# What You Need to Know About SDN Flow Tables

Maciej Kuźniar*, Peter Perešíni*, and Dejan Kostić‡

EPFL* KTH Royal Institute of Technology‡

**Abstract.** SDN deployments rely on switches that come from various vendors and differ in terms of performance and available features. Understanding these differences and performance characteristics is essential for ensuring successful deployments. In this paper we measure, report, and explain the performance characteristics of flow table updates in three hardware OpenFlow switches. Our results can help controller developers to make their programs efficient. Further, we also highlight differences between the OpenFlow specification and its implementations, that if ignored, pose a serious threat to network security and correctness.

## 1 Introduction

**Background** In OpenFlow-based Software Defined Networking (SDN) deployments [2, 5], SDN developers and network administrators (developers for short) write network programs at a logically centralized controller to control the network. The control plane involves the controller communicating with the switches' OpenFlow agents to instruct them how to configure the data plane by sending flow modification commands that place rules in the forwarding tables. OpenFlow's transition from research to production means that the new frameworks are taking reliability and performance [6, 12–15] to new levels that are necessary in the production environment. All of these assume quick rule installation latency, and rely on the switches to confirm successful rule installations.

**Measuring switch performance is a challenging task.** The biggest issue is that each switch under test has many "quirks" which result in unexplained performance changes. Therefore, the thorough evaluation and explanation of these phenomena takes a substantial effort and cannot be easily automated. For example, a switch may have vastly different performance characteristics for similar experiment setups and finding the responsible parameter and its value requires many tests. Same applies to trying out combinations of rule modifications.

**Our goal.** In this paper, we set out to advance the general understanding of OpenFlow switch performance. Specifically, our focus is on analyzing control plane processing times and flow table update rate in hardware OpenFlow switches that support version 1.0 of this protocol. This paper is *not* about data plane forwarding performance. Our contributions are as follows: (*i*) we go a step further in measuring OpenFlow switch control plane performance and its interaction with the data plane (for example, we dissect rule installation latency in a number of scenarios that bring the switch to the limit), (*ii*) we devise a more systematic way of switch testing, *i.e.*, along many different dimensions, than the

existing work, and (*iii*) we believe we are the first ones to report several new types of anomalous behavior in OpenFlow switches.

**Related work.** Curtis *et al.* [3] identify and explain the reasons for relatively slow rule installation rate on an HP switch. OFLOPS [16] observed that some OpenFlow agents did not support the barrier command. OFLOPS also reported some delay between the control plane's rule installation and the data plane's ability to forward packets according to the new rule. Huang *et al.* [4] perform switch measurements while trying to build High-Fidelity Switch models, and report slow flow setup rates. Relative to these works, we dissect switch performance at a finer grain, over a longer period of time, and more systematically in terms of rule combinations, initial parameters, etc. In addition, we identify the thresholds that reveal previously unreported anomalous behavior. Jive [11] proposes to build a proactive OpenFlow switch probing engine, and store switch behavior in a database. We show that the switch performance depends on so many factors that such a database would be difficult to create. NOSIX [17] optimizes commands for a particular switch based on its capabilities and performance. However, the authors do not analyze dynamic switch properties as we do; our work would be useful in improving the NOSIX optimization process.

**Key findings and impact.** Our key findings are as follows: (*i*) control plane performance is widely variable, and it depends on flow table size, priorities, batching of commands and even rule update patterns; (*ii*) switches might periodically or randomly stop processing control plane commands for up to 400 ms; (*iii*) data plane state might not reflect control plane—it might fall behind by up to 400 ms and it might also manifest rule installations in a different order; (*iv*) seemingly atomic data plane updates might not be atomic at all.

The impact of our findings is multifold and profound. The non-atomicity of seemingly atomic data plane updates means that *there are periods when the network configuration is incorrect despite looking correct from the control plane perspective*. The existing tools that check the control plane configuration [7–9] are unable to detect these problems. Moreover, the data plane can fall behind and unfortunately *barriers cannot be trusted*. Thus, the approaches for performing consistent updates need to devise a different way of defining when a rule is installed; otherwise they are not providing any firm guarantees. Our results show that interoperability between switches and controllers cannot be taken for granted. We hope that SDN controller and framework developers will find our findings useful in trying to ensure consistent performance and reliability from the variety of switches they may encounter. Also, efforts that are modeling switch behavior [4] should consult our study.

