Mazu: Taming Latency in Software Defined Networks

Keqiang He[1], Junaid Khalid[1], Sourav Das[1], Aditya Akella[1], Li Erran Li[2], and Marina Thottan[2]

[1]University of Wisconsin - Madison

[2]Bell Labs, Alcatel-Lucent

ABSTRACT

Timely interaction between an SDN controller and switches is crucial to many SDN management applications such as fast rerouting during link or switch failure and reactive path setup for latency-sensitive flows. However, our measurement study using two vendor platforms 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. To overcome the latencies and achieve responsive control, we develop Mazu, a systematic framework leveraging both the logically central view and global control in SDN, and the dissection of latencies from our measurement study. Mazu avoids the switch CPU processing tasks due to data plane packet arrivals by redirecting the packets to a fast proxy that is tasked with generating messages for the controller. Mazu presents novel controller algorithms to spread the rule updates across multiple switches, optimally ordering rules during insertion. With reduced number of rules to update per switch, and hardware-friendly ordering, rule update tasks finish much faster. Controlled simulations and testbed experiments show that our techniques can reduce the latency to update network state by almost 5X. Thus, Mazu makes SDN-based control suitably responsive for critical management applications.

1. INTRODUCTION

Software defined networking (SDN) advocates separating control and data planes in network devices, and provides a logically centralized platform to program data plane state [4, 15]. This has opened the door to rich network control applications that can adapt to changes in the underlying network or its traffic patterns more flexibly and at smaller time scales than legacy control planes [6, 7, 10, 11, 18, 19]. Therefore, it is not surprising that SDN is being rapidly deployed or under developed in many domains, including data center (DC) [3, 6, 13], cloud [18], inter-DC WAN [7, 10] and, cellular [11] networks.

However, a number of important management applications in these domains, such as fast fail-over, reactive routing of latency-sensitive flows, and fine-grained DC traffic engineering [3], are stretching SDN's capabilities. These applications require the ability to reprogram data plane state at very fine time-scales to optimally meet their objectives. For instance, fine-grained DC traffic engineering approaches require routes to be set up within a few hundred milliseconds to leverage short-term traffic predictability [3]. Setting up routes in cellular networks (when a device becomes active, or a during handoff) must complete within $\sim$30-40ms to ensure users can interact with Web services in a timely fashion. It is important that SDN support such applications, otherwise operators will be forced to adopt expensive custom solutions alongside SDN (e.g., bandwidth reservation, custom hardware/protocols etc.), which can undermine SDN's "CapEx" and "OpEx" benefits.

For such applications, timely interaction between the logically central SDN control plane and network switches is crucial. Timeliness is determined by: (i) the speed of control programs, and latency to/from the logically central controller, and (ii) the responsiveness of network switches in interacting with the controller, specifically, in generating the necessary input messages for control programs, and then modifying forwarding state as dictated by them. Robust control software design and advances in distributed controllers [12] have helped overcome the first issue. However, with most of the focus today being on the flexibility benefits of SDN relative to legacy technology, the latter issue has not gained much attention from vendors and researches alike.

Alarmingly, preliminary studies [9, 20] and anecdotal evidence suggest that latencies underlying switch actions in (ii) could be significant. However, it is not clear what factors impact these latencies, what the underlying causes are, and whether the causes are fundamental to switch designs. As a result, whether it is possible to overcome these latencies at all is not known. Thus, SDN's ability to provide sufficiently responsive control to support the aforementioned applications remains in question.

To this end, we make two contributions. In this first part, we present a thorough systematic exploration of these latencies in production SDN switches from 2 different vendors—Vendor A and Vendor B—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 rule on average on Vendor B). We find the delay is particularly high whenever the switch CPU 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 Vendor A). The latency crucially depends on the priority patterns both in rules being inserted as well those already in a switch's table. We find that there are significant differences in latency trends across the two switches, pointing to different internal optimizations.

We find that poor switch software design contributes significantly to the observed high latencies. However, pathological interactions between some rule priority input sequences and switch hardware table heuristics for rule layout play a significant role in inflating latency as well. Crucially, the latter issue is fundamental and may be hard to overcome.

Informed by our measurements, we design a framework called Mazu that *leverages* the centralized view and global control in SDN to overcome or mitigate the impact of latencies, including both latencies from implementation-related causes and those from fundamental ones. The first technique in Mazu is to avoid the switch CPU processing events due to data plane packet arrivals by redirecting packets to a fast proxy that is tasked with generating the necessary messages for the controller. By leveraging a fast commodity processor and decoupling inbound and outbound actions, this eliminates inbound latency.

Because outbound latency is intrinsically linked to switch software and hardware, we develop techniques that can be used (in isolation or together) for systematically *mitigating* it. Our first technique, *flow engineering*, leverages our empirical latency models to compute paths such that the latency of installing forwarding state at any switch is minimized. Our second technique, *rule offloading*, computes strategies for opportunistically offloading portions of forwarding state to be installed at a switch to other switches downstream from it. By virtue of reducing installation latency per switch and enabling parallel execution of updates, these two techniques ensure rule update tasks to finish much faster. Finally, we provide rules for installation at a switch in an order that is optimal for the vendor in question.

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

Our work focuses mainly on switches supporting the popular OpenFlow [15] protocol, specifically OpenFlow 1.0, which is widely available in production switches today. As we will be clear, our results apply qualitatively to more recent versions of OpenFlow e.g., OpenFlow 1.3., as well.

2.1 Basics

An OpenFlow SDN switch by default does not run control plane protocols, as these are delegated to an external application running on a logically central controller. Applications determine the routes traffic must take through the network and instruct the controller to update the switching substrate with the appropriate forwarding state. OpenFlow is the API employed by switches to communicate with the controller to enable such state update operations.

