in partition 2. Since we have $R_3 \rightarrow R_1$ and R_3's next hop is the same as R_1 (which is S_3 through port A), and R_2 (nexthop S_2) and R3 have no dependency, then R_3 will be in partition 1. Similarly, R_4 will be in partition 2. For R_5, $R_5 \rightarrow R_3$, and $R_5 \rightarrow R_4$; thus, R_5 has to be in the root partition ("pinned" to the ingress switch S_1). Also all rules R' such that $R' \rightarrow R_5$ will be pinned down in a similar fashion. R_6 is such a rule. So R_6 will be in the root partition.

The outcome of the above routine is an allocation of rules to the root (ingress switch S_1), and to its two next hops. Because we need to direct traffic to the appropriate next hop for offloading, we need to create default rules to cover the flowspace of the partitions. Suppose one rule R_A covers the flowspace of partition 1 and one rule R_B covers the flowspace of partition 2. The final rules to install at switches S_1, S_2, S_3 is shown in Figure 16(c). Four rules will be installed in switch S_1 and two each will be installed at switches S_2, S_3 respectively. This reduces the number of rules to install at switch R_1 by one third.

We start by picking a bound $H < N$ for the rules at any switch, where N is the total number of rules we started with that were to be installed at the edge switch. We also use a bound H_{core} that controls the maximum number of rules *any* edge switch can offload to a given core switch. If at any iteration, partitioning at a node causes either of these bounds to be violated at a downstream core switch, then we terminate partitioning for the node.

We then run the above routine recursively starting at the edge switch, followed by running it at the next hop core switches over the rules allocated to them, and so on. The termination condition is that a set number of next hops (downstream switches in the tree rooted at the edge switch) are explored. If at termination, the number of rules accommodated at every core 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 offload rules in parallel.

When running this scheme across the network, we sort edge nodes in decreasing order of rules to be installed and run the above algorithm on them in this order.

For ease of description, in the above algorithm, we do not account for switch table occupancy or consider the detailed delay model as in Section 6.1. To accommodate table occupancy, we can stop rule offloading process on a particular switch if the occupancy level will exceed a threshold. To avoid high delays due to rule structure in core switches, we apply the detailed delay model to our partition results. If the estimated delay is higher than no offloading because of a particular core switch, we will remove that switch from consideration and rerun the algorithm. It is also easy to consider the delay model directly in our algorithm as we have done for flow engineering in Section 6.1. However, for simplicity, we omit the details.

Next, we discuss computing default rules to direct traffic to downstream switch for eventual processing.

Computing default rules: Given two partitions A and B computed above, we wish determine default rules that need to go into A and B's root partition. The main challenge is dealing with the fact that the intersection of default rules may have rules from either A or B. This introduces ambiguity at the root. Splitting up the default rules into smaller parts to try and deal with this may introduce too may default rules. We present a heuristic optimization, described briefly below.

```
//G: rule dependency graph with nodes annotated with next hop label
//$\mathcal{P}_i$: partition $i$ for next hop $i$, initially empty
//$C_i$: set of rules covering the flowspace of partition $i$
//$N$: threshold for extra covering rules
While (BFS from leaf node) { //traverse reverse edges
  If rule $R_i$ with label $L_i$ depends on no other rules,
      include $R_i$ in $\mathcal{P}_{L_i}$
  Else If rule $R_i$ depends on rules with more than one distinct label
      pin the rule to the root partition
  Else
      If rule $R_i$ results in $n > N$ covering rules,
          skip $R_i$
      Else include $R_i$ in $\mathcal{P}_{L_i}$
}
```

Figure 17: Recursive Rule Partition Algorithm

We assume that each rule can be represented by a rectangle (src IP, dst IP) for simplicity. Our heuristic below can be easily extended to higher dimensions. Given the rules in A and B, we create *covering rectangles* one each for the rules in A and B, called C_A and C_B. A covering rectangle is one whose src IP range covers the entire src IP range specified in the rules (likewise for destination IPs).

We check if the number of rules from either A or B in $C_A \cap C_B$ is below some threshold Θ. If so, all such rules are "promoted" to the root partition and get pinned there. Furthermore, we create two default rules, one each for C_A and C_B and install them in the root.

If, however, the number of rules in $C_A \cap C_B$ exceeds Θ, then we further divide C_A and C_B in two sub-rectangles each. We repeat the process above for pairs of sub-rectangles one corresponding to A and the other to B.

We recursively repeated this process for a small number of steps. If at the end of these steps, the combined number of default rules and pinned rules to be installed at the root is significant ($> \Omega$), then we merge A and B and simply install all of it at the root.