## 2 Measurement Methodology

**Tools and experimental setup.** The main requirements for our tool are (*i*) portability, (*ii*) flexibility, and (*iii*) sufficient precision. First, since the switches we test are often in locations with limited physical access, the measuring tool cannot use customized hardware (*e.g.*, FPGAs). Our previous experience suggests that switches behave unexpectedly, and thus we need to tailor the experiments to

locate and dissect problems. Finally, as the tested switches can modify at most a couple thousands of rules per second, we assume that a millisecond measurement precision is sufficient. To achieve the aforementioned goals we built a tool that consists of three major components that correspond to the three benchmarking phases: input generation, measurement and data analysis (Fig. 1).

First, an input generator creates control plane rule modification lists as well as data plane packet traces and saves them to text and pcap files. Unless otherwise specified, the rules match packets based on IP src/dst and forward to a single port. Because we noticed that some switches optimize updates for the same rule, we use consecutive IPs for matches (to make sure we modify different rules), but we also cross-check our results using random matches and update patterns.

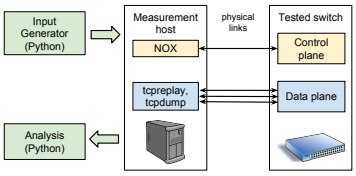


Fig. 1: Overview of our measurement tools and testbed setup.

We refer to the control plane measurement engine as the controller as it emulates the behavior of an OpenFlow controller. We implement it as a module in the NOX controller platform that can issue rule updates at a much higher rate than what the hardware switches can handle.[1] The engine records time of interactions with the switch (*e.g.*, flow modification sent, barrier reply received).

Our experiments require injecting and recording data plane packets to precisely measure when the flow table is updated. We rely on tcpreplay and tcpdump tools to send packets based on a pcap file and record them. To avoid time synchronization issues, the switch is connected to a single host. The host handles the control plane and generates and receives data plane traffic.[2] Network RTT between the host and the switches is between 0.1 and 0.5ms. Finally, an analysis tool reads the outputs and computes the metrics of interest. Modularity lets us easily analyze different aspects of the captured data.

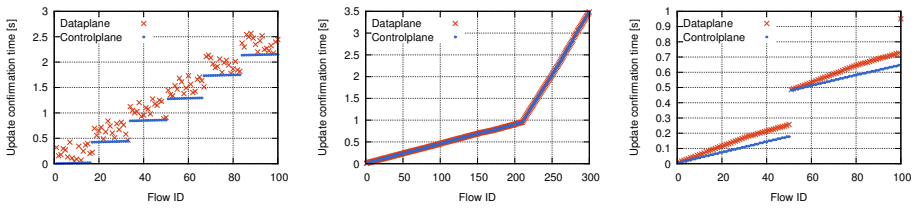
**Switches under test.** We benchmark three switches with OpenFlow 1.0 support: HP ProCurve 5406zl with K.15.10.0009 firmware, Pica8 P-3290 with PicOS 2.0.4, and Dell PowerConnect 8132F with beta[3] OpenFlow support (both P-3290 and 8132F belong to the newest generation of OpenFlow switches). They use ProVision, Broadcom Firebolt and Broadcom Trident+ ASICs respectively. Such switches have two types of forwarding tables: software and hardware. While hardware table sizes (about 1500, 2000, and 750 rules, respectively) and levels of OpenFlow support vary, we make sure that all test rules ultimately end up in hardware tables. The point of this study is not to advertise or discredit any switch but to present interesting characteristics and to highlight potential issues.

**General experiment setup.** In most experiments in this paper we use the following generic setup and modify only particular parameters. At the beginning of each experiment we prepopulate the switch flow table with $R$ default priority,

[1] Our benchmark with software OpenVSwitch handles ∼42000 rule updates/s.

[2] Note that we do not need to fully saturate the switch data plane, and thus a conventional host is capable of handling all of these tasks at the same time.

[3] The software is going to be optimized and productized in a near future.