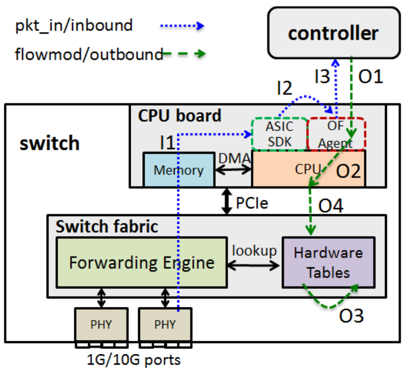


Figure 1: Schematic of an OpenFlow switch. Different rectangular blocks indicate different components of a switch; different boxes do not mean different chips. We also show the factors contributing to inbound and outbound latency

***packet_in* processing:** When a packet arrives, the switch ASIC first looks it up against the switch's hardware forwarding tables. If a match is found, the packet is forwarded at line rate. Otherwise the following steps take place (Figure 1): (I1) 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 spanning the first 128B including header. All of three steps, I1–I3, can impact the latency in generating *packet_in* which we call *inbound latency*.

The application residing on the controller processes the message and upon determining routes for packets belonging to the corresponding flow, sends a *flow_mod* and a *packet_out* message. The *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 typical "simple" rule is *srcip=A,dstip=B,action=output:3* (the rule can cover a total of 12 matching fields in Openflow 1.0); wild-cards can be used. 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. The *packet_out* message simply releases the packets buffered at the switch to be forwarded along.

***flow_mod* processing:** When it receives a *flow_mod* the switch takes the following steps: (O1) The switch software running on the CPU parses the OpenFlow message. (O2) The software 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 on the switch to be moved around, e.g., to accommodate high priority rules. (O4) The hardware table is updated with the rule. All of four steps O1–O4 impact the total delay in fully executing *flow_mod* action, which we call *outbound delay*.

SDN applications can be of two forms: *reactive* or *proactive*. The former work by enforcing default-off forwarding to flow sub-spaces; this causes any flow in that sub-space to generate a *packet_in* event when it reaches an ingress switch for the first time. The application then determines the forwarding action and sends the corresponding *flow_mod* messages down to the switch. These applications are impacted by both in- and outbound delays. Proactive applications directly update network forwarding state using *flow_mod* messages. They are mainly impacted by outbound delays.

2.2 Motivating Applications

In what follows, we provide examples of management applications that require fine-grained control over data plane state. We highlight the impact of inbound and outbound latencies on the applications' objectives.

Mobility: Recent work [11] advocates using SDN to simplify path setup and management in cellular networks. In these settings, paths need to be set up whenever devices want to access the Internet. In addition, these paths need to be reconfigured during handoff. Currently these paths are implemented as GPRS Tunneling Protocol (GTP) tunnels in LTE. To avoid impacting the performance of subscribers, these tunnels must be setup within a small latency bound. For example, when a mobile device in idle state wants to communicate with an Internet server it needs first to transition from idle state to connected state. According to recent measurement studies [8], this transition delay is around 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.

Failover: 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 [7]. However, this requires significant updates to network state at multiple network switches. The longer this update takes, 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.

Switch	CPU	RAM	Flow Table Size	Data Plane
Vendor A	1Ghz	1GB	896	14*10Gbps + 4*40Gbps
Vendor B	2Ghz	2GB	4096	40*10Gbps + 4*40Gbps

Table 1: Specific details of the switches

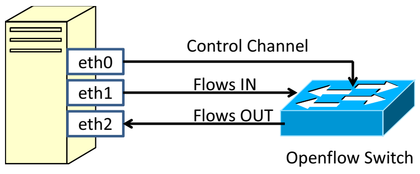


Figure 2: Measurement experiment setup.

Intra-DC Traffic Engineering: Some recent proposals, such as MicroTE [3] and Hedera [2] 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 a 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).

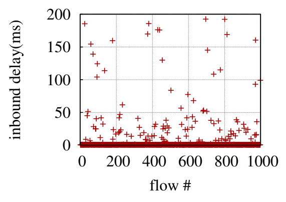
3. LATENCY MEASUREMENTS

An important first step to taming in/outbound latencies is to understand the various factors that affect them within the SDN switch. We conduct a variety of measurements aimed at carefully isolating these factors. To draw general observation, we use 2 commercial switch platforms (Table 1). To ensure that we are experimenting in the optimal regimes for the different switches we take into account switch specifics such as maximum flow table sizes as well as support for priority in rule set up.

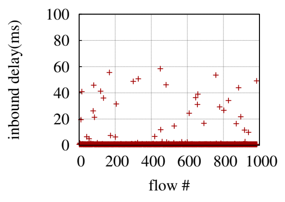
3.1 Measurement Methodology

Our empirical setup is shown in Figure 2. The PC has one 1Gbps and two 10Gbps ports that are connected to the switch under test. The eth0 port is connected to the control port of the switch on one side and a POX SDN controller running on the PC is set to listen on this port. The ports eth1 and eth2 are connected to the data ports on the switch. The propagation delay between the switch and the controller is negligible (about 0.1 ms). The controller is used to send a set of Openflow 1.0 *flow_mod* commands to the switch in burst mode. To generate traffic for the 10Gbps NIC on the data plane, we use pktgen [17] in kernel space. Using this generator we are able to generate traffic at 600-1000 Mbps.

Prior work notes that accurate execution of open flow commands on commercial switches can only be accurately observed in the data plane [20]. Thus, our experiments are crafted toward ensuring that the impact of various factors on



(a) with flow_mod/pkt_out



(b) w/o flow_mod/pkt_out

Figure 3: Inbound delay on Vendor B. 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 Vendor B switch

the latencies can be measured directly from the data plane (at eth2 in figure), except for *packet_in* part of inbound latency. We use *libpcap* running on a high performance host to accurately time stamp the different packet and rule processing events of each flow. We first log the timestamps in memory and when the experimental run is completed, the results are dumped to the disk and processed. We use the time stamp of the first packet associated with a particular flow as the finish time of the corresponding *flow_mod* command. Further details that depend on the specific issues we measure are presented in later sections.

3.2 Dissecting Inbound Delay

To capture inbound delay, we empty the table at the switch. We generated 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. In the first experiment, the *packet_in* will trigger corresponding *flow_mod* and *packet_out* messages; the *flow_mod* messages insert simple OpenFlow rules (differing just in destination IP). In the second experiment, the *packet_in* message is dropped silently by the controller.