6.3 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 Intel, the optimal order is to insert rules in *increasing* order of priority, whereas the *opposite* is true for Broadcom. Given this observation, Mazu controls the actual rule insertion using the pattern that is optimal for the switch.

We assume one-shot consistent updates [22] are in use. In this case, new rules will not take effect unless all of them are installed. Therefore, Mazu can optimize the ordering without causing temporal policy violations. Mazu's techniques

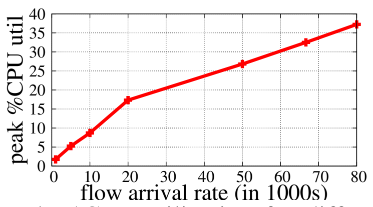


Figure 18: Peak %CPU utilization for different *packet_in* generation rate

can also be adapted for other update schemes [16].

7. EVALUATION

In this section, we conduct a thorough analysis of the effectiveness of Mazu's modules toward mitigating flow setup latencies. Ultimately, our goal is to understand whether Mazu can help SDN be more effective in providing fine time-scale control over network state.

7.1 Inbound Latency

We prototyped the proxy described in §5 on a commodity host (Intel quad-core 2.66GHz CPU, 8GB RAM). We measured the inbound latency using the same setup as described in §3.2; the only difference is that the switch now forwards packets at line rate to the proxy, which generates *packet_in* messages. At a flow arrival rate of 200/s, we recorded an average inbound latency of 0.199ms and a 99th percentile latency of 0.476ms. Even at a much higher flow rate of 2000/s, we recorded values of 0.146ms and 3.56ms for average and 99th percentile inbound latencies respectively. Compared with the status quo (average=8ms, 99th percentile=192ms for 200flows/sec) this improvement is substantial, especially for latency sensitive applications like VoIP calls in cellular networks. We also note that other operations like *flow_mod* on the switch do not impact inbound latency as the proxy is physically decoupled from it.

Along with the significant reduction in inbound latency, the solution also needs to scale to be practically viable. Figure 18 shows that the peak %cpu utilization of the proxy at different *packet_in* generation rates only increases linearly. Furthermore, our prototype is capable of generating 80,000 *packet_in* messages per second. This enables a considerably high number of switches to use a single proxy, making it a scalable, cost-effective, and efficient immediate solution.

7.2 Outbound Latency

In what follows, we use large-scale simulations with various topologies and workloads to study Mazu's effectiveness in meeting the needs of management applications that require fine-grained, low-latency control of data plane state. We consider three such applications: failover in a tunneled WAN, two-level responsive traffic engineering, and MicroTE [5]. Our simulations leverage switch latency models derived from our measurements in §3.3.

7.2.1 Failover in a Tunneled WAN

Work-load	**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

We first evaluate the effectiveness of Mazu's FE and RO techniques 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 3). 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.

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 3). 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 (§6.1). (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 (§6.2).

In all cases, we assume that one-shot consistent updates [22] 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 19 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

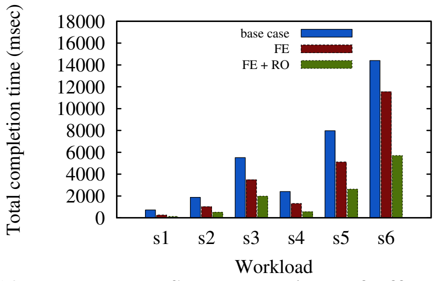
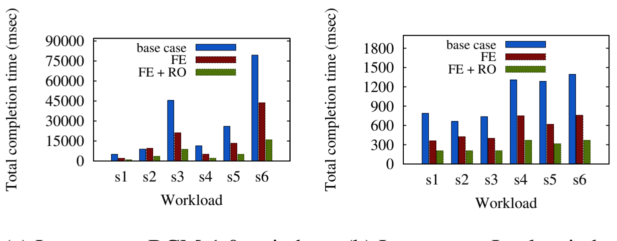


Figure 19: Worst case flow setup time of affected flows in the failover scenario with BCM-1.0 switches



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

Figure 20: Worst case flow set up time in the two level traffic engineering scenario

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, Mazu provides operators additional flexibility in designing schemes to better meet failover requirements in their networks.

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).

7.2.2 *Two-level Responsive Traffic Engineering*

Next, we evaluate the effectiveness of Mazu's FE and RO techniques 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.

We use the same topology and workloads described in §7.2.1. 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 20a and 20b 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*).