(a) P-3290 (b) 8132F (c) 5406zl

Fig. 2: Control plane confirmations and data plane probes. Data plane updates may fall behind the control plane acknowledgments and may be even reordered.

non overlapping rules forwarding packets matching flows number $1 - R$ to 0 to port A. After the switch applies this update in the hardware flow table, the measured run starts. We send $B$ batches of rule updates, each batch consisting of: $B_D$ rule deletions, $B_M$ rule modifications and $B_A$ rule insertions followed by a barrier request. In the default setup $B_D = B_A = 1$ and $B_M = 0$. Batch $i$ deletes the rule matching flow number $i - R$ and installs a rule that matches flow $i$ and forwards packets to port A. Note that the total number of rules in the table is stable during the experiment (in contrast to previous work that measures only the time needed to fill an empty table). If the experiment requires injecting and capturing data plane traffic, we send packets that belong to flows $F_{start}$ to $F_{end}$ at a rate of about 1000 packets/s. For clarity, when describing an experiment we change only one parameter. In reality we vary different parameters as well to confirm the observations. Finally, unless an experiment shows variance greater than 5% across runs, we repeat it three times and report the average.

## 3 Data Plane

While the only view the controller has of the switch is through the control plane, the traffic forwarding happens in the data plane. In this section we present experiments where we monitor rule updates in the control plane and at the same time send traffic to exercise the updated rules. [4]

### 3.1 Synchronicity of Control and Data Planes

Many solutions essential for correct and reliable OpenFlow deployments (*e.g.*, [12, 15]) rely on knowing when the switch applied a given command *in the data plane*, and they resort to using the barrier message for the task.[5] However, as authors of [16] already hinted, the state of the data plane may be different than the one advertised by the control plane. Thus we set out to measure how do these two views correspond to each other at a fine granularity.

[4] While experimenting and digging deep to understand the root causes of various behaviors we made other, less critical observations described in a tech report [10].

[5] As specified, after receiving a barrier request, the switch has to finish processing all previously-received messages before executing any messages after the barrier request. When the processing is complete, the switch must send a barrier reply message [1].

We use the default setup extended with one match-all low priority rule that drops all packets[6]and we inject data plane flows number $F_{start}$ to $F_{end}$. For each update batch $i$ we measure the time when the controller receives a barrier reply for this batch and when the first packet of flow $i$ reaches the destination. To work around the limited rate at which the testing machine can send and capture packets (100000 packets/s), we send traffic in 100-flow parts. Since the results for 5406zl and P-3290 are similar for each part we show plots for only one range. For 8132F we merge the results for three ranges to show the change in behavior.

Results for $R = 300$, $B = 500$, $F_{start} = 1$ with $F_{end} = 100$ (5406zl and P-3290) and $F_{end} = 300$ (8132F) are in Fig. 2. Each switch behaves differently.

**5406zl:** The data plane configuration of *5406zl is slowly falling behind* the control plane acknowledgments – packets start reaching the destination long after the switch confirms the rule installation with a barrier reply. After about 100 rule updates (we observed that adding or deleting a rule counts as one update, and modifying an existing rule as two), the *switch stops responding with barrier replies for 300ms*, which allows the flow tables to catch up. After this time the process of diverging starts again. The divergence increases linearly and, in this experiment, reaches up to 82ms, but *can be as high as 250ms* depending on the number of rules in the flow table. The 300ms inactivity time is constant across all experiments we run, but happens three times more often (every 33 updates) if there are over 760 rules in the flow table. Moreover, the frequency and the duration of this period do not depend on the rate at which the controller sends updates, as long as there is at least one update every 300ms. The final observation is that *5406zl installs rules in the order of their control plane arrival.*

**P-3290:** Similarly to 5406zl, P-3290 stops responding to barriers in regular intervals. However, unlike 5406zl, it is either processing control plane (handling update commands, responding to barriers), or installing rules in TCAM and never does both at the same time. Moreover, *despite the barriers, the rules are not installed in hardware in the order of arrival.* The delay between data and control plane reaches *up to 400ms* in this experiment. When all remaining rules get pushed into hardware, the switch starts accepting control plane commands again. We confirmed with a vendor that because the synchronization between the software and hardware table is expensive, it is performed in batches and the order of updates in a batch is not guaranteed. When the switch pushes updates to hardware, its CPU is busy and it stops dealing with the control plane.[7]

**8132F:** Finally, *8132F makes sure that no control plane confirmation is issued before a rule becomes active* in hardware. There are also no periods of idleness as the switch pushes rules to hardware all the time and waits for completion if necessary.[8] Interestingly, the switch starts updating rules quickly, but suddenly

[6] We need to use such a rule to prevent flooding the control channel with the PacketIn messages caused by data plane probes or flooding the probes to all ports.