We record the timestamp (t_1) when each packet is transmitted on the measurement server's NIC. We also record the timestamp (t_2) when the server receives *packet_in* message. The difference $t_2 - t_1$ is the inbound delay.[1]

Representative results for these two experiments are shown in Figures 3(a) and (b), respectively, for the Vendor B switch; results for the Vendor A switch are qualitatively similar. For the first experiment (a), we see that the inbound delay is quite variable with a mean of 8.33 ms and standard deviation of 31.34; also, it increases with the *packet_in* rates (e.g., the mean is 3.32 ms for 100/s; not shown). For the second experiment (b) the inbound delay is significantly small for most of the time. The only difference across the two experiments is that in the former case, the switch CPU is processing *flow_mod* and *packet_out* alongside generating *packet_in* messages. As such, we see significant CPU utilization during this experiment (Table 2). Thus, we conclude that inbound delay is mainly caused by switch CPU and due to interference with *flow_mod* and *packet_out* processing.

[1]This measurement technique differs from the approach used in [9], where the delay was captured from the switch to the POX controller which includes the overhead at the controller.

3.3 Dissecting Outbound Delay

Before we perform the outbound delay measurements, first we install a single default low priority rule which instructs the switch to drop all the traffic. Then we install specially designed Openflow rules at the switch; while they simply specify the destination IP address leaving other fields wildcarded, they may have different priorities. All instruct the switch to output traffic to the port which is connected to the measurement host on which we are monitoring.

We examine outbound latencies for three different *flow_mod* operations in turn, namely, insertion, modification and deletion. We examine the impact of key factors on these latencies, namely, table occupancy and rule priority structure.

3.3.1 Insertion Latency

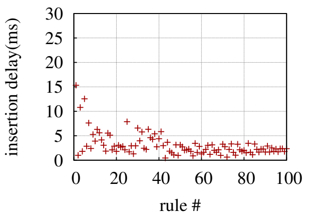
We conduct a variety of tests to examine how different patterns of rule workloads impact insertion latency. In almost all experiments, we install a burst of rules. Let us denote these rules in a sequence as $r_1, r_2, \cdots, r_i, \cdots, r_B$. Denote $T(r_i)$ as the time we observe the first packet matching r_i emerging from the intended port of the rule action. We define insertion latency as $T(r_i) - T(r_i - 1)$.

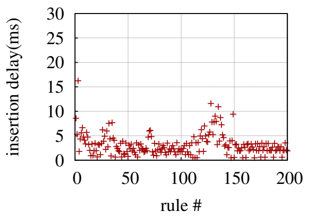
Table occupancy: To understand the impact of table occupancy, we insert a burst of B rules back to back into a switch that already has S rules in it. All $B + S$ rules have the same priority. We fix B and experiment with different S on both Vendor A and Vendor B switches. We ensure $B + S$ rules can be accommodated in the switch's hardware table.

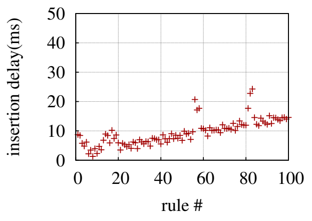
Taking $B = 400$ as an example, we found that the flow table occupancy *does not* impact the insertion operation if all rules have the same priority. The mean insertion delay is 3.14, 1.11 ms and the standard deviation is 2.14, 0.18 for Vendor A and Vendor B respectively, irrespective of S.

Rule priority: To understand rule priority effects 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. 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.

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


(a) burst size 100, same priority


(b) burst size 200, same priority


(c) burst size 100, incr. priority

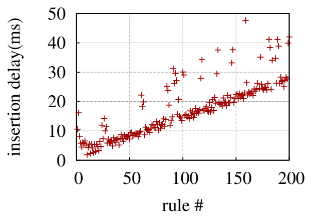
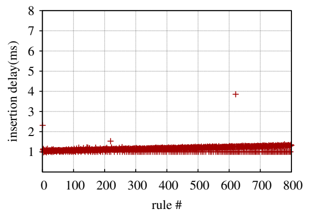

(d) burst size 200, incr. priority

Figure 4: **Vendor A** priority per-rule **insert** latency results


(a) burst size 800, incr. priority

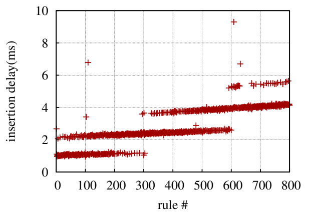

(b) burst size 800, decr. priority

Figure 5: **Vendor B** priority per-rule **insert** latency results

Vendor A: increasing priority. Figure 4(c) shows the result for $B = 100$. We note that the per rule insertion delay actually *increases linearly* with the number of rules inserted. Figure 4(d) 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 can be much larger 9.47 (17.66) ms vs 3.12 (3.02) ms, for $B = 100$ (200). Results for other B's 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!

Vendor A: decreasing priority. We also performed decreasing priority insertion (not shown in figure). We observe that the burst of n rules is divided into a number of groups, and each group is reordered and inserted in TCAM in order of increasing priority. This indicates that Vendor A firmware reorders the rules and prefers increasing priority insertion.

Vendor B: same priority. For $B = 800$ on Vendor B we see that the per rule insertion delay is similar across the 800 rules, with a median of 1.17 ms and standard deviation of 0.16 (not shown). The results for other Bs are similar. Thus, similar to Vendor A, same priority rule insertion delay does not vary with burst size on Vendor B.

Vendor B: increasing priority. Figure 5(a) shows per-rule latencies for $B = 800$. *Surprisingly*, in contrast with Vendor A, the per rule insertion delay among the rules is more or less the same, with a median of 1.18 ms and a standard deviation of 1.08. We see similar results for other Bs. This shows that the Vendor B TCAM architecture is fundamentally different from Vendor A. Rules are ordered in Vendor B's TCAM in a way that higher priority rule insertion does not displace existing low priority rules.

Vendor B: decreasing priority. Figure 5(c) shows per-rule latencies for $B = 800$. We see two effects: (1) the latencies alternate between two modes at any given time, and (2) a step-function affect after every 300 or so rules.