7.2.3 *MicroTE*

MicroTE [5] leverages the partial and short term predictability of a data center traffic matrix to perform traffic engineering at small time-scales. As noted in §6.1, 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 [3] 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 21 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).

In summary, all of Mazu's mechanisms are crucial to ensuring that route setup is sufficiently fast.

8. 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 perfor-

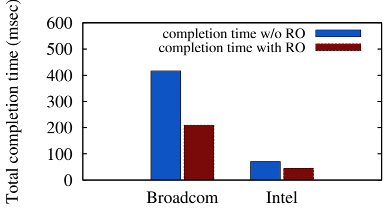


Figure 21: Flow setup time of MicroTE with and without RO in a Fat Tree (k=8) topology

mance issues, or do not explore implications on applications that require tight control of latency. Devoflow [7] 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 [11,23] provided a more in-depth look into switch performance across various vendors. In [23], 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 [11], 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 rule priority structures. 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 [7] 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. Mazu's techniques are largely complementary to this. DIFANE [24] 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 [18] 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 [25] optimally schedules a set of rule updates while maintaining desirable consistency properties (e.g., no loops and no blackholes). Dionysus assumes the target network state is given, while Mazu computes a target network state that spreads rule updates among more switches in order to increase parallelism. Thus, we believe that Dionysus and Mazu are complementary.

9. CONCLUSION

Critical SDN applications such as fast failover and mobility demand tight interaction between switch control and data planes. However, our measurements across three 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). To mitigate the challenges these latencies create for SDN in supporting critical management applications, we present a system called Mazu. Mazu bypasses switch processing of *packet_in* messages by redirecting unmatched packets to a proxy. To reduce the latency of *flow_mod* operations, Mazu employs: (1) *flow engineering*, a mechanism to allow management applications to take path setup latency as a second objective, and, (2) *rule offloading*, which computes strategies for opportunistically offloading portions of forwarding state to be installed at a switch to other switches downstream from it. Our evaluation shows that these mechanisms can tame flow setup latencies, thereby enabling SDN-based control of critical applications.

10. REFERENCES

[1] Hp network protector sdn application. `http://bit.ly/SAWTj1`.
[2] Private communication with Cristian Estan.
[3] Al-Fares et al. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.
[4] Al-Fares et al. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.
[5] Benson et al. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *CoNEXT*, 2011.
[6] Casado et al. Rethinking Enterprise Network Control. *IEEE/ACM Trans. Netw.*, 2009.
[7] Curtis et al. DevoFlow: Scaling Flow Management for High-performance Networks. In *SIGCOMM*, 2011.
[8] Heller et al. ElasticTree: Saving Energy in Data Center Networks. In *NSDI*, 2010.
[9] Hong et al. Achieving High Utilization with Software-driven WAN. In *SIGCOMM*, 2013.
[10] Huang et al. A Close Examination of Performance and Power Characteristics of 4G LTE Networks. In *MobiSys*, 2012.
[11] Huang et al. High-fidelity Switch Models for Software-defined Network Emulation. In *HotSDN*, 2013.
[12] Jain et al. B4: Experience with A Globally-deployed Software Defined WAN. In *SIGCOMM*, 2013.
[13] Jin et al. SoftCell: Scalable and Flexible Cellular Core Network Architecture. In *CoNEXT*, 2013.
[14] Koponen et al. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.
[15] Liu et al. zUpdate: Updating Data Center Networks with Zero Loss. In *SIGCOMM*, 2013.
[16] Mahajan et al. On Consistent Updates in Software Defined Networks. In *HotNets*, 2013.
[17] McKeown et al. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 2008.
[18] Moshref et al. Scalable Rule Management for Data Centers. In *NSDI*, 2012.
[19] Olsson et al. Pktgen the Linux Packet Generator. In *Proceedings of the Linux Symposium, Ottawa, Canada*, 2005.
[20] Patel et al. Ananta: Cloud Scale Load Balancing. In *SIGCOMM*, 2013.
[21] Qazi et al. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *SIGCOMM*, 2013.
[22] Reitblatt et al. Abstractions for network update. In *SIGCOMM*, 2012.

[23] Rotsos et al. Oflops: An Open Framework for Openflow Switch Evaluation. In *PAM*, 2012.

[24] Yu et al. Scalable Flow-based Networking with DIFANE. In *SIGCOMM*, 2010.

[25] Xin Jin, Hongqiang Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Jennifer Rexford, Roger Wattenhofer, and Ming Zhang. Dionysus: Dynamic scheduling of network updates. In *SIGCOMM*, 2014.