[7] The vendor claims that this limitation occurs only in firmware prior to PicOS 2.2.

[8] We observe periods when the switch does not install rules or respond to the controller, but these periods are rare, non reproducible and seem unrelated to the experiments. We think they are caused by periodic background processing at the switch.

slows down after 210 new rules installed and maintains this slower speed (verified up to 2000 batches). However, even after the slowdown, the control plane reliably reflects the state of the data plane configuration.

***Summary**: To reduce the cost of placing rules in a hardware flow table, vendors allow for different types (e.g., falling behind or reordering) and amounts (e.g., up to 400ms) of temporary divergence between the hardware and software flow tables. Therefore, the barrier command does not guarantee flow installation.* ***Ignoring this problem leads to an incorrect network state that may drop packets, or even worse, send them to an undesired destination!***

### 3.2 Rule Modifications Are not Atomic

Previously, we observed unexpected delays for rule insertions and deletions. A natural next step is to see if modifying existing rules exhibits a similar behavior.
**A gap during a FlowMod:** As before, we prepopulate the flow table with one low priority match-all rule dropping all packets and $R = 300$ flow specific rules forwarding to port A. Then, we modify these 300 rules to forward to port B. At the same time, we send data plane packets matching rules $101 - 200$ at a rate of 1000 packets/s per flow. For each flow, we record a gap between when the last packet arrives at the interface connected to port $A$ and when the first packet reaches an interface connected to B. Expected time difference is 1ms because of our measurement precision, however, we observe gaps lasting up to 7.7, 12.4 and 190ms on P-3290, 8132F and 5406zl respectively. At 5406zl the longest gaps correspond to the switch inactivity times described earlier (flow 150, 200).
**Drops and unexpected actions:** To investigate the forwarding gap issue further we add a unique identifier to each packet to detect if they are being lost or reordered. Moreover, to get higher precision, we probe only a single rule (number 151 – a rule with an average gap, and number 150 – a rule with a long gap on 5406zl) and increase our probing rate to 5000 packets/s.

We observe that P-3290 does not drop any packets. A continuous range of packets arrive at port A and the remaining packets at B. On the other hand, both 8132F and 5406zl drop packets at the transition period for rule 151 (3 and 17 packets respectively). For rule number 150, 5406zl drops an unacceptable number of 782 packets. When we replace the drop-all rule with a rule that forwards all traffic to port C, identifiers of packets captured on port C for both 5406zl and 8132F fit exactly between the series at ports A and B. This suggests that the *update is not atomic*—a rule modification deactivates the old version and inserts the new one, with none of them forwarding packets during the transition.

To further investigate this behavior, we repeat the experiment with no low priority rule at all. Both switches flood packets to all ports during the transition. While it follows the no match behavior of 8132F, it is surprising for 5406zl, since by default non-matching packets cause PacketIn messages. The only imperfection of P-3290 is that if the output port of the same rule gets updated between ports A and B frequently, some packets arrive at the destination out of order.

***Summary**: Two out of three tested switches have a transition period during a rule modification when the network configuration is neither in the initial nor*

| Switch | Observed/inferred behavior |
|---|---|
| P-3290 | May temporarily reorder for overlapping matches (depending on wildcards). OK for the same match. |
| 8132F | OK (Note: May temporarily reorder if not separated by a barrier) |
| 5406zl | Ignores priority, last updated rule permanently wins |

Table 1: Priority handling of overlapping rules. Only 8132F behaves as defined in the OpenFlow specification.