A likely explanation for #1 is bus buffering. Since it is part of the control path of the switch it is not really optimized for latency. The latter can be explained as follows: Examining Vendor B switch architecture, we found that it has 24 slices, say $A_1 \ldots A_{24}$, 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. Subsequently, when the next low priority rule is inserted in our experiment, 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 with Vendor B.

Summary and root causes: We observe that (1) same priority insertions are fast and are not affected by flow table occupancy. This is true for both Vendor A and Vendor B; (2) priority insertion patterns can affect insertion delay very differently. For Vendor B, increasing priority insertion is similar to same priority insertion. However, insertion in decreasing priority can incur much higher delay. For Vendor A, insertions with different priority patterns are all much higher than insertions with same priority.

Key root cause for observed latencies are how rules are organized in the TCAM, and the number of slices.

Even in the faster of the two switches, Vendor B, per rule insertion latency of 1ms is higher than what the state-of-the-art TCAMs can support in terms of update rates. Thus, there appears to be an intrinsic switch software overhead contributing to all latencies as well.

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 understand the impact of table occupancy, we pre-insert S rules into a switch (simple rules as before), 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 Vendor A at $S = 100$ and $S = 200$ are shown in Figure 6(a) and (b) respectively. We

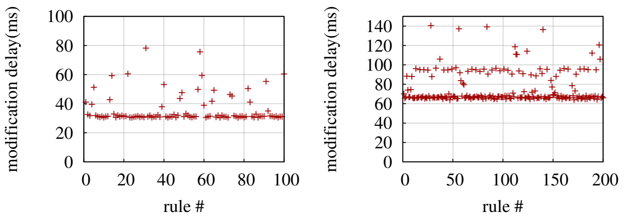

(a) 100 rules in table (b) 200 rules in table

Figure 6: **Vendor A** per-rule **mod.** latency: same priority

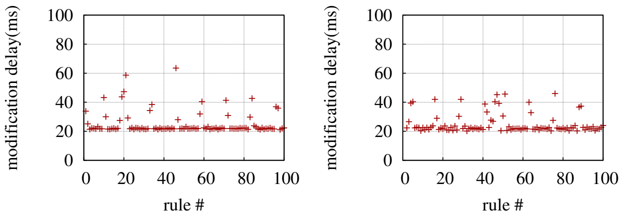

(a) burst size 100, incr. priority (b) burst size 100, decr. priority

Figure 7: **Vendor A** priority per-rule **modification** latency results

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 (3.12 ms per rule) (§3.3.1). For Vendor B, the modification delay for $S = 100, 200$ is around 1 ms (standard deviation 0.06) for all modified rules, similar to insertion delay with same priority, in contrast with Vendor A.

Rule Priority: We conducted two experiments. In each, we insert B rules into an empty table. 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. Likewise, we define the *decreasing priority* experiment. We vary B.

Vendor A: increasing/decreasing priority. Figure 7(a) and (b) show the results for the increasing and decreasing priority experiments, respectively, for $B = 100$. In both cases, we see: (i) the per rule modification delay is similar across the rules, with a median of 25.10 ms and a standard deviation of 6.74, (ii) the latencies are identical across the experiments. We observed that the latencies grew with B for both experiments.

Taken together with the table occupancy results above, we conclude that the per rule latency for modification on Vendor A is impacted purely by table occupancy, not by rule priority structure. For Vendor B, the modification delay is 1 ms independent of rule priority, table occupancy or B.

Summary and root causes: We observe that flow rule priority do not impact modification delay. Modification delay in Vendor A is a function of table occupancy, whereas this is not the case for Vendor B where modification is as fast as insertion. Conversations with Vendor A indicated that TCAM modification should ideally be fast, so the underlying cause appears to be poorly optimized switch software.

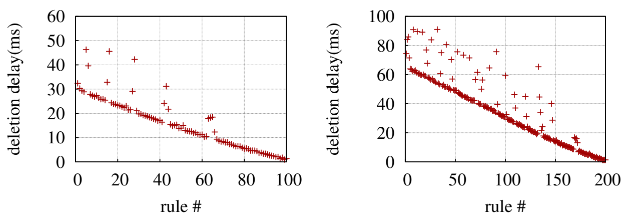

(a) 100 rules in table (b) 200 rules in table

Figure 8: **Vendor A** per-rule **del.** latency: same priority

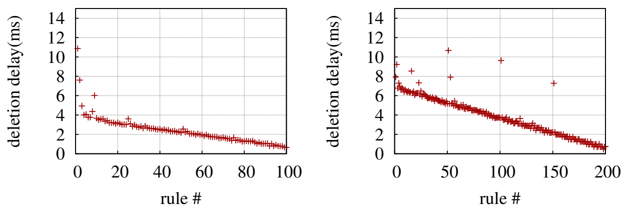

(a) 100 rules in table (b) 200 rules in table

Figure 9: **Vendor B** per-rule **del.** latency: same priority

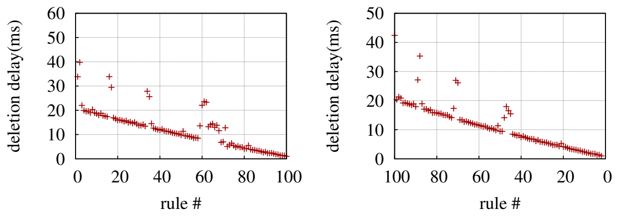

(a) increasing priority (b) decreasing priority

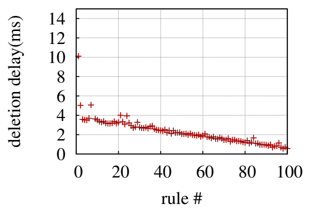
Figure 10: **Vendor A** priority per-rule **deletion** latency results, burst size 100: (a) incr. priority (b) decr. priority

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 Vendor A at $S = 100$ and $S = 200$ are shown in Figure 8(a) and (b), respectively. We see that per rule deletion delay decreases as the table occupancy drops. We see a similar trend for Vendor B (Figure 9(a) and (b)).

Rule Priorities: From B existing rules in the switch, we delete one rule at a time, "with" and "without priority". In case of priority, we delete rules in increasing and decreasing priority order. As shown in Figures 10(a), (b) for Vendor A, and Figures 11 (a) and (b) for Vendor B, deletion is not af-


(a) increasing priority

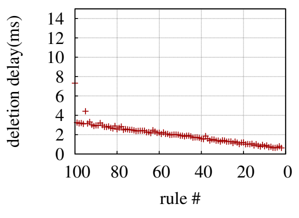
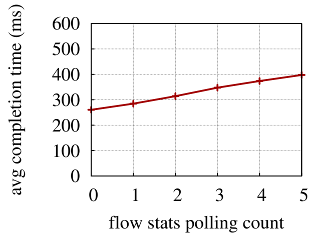

(b) decreasing priority

Figure 11: **Vendor B** priority per-rule **deletion** latency results, burst size 100: (a) increasing priority (b) decreasing priority


(a) init. table occupancy 0

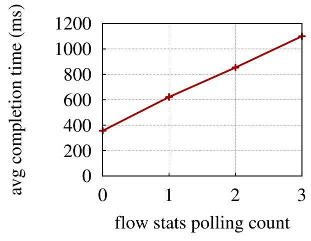

(b) init. table occupancy 500

Figure 12: Polling effects on completion time on Vendor A switch. Burst size 100. Measured using simple openflow rules (i.e., just vary destination IP).

fected by rule priorities in the table and the order of deletion for either platform. However, deletion delay is affected by the number of rules in the table.

Root cause: Since deletion delay decreases with rule number in all cases, we conclude that deletion is incurring TCAM reordering. We also observe that rule time out processing does not impact *flow_mod* perations much. Given these two observations, our recommendation is to let rules time out rather than explicitly delete them, if possible.

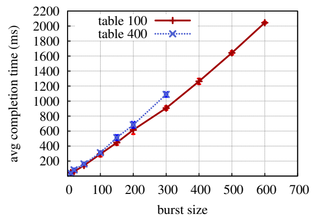
3.3.4 *Impact of concurrent switch CPU jobs*

To investigate the impact of concurrent switch CPU activities, we instruct the switch to perform flow statistics queries. Figure 12 shows that concurrent activities such as polling statistics can have a great impact on insertion delay, especially at high table occupancy. E.g., the total completion time of inserting a burst of 100 rules with same priority into a table with 500 rules can take around 853 ms when there are two polling events during the insertion process. In contrast, it takes 356 ms when there is no polling event.

3.4 Overall burst insertion completion time

With the understanding of per-rule insertion latency, we present burst rule completion time as this is the metric many applications such as failover depend on.

We conduct two experiments. With S rules in the table, we insert a burst of B rules. For the first experiment, S has high priority and we insert the burst with low priority. For the second experiment, if it is Vendor A, S has low priority


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

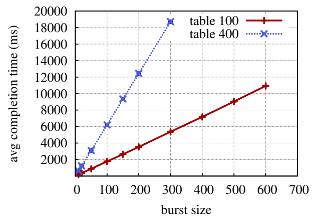

(b) high priority rules into a table with low priority rules

Figure 13: Overall completion time on Vendor A switch. Two priority effects. Initial flow table occupancy is S high (low) priority rules. Then, insert a burst of low (high) priority rules. Averaged on 5 rounds.

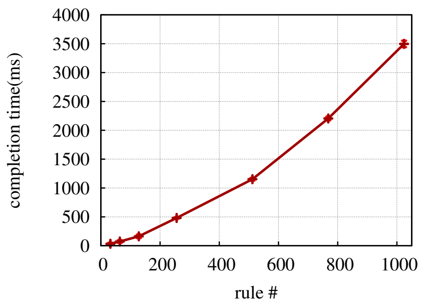

(a) decreasing priority

Figure 14: **Vendor B** overall completion time

and we insert rules with high priority; if it is Vendor B, S has high priority and we insert rules in *decreasing* priority.

For Vendor A, based on our hypothesis, as long as the same number of rules get displaced, the completion time should be the same. Indeed, from Figure 13(a), 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 when there is only 100 high priority rules in the table. In Figure 13(b), since newly inserted high priority rules will displace 400 low priority rules in the table, the completion time will be about three times higher than $S = 100$.

For Vendor B, we also run the same two experiments as for Vendor A. The results are similar to rule insertion with same priority. This indicates that Vendor B optimizes for rule priority better than Vendor A. When we insert in decreasing priority, as shown in Figure 14, the completion time is about 3.5 seconds, three times higher than the case of insertion with same priority.

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 15).

The Mazu controller implements a number of modules.

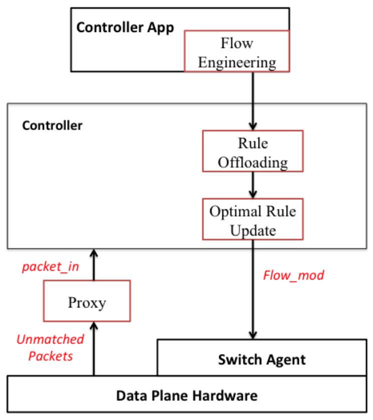


Figure 15: Mazu framework

The first, i.e., proxy, handles inbound processing. We show that the proxy can in fact eliminate all inbound delays (§5).

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 compute routes that spreads 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.

Optimal rule update 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

To overcome the inbound latency entirely a simple idea we employ is to *physically decouple the switch's handling of packet_in* and *packet_out messages from* flow_mod *messages*. We punt all *packet_in* message generation (and *packet_out* processing) to a separate optimized processing unit, i.e., a custom proxy, co-located with ingress switches in a network; multiple ingress switches can share a proxy too.

We establish a (short) label-switched path between the switch and its corresponding proxy. The switch continues to have a control channel to the controller (an SSL connection); in addition, we establish a control channel (SSL) between the proxy and the controller. The controller must associate each switch with its relevant proxy.