*the final state.* ***The observed action of forwarding packets to undesired ports is a security concern****. Non-atomic flow modification contradicts the assumption usually made by controller developers and network update solutions. Our results suggest that either switches should be redesigned or the assumptions made by the controllers have to be revisited to guarantee network correctness.*

### 3.3 Priorities and Overlapping Rules

The OpenFlow specification clarifies that, if rules overlap (*i.e.*, two rules match the same packet), packets should always be processed only by the highest priority matching rule. Since our default setup with IP src/dst matches prevents rule overlapping, we run an additional experiment to verify the behavior of switches when rules overlap. We install rules that can match the same packet: $R_{hi}$ that has a higher priority and forwards to port A, and $R_{lo}$ that forwards to B. $R_{hi}$ is always installed before and removed after $R_{lo}$ to prevent packets from matching $R_{lo}$. Initially, there is one low priority drop-all rule and 150 pairs of $R_{hi}$ and $R_{lo}$. Then we send 500 update batches, each removing and adding one rule: $(-R_{lo,1}, +R_{hi,151}), (-R_{hi,1}, +R_{lo,151}), (-R_{lo,2}, +R_{hi,152}), \ldots$ We send data plane traffic for 100 flows. If a switch works correctly, no packets should reach port B.

Table 1 summarizes the results. First, as we already noted, 8132F does not reorder updates between batches and therefore, there are no packets captured at port B. The only way to allow some packets on port B is to increase the batch size – the switch freely reorders updates inside a batch (which is allowed by the specification) and seems to push them to hardware in order of priorities. On the other hand, P-3290 applies updates in the correct order only if the high priority rule has the IP source specified. Otherwise, for a short period of time—210 ms on average, 410 ms maximum in the described experiment—packets follow the low priority rule. Our hypothesis is that the software flow table data structure sorts the rules such that when they are moved to hardware the ones with IP source specified are pushed first. Finally, in 5406zl, only the first few packets of each flow (for 80 ms on average, 103 ms max in this experiment) are forwarded to A and all the rest to B. We conclude that the switch ignores priorities in hardware (as documented for the older firmware version) and treats rules installed later as more important. We confirm this hypothesis with additional experiments not reported here. Further, because the priorities are trimmed in hardware, installing rules with the same match but different priorities and actions causes an error.

***Summary****: Results (Table 1) suggest that switches may permanently or temporarily forward according to incorrect, low priority rules.*

| Experiment | In-flight batches | Batch size (del+add) | Initial rules $R$ |
|---|---|---|---|
| In-flight batches | 1-20 | 1+1 | 300 |
| Flow table size | 2 | 1+1 | 50 to max for switch |
| Priorities | as in Flow table size + a single low priority rule in the flow table | | |
| Access patterns | 2 | 1+1 | 50 to max for switch +priorities |
| Working set | as in Flow table size, vary the number of rules that are not updated during the experiment | | |
| Batch size | 2 | 1+1 to 20+20 | 300 |

Table 2: Dimensions of experimental parameters we report in this section. Note that we also run experiments for other combinations to verify the conclusions.

## 4 Flow Table Update Speed

The goal of the next set of experiments is to pinpoint the most important aspects that affect rule update speed. From the previous section we know that although the control plane information is imprecise, in a long run the error becomes negligible (all switches synchronize the data and control plane views regularly). We first identify various performance-related parameters: the number of in-flight commands, current flow table size, size of request batches, used priorities, rule access patterns. Then we sample the whole space of these parameters and try to identify the ones that cause some variation. Based on the results, we select a few experimental configurations which highlight most of our findings in Table 2.

### 4.1 Two In-flight Batches Keep the Switch Busy

The number of commands a controller should send to the switch before receiving any acknowledgments is an important design decision [14]. Underutilizing or overloading the switch with commands is undesired. Here, we quantify the tradeoff between rule update rate and the servicing delay (time between sending a command and the switch applying it) to find a performance sweet spot.

We use the default setup with $R = 300$ and $B = 2000$ batches of rule updates. The controller sends batch $i + k$ only when it receives a barrier reply for batch number $i$. We vary $k$ and report the average update rate, which we compute as $(1 + 1) * B$ (because each batch contains one add and one delete) divided by the time between sending the first batch and receiving a barrier reply for the last.

Figure 3 shows the average rate across eight runs. The rule update rate with one outstanding batch is low as the switch is idle for at least a network RTT. However, even two in-flight batches are sufficient to saturate all tested switches given our network latencies. Thus, we use 2 in-flight batches in all experiments.

Looking deeper into the results, we notice that with a changing number of in-flight batches 5406zl responds in an unexpected way. In Fig. 4 we plot the barrier reply arrival times normalized to the time when the first batch was sent for $R = 300$, $B = 50$ and a number of in-flight batches varying between 1 and 50. We show the results for only 4 values to improve readability. If there are requests in the queue, the switch batches the responses and sends them together

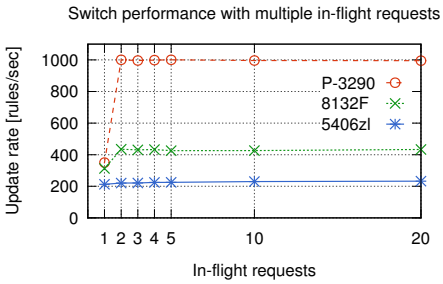



Fig. 3: Update rate improvement for over 2 in-flight requests is negligible.

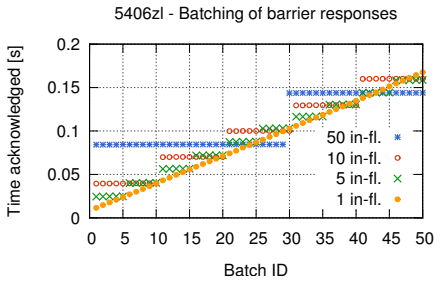



Fig. 4: 5406zl barrier reply arrivals. It holds replies for up to 29 requests.

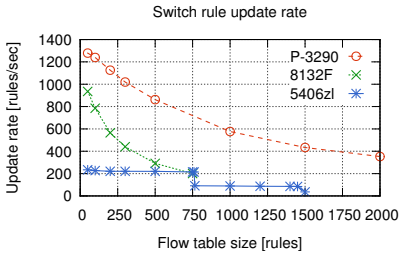



Fig. 5: Update rate decreases when the number of rules in the flow table grows.

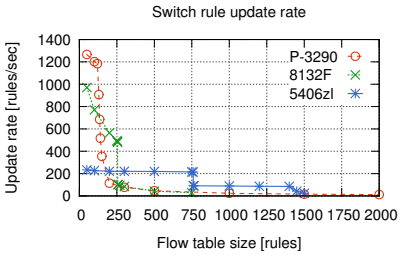



Fig. 6: Priorities cripple performance. One low-priority rule significantly decreases update rate.

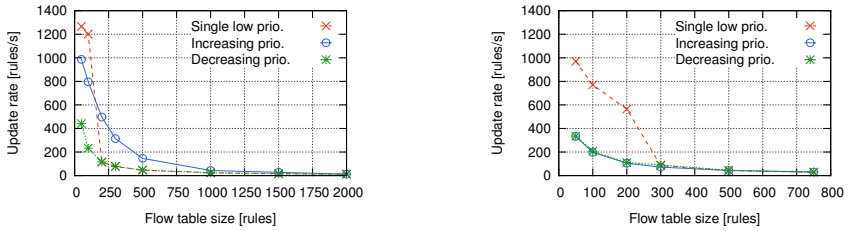
in bigger groups. If a continuous stream of requests is shorter than 30, the switch waits to process all, otherwise, the first response comes after 29 requests.

***Summary**: We demonstrated that with LAN latencies two in-flight batches suffice to achieve full switch performance. Since, many in-flight requests increase the service time, controllers should send only a handful of requests at a time.*

### 4.2 Current Flow Table Size Matters

The number of rules stored in a flow table is a very important parameter of a switch. Bigger tables allow for a fine grained traffic control. However, there is a well known tradeoff—TCAM space is expensive, so tables that allow for complex matches usually have limited size. We discover another, hidden cost of full flow tables. We use the default setup fixing $B = 2000$ and changing the value of $R$.

In Fig. 5 we report the average rule update rate. There are two distinct patterns. Both P-3290 and 8132F express similar behavior—the rule update rate is high with a small number of entries in the flow table but quickly deteriorates as this number increases. As we confirmed with one of the vendors and deduced based on statistics of the other switch, there are two reasons why the performance drops. First, even if a switch installs rules in hardware, it keeps a software flow table copy as well. The flows are first updated in the software data structure which takes more time when the structure is bigger. Second, the rules need to be pushed into hardware (the switch ASIC), which may require rearranging the




(a) P-3290 (b) 8132F

Fig. 7: Switch rule update performance for different rule access patterns.

existing entries. On the other hand, 5406zl maintains a lower, but stable rate following a step function with a breaking point around 760 rules in the flow table. This stability is caused by periods of inactivity explained in Section 3.

***Summary**: The performance of all tested switches drops with a number of installed rules, but the absolute values and the slope of this drop vary. Therefore, controller developers should not only take into account the total flow table size, but also what is the performance cost of filling the table with additional rules.*

### 4.3 Priorities Decrease the Update Rate

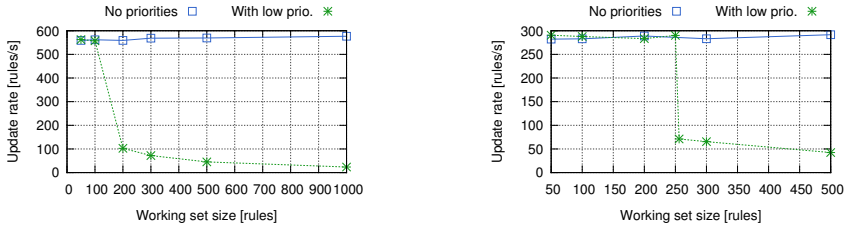
Next, we conduct an experiment that mimics a situation where a lowest priority all-matching rule drops all packets that do not match any other rule. The experiment setup is exactly the same as the one described in Section 4.2 with one additional lowest priority drop-all rule installed before all flow-specific rules.

Figure 6 shows that for a low flow table occupancy, all switches perform comparably as without the low priority rule. However, P-3290 and 8132F suffer from a significant drop in performance at about 130 and 255 installed rules respectively. After this massive drop, the rate gradually decreases until it reaches 12 updates/s for 2000 rules in the flow table for P-3290 and 30 updates/s for 750 rules in the flow table for 8132F where both switches have their tables almost full. Interestingly, 5406zl's update rate does not decrease so much, possibly because it ignores the priorities. We confirm that the results are not affected by the fully wildcarded match or the drop action in the low priority rule by replacing it with a specific IP src/dst match and a forwarding action.

Finally, we rerun the experiments from Section 4.1 with a low priority rule. The absolute rates are lower, but the characteristics and the conclusions hold.

**More priorities:** Now, we check what is the effect of using different priorities for each rule. We modify the default set-up such that each rule has a different priority assigned and install them in an increasing or decreasing order.

Switches react differently: P-3290's and 8132F's performance follows a similar curve as in the previous experiment, but there is no breaking point (Figure 7). In both cases the rate is higher with one different priority rule until the breaking point, after which they equalize. Moreover, P-3290 updates rules quicker in the increasing priority order (consistent with [11], but the difference is smaller as




(a) P-3290 (b) 8132F

Fig. 8: Size of the rule working set affects the performance. For both P-3290 and 8132F with low priority rule, the performance depends mostly on the number of rules constantly updated and not on the total number of installed rules.

for each addition we also delete a rule). 5406zl is unaffected by the priorities, but our data plane study shows a serious divergence between the control plane reports and the reality for this switch in this experiment (see Trivia in [10]).

**Working set size:** Finally, we check what happens if only a small subset of rules in the table (later referred as "working set") is frequently updated. We modify the default setup such that batch $i$ deletes the rule matching flow $i - W$ and installs a rule matching flow $i$. We vary the value of $W$. In other words, the first $R - W$ rules never change and we update only the last $W$ rules.

The results show that 5406zl's performance remains the same as presented in Figures 5 and 6. Further, for both P-3290 and 8132F a small update working set makes no difference if there is no low priority rule. For a given $R$ (1000 for P-3290 and 500 for 8132F in Fig. 8), the performance is constant regardless of $W$. However, with the low priority rule installed, the update rate characteristic changes (Figure 8). For both switches, as long as the update working set is smaller than their breaking point revealed in Section 4.2, the performance stays as if there was no drop rule. After the breaking point, it degrades and is marginally worse compared to the results in Section 4.2 for table size $W$.

***Summary**: The switch performance is difficult to predict—a single rule can degrade the update rate of a switch by an order of magnitude. Controller developers should be aware of such behavior and avoid potential sources of inefficiencies.*

### 4.4 Rule Modifications