To exercise the proxy, we insert a default low priority rule in the switch; this redirects at line rate all unmatched packets on the label-switched path to the proxy. The switch stamps the incoming port ID in the IPID field on the packet before label-switching it to the proxy. The proxy generates the necessary *packet_in* messages reflecting the switch's incoming port ID, and forwards them on its control channel to the controller, and buffers *packet_in* locally (similar to a regular switch). The controller sends *packet_out* messages to the proxy; the proxy processes the message and forwards the buffered packet corresponding to the *packet_in* back to the switch for routing to the eventual destination. *flow_mod* messages are sent directly to the switch.

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 never delete rules*. Instead, we simply let them time out (and insert higher priority rules to supersede them as needed).

2. *Avoid modifying rules for Vendor A*: Our measurements with the Vendor A switch showed that modifying a rule can be much 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 Vendor A). For the Vendor B 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, e.g., 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 [7, 16, 19]. Unfortunately, these techniques do not provide sufficient control over outbound delay.

Minimize maximum flow table occupancy is very sub-optimal: For example, consider a simple setting where there are three candidate paths between a pair of nodes as shown in Figure 16. Each path has one Vendor A 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

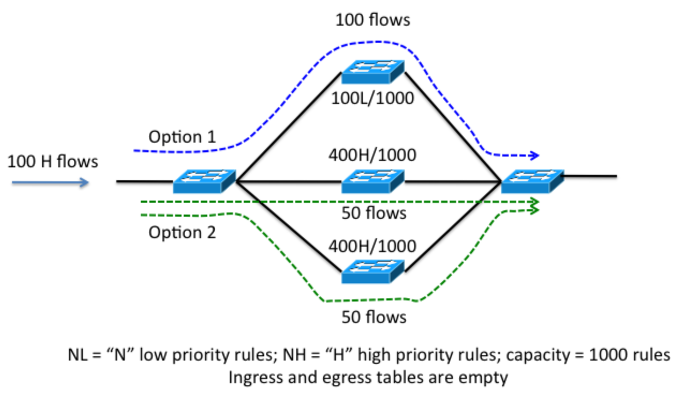

100 flows
100L/1000
Option 1
100 H flows
400H/1000
50 flows
Option 2
400H/1000
50 flows
NL = "N" low priority rules; NH = "H" high priority rules; capacity = 1000 rules
Ingress and egress tables are empty

Figure 16: Illustration of flow engineering assuming a Vendor A switch

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 Vendor A 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 13, it is about 200 ms vs 2 seconds, *a 10X difference!* Vendor B 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. Our heuristic works by exploring for each source-destination pair whether a path can accommodate both its demand, and the path setup latency is within some bound. If either is violated, we try out 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 of magnitude 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; supposed $P^{1..K}_{(u,v)}$ is 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:

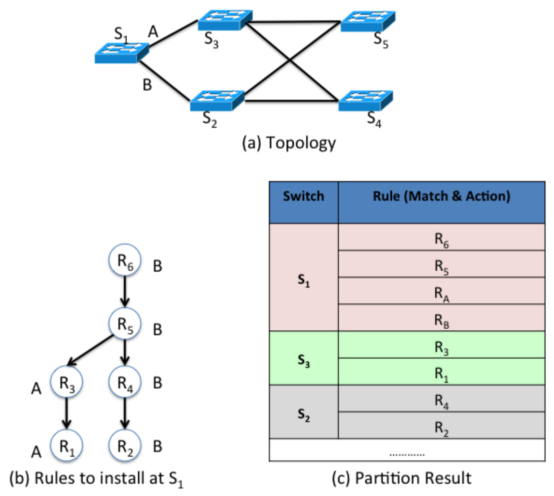
Per-switch latency: Given our measurement results, for every switch $s \in P^1_{(u,v)}$, we can model the latency at s due to routing (u, v) as $L_s = max(a, (b + c * Disp_s(Pri(u, v))))$. Here, $Disp_s(Pri(u, v))$ is the number of rules at s that will be displaced by the rule for (u, v). For the Vendor A switch, this is the number of rules of priority lower priority than $Pri(u, v)$, whereas for the Vendor B 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 Vendor B. For Vendor A, 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 im-



Switch	Rule (Match & Action)
S1	R6
	R5
	RA
	RB
S3	R3
	R1
S2	R4
	R2
	....

Figure 17: Illustration of rule offloading

plemented 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 [3] (§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 [7, 10]. In such networks, fine-grained SDN applications performing traffic engineering or reactive routing typically control tunnel end-points, setting up 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.*

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. In other words, packets matching 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 [16]'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 [16]. In contrast, we offload by grouping the next hop of rule actions to increase offloading opportunities.

We illustrate this using an example in Figure 17. Figure 17(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 17(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 $R \to R'$). 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 $R_3 \to R_1$ 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, $R_5 \to R_3$, and $R_5 \to R_4$; thus, R_5 has to be in the root partition ("pinned" to the ingress switch S_1). Also all rules R' such that $R' \to R_5$ 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 17(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. 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 parallely installing the offload rules.

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 that direct traffic from upstream switches for eventual processing at downstream switches.

```
//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 18: Recursive Rule Partition Algorithm

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 challange is dealing with the fact that the simplistic default rules described above may have a non-empty intersection, where the intersection has 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.

We assume that each rule can be represented by a rectangle (src, 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 Ordered Rule Insertion

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

We assume consistent update mechanisms [13,14] are used for network path update. In consistent update, new rules will not take effect unless all of them are installed. Therefore, Mazu can optimize the ordering without causing temporal policy violations.

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 CPU at 2.66Ghz and 8GB RAM). We evaluated it using the same setup as described in §3.2. We found that the proxy almost completely eliminates the inbound delay: the delay is under 0.199ms (0.146ms) for a flow arrival rate of 200/s (2000/s); the 99th percentile delay is as small as 0.476ms (3.56ms), which is mainly due to the proxy's software overhead. Since the proxy is physically decoupled from the switch, there is no impact of the switch's *flow_mod* or *packet_out* operations on the proxy's *packet_in* operations. In contrast, without the proxy, the mean and 99th percentile delays are 8ms and 192ms respectively, with a flow arrival rate of 200/s. These improvements are significant, especially for latency sensitive applications such as VoIP calls in cellular networks.

Workload	Popularity Index	Popularity Index for high priority traffic	No of low priority rules in the flowtable
s1	1-10	1-5	0-50
s2	1-10	1-5	100-200
s3	1-10	1-5	300-500
s4	1-20	1-7	0-50
s5	1-20	1-7	100-200
s6	1-20	1-7	300-500

Table 3: Workloads used in simulation

7.2 Outbound Latency

In what follows, we use a variety of large-scale simulations on various topologies with different workloads to study the impact of delays imposed by outbound latencies and the improvements offered by Mazu. We leverage the switch latency models derived from our measurements in §3.3.

7.2.1 Flow Engineering

To evaluate the effectiveness of Mazu's flow engineering technique, we simulate a failover scenario in a tunneled WAN network, where a random link experiences a failure.

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 $[100, 1000]$.

Traffic matrix: We assign a popularity index (random number) 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 matrix is routed in a way 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 rules in them, a subset of which are displaced by the new rules. We randomly pick the number of such displaced rules within some interval (explained in more detail below). For simplicity, we assume that there are no dependencies across rules; we consider dependencies in subsequent sections.

Workloads: We consider six workloads ("s1", "s2", "s3", "s4", "s5", and "s6") shown in Table 3. In low traffic workloads, s1, s2 and s3, the number of rules that can be displaced at any switch is in $[0, 50]$, $[100, 200]$, $[300, 500]$ respectively; and the number of flows between any pair of nodes is on average 50 with a maximum of 100. In high traffic workloads, s4, s5 and s6, the number of rules that can be displaced at any switch is in $[0, 50]$, $[100, 200]$, $[300, 500]$ respectively; and the number of flows between any pair of nodes is on average 200 with a maximum of 400.

To simulate failures we randomly select a tunnel in the mesh and fail it. We assume that there is enough spare capacity in the network to route the affected traffic.

We study three mechanisms: (a) Base case, which reroutes the affected flows while minimizing the maximum link load, ignoring setup latencies.(b) Flow engineering (FE) to select paths for affected flows such that flow installation latency is minimized (§6.1). (c) Flow Engineering and Rule Offload (FE + RO), which implements rule offloading (§6.2) in addition to FE. It 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.*

On a link failure, around 70 flows get rerouted for low traffic workloads and 220 for high traffic workloads. All the rerouted flows are treated as new flows.

We simulate both with Vendor A and Vendor B, assuming all switches in the network are from the same vendor.

Figure 19 shows the latencies with Vendor A 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 rest of the 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, (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 Vendor B switch model. Recall that all rules we insert have the same priority. Since the Vendor B 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 in base case, vs 129ms with Mazu).

7.2.2 Rule Offload in Depth

ClassBench: While the above scenarios did leverage rule offload, the rules inserted had a flat priority structure and did not have dependencies. In what follows, we study how well rule offload works when these simplifying criteria may not apply. We leverage ClassBench [21] to generate a variety of rule sets representative of real world access control. Since dependent rule sets in most cases hinder rule offloading to the next hops switches we believe that the ACL rule sets generated from ClassBench–which have a significant amount of dependencies–will present a near worst case scenario for our rule offload scheme.

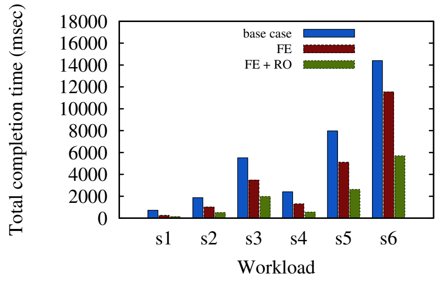


Figure 19: Worst case flow setup time of affected flows in failover scenario with base case, FE and FE+RO techniques in a full mesh (25 nodes) topology with Vendor A switches

We consider the following simple setup: we use a three-level FatTree topology [1] with degree 8, containing 128 servers connected by 32 edge switches, 32 aggregate switches and 16 core switches. We use ClassBench to generate 90 rules for each edge switch. Each rule is assigned a different priority and a tunnel tag to indicate its tunnel or path.

Without rule offloading, assuming one-shot updates, the installation time will be the maximum at any edge switch when inserting 90 rules. Under rule offload, we assume that a core switch cannot accommodate more than 60 rules in total from *all* of its immediate upstream neighbors.

We experimented with a variety of rule sets generated by ClassBench. On average, we found that the speedup from using rule offload relative to not using it is 2.1 for Vendor A switches and 1.4 for Vendor B switches. This speedup is made possible by rule offload "spreading" out rules to downstream switches, enabling parallel execution of updates.

MicroTE: We now consider an important scenario discussion in §2, namely fine-grained intra-DC traffic engineering using MicroTE [3]. MicroTE leverages the partial and short term predictability of the traffic matrix in a datacenter 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 rule offload, the extent of which we study next.

We use the same data center topology as discussed earlier in this section. We assume that the traffic rate between a pair of servers is derived from a Zipfian distribution. Figure 20 shows the rule installation completion time. We see that RO provides a 2X improvement (400ms to 200ms) assuming the Vendor A switch. Given the time-scales of predictability considered, this can help MicroTE leverage traffic predictability longer, thereby achieving more optimal routing. The improvement in case of Vendor B is 1.6X (80ms to 48ms).

7.2.3 Two-level Responsive Traffic Engineering

Our experiments above evaluated individual components of Mazu in isolation. Next, we consider a network management application where FE and RO (with priority handling) both come into play.

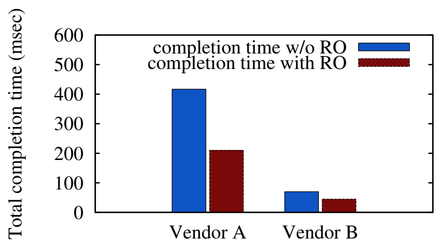
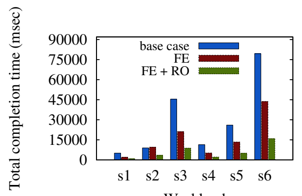


Figure 20: Flow setup time of MicroTE with and without RO in a Fat Tree (k=8) topology

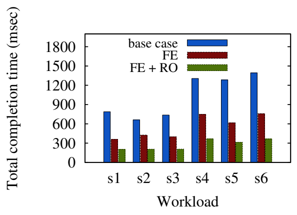
Our application is a two-level responsive traffic engineering scheme whose goal is to simultaneously route two categories of traffic – high and low priority – according to different objectives over the same underlying network. This could apply to an ISP that deploys (two classes of) service differentiation. The question we address here is how quickly can the network establish routes when requests arrive closely in time for both categories of traffic. The faster this is, the closer the network's ability to meet the SLAs for the corresponding classes. This experiment underscores the ability of Mazu to help applications that desire fine-grained state control in order to meet complex objectives.

To emulate this, in a somewhat simplistic fashion, we use different objective functions to route each category. For routing 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 in the switch to route this traffic. The objective for higher priority traffic is to minimize the overall link cost, where cost of a link could be latency. To route the high priority traffic we install fine grained high priority rules. These fine grained rules could overlap with the coarse grained rules and when they do, they have higher rule priority. At first we route the low priority traffic and then on the remaining network capacity we route the high priority traffic. For simplicity we assume that both categories of traffic can be accommodated without causing any congestion. The rest of the setup we use is same as in §7.2.1. The high and low priority traffic between any pair of nodes is proportional to their respective "popularity" indices (we assign popularity to the nodes at random in the range as shown in Table 3). For high volume workloads (s4, s5 and 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, s2 and s3) the volume of high priority and low priority traffic between any two overlay nodes is about 12 and 50 flows respectively.

Figures 21a and 21b show the total completion time using Vendor A and Vendor B 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 Vendor A. This implies that ignoring flow setup latency can cost traffic engineering dearly in terms of being responsive. For low volume



(a) Latency on Vendor A switch



(b) Latency on Vendor B switch

Figure 21: Worst case flow set up time of two level traffic engineering scheme with base case, FE and FE+RO techniques in a full mesh topology with 25 nodes

workloads (s1, s2, s3) the factor of improvement from just FE is about 2.5X for Vendor A and 1.8X for Vendor B, and is about 5X for Vendor A and 4X for Vendor B with FE+RO. We observe similar speedups for high volume workloads (s4, s5, s6).

We conclude that all the mechanisms in Mazu are crucial to ensuring that the route setup is sufficiently fast for management applications that desire fine-grained control to achiev complex objectives.

8. RELATED WORK

A few studies have considered SDN switch performance in the past. However, they have either focused on narrow issues, have not offered sufficient in-depth explanations for observed performance issues, or they did not explore implications on applications that require tight control. Devoflow [5] 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 [9, 20] provided a more in-depth look into switch performance across various vendors. In [20], 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 [9], 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. [5] 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 [22] 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. In [16], the authors design a rule manager that automatically partitions and places rules at both hypervisors and switches. Different from their goal of reducing computational load of host hypervisor, we wish to reduce path setup latency by enabling fast parallel execution of updates.

9. CONCLUSION

With the promise of flexible control by SDN, critical applications such as fast failover and mobility demand a tight interaction between the control plane and data plane. However, our measurement studies show that latency between controller and switches are highly variable depending on rule priorities in the flow table, the order of *flow_mod* operations and concurrent switch CPU activities. This creates significant challenges for SDN to support critical management applications. We present Mazu, a systematic framework to minimize such latencies. To reduce the latency from the switch to the controller, we bypass the slow embedded switch CPU completely by redirecting unmatched packets to a proxy. To reduce the latency of *flow_mod* operations, we introduce a novel concept, *flow engineering*, a mechanism to allow managemet applications to take path setup latency as a second objective. We then present 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 our mechanisms can tame flow setup latencies, thereby enabling SDN-based control of critical applications.

10. REFERENCES

[1] Al-Fares et al. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.
[2] Al-Fares et al. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.
[3] Benson et al. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *CoNEXT*, 2011.
[4] Casado et al. Rethinking Enterprise Network Control. *IEEE/ACM Trans. Netw.*, 2009.
[5] Curtis et al. DevoFlow: Scaling Flow Management for High-performance Networks. In *SIGCOMM*, 2011.
[6] Heller et al. ElasticTree: Saving Energy in Data Center Networks. In *NSDI*, 2010.
[7] Hong et al. Achieving High Utilization with Software-driven WAN. In *SIGCOMM*, 2013.
[8] Huang et al. A Close Examination of Performance and Power Characteristics of 4G LTE Networks. In *MobiSys*, 2012.
[9] Huang et al. High-fidelity Switch Models for Software-defined Network Emulation. In *HotSDN*, 2013.
[10] Jain et al. B4: Experience with A Globally-deployed Software Defined WAN. In *SIGCOMM*, 2013.
[11] Jin et al. SoftCell: Scalable and Flexible Cellular Core Network Architecture. In *CoNEXT*, 2013.
[12] Koponen et al. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.
[13] Liu et al. zUpdate: Updating Data Center Networks with Zero Loss. In *SIGCOMM*, 2013.
[14] Mahajan et al. On Consistent Updates in Software Defined Networks. In *HotNets*, 2013.
[15] McKeown et al. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 2008.
[16] Moshref et al. Scalable Rule Management for Data Centers. In *NSDI*, 2012.

[17] Olsson et al. Pktgen the Linux Packet Generator. In *Proceedings of the Linux Symposium, Ottawa, Canada*, 2005.
[18] Patel et al. Ananta: Cloud Scale Load Balancing. In *SIGCOMM*, 2013.
[19] Qazi et al. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *SIGCOMM*, 2013.
[20] Rotsos et al. Oflops: An Open Framework for Openflow Switch Evaluation. In *PAM*, 2012.
[21] Taylor et al. ClassBench: A Packet Classification Benchmark. *IEEE/ACM Trans. Netw.*, 2007.
[22] Yu et al. Scalable Flow-based Networking with DIFANE. In *SIGCOMM*, 2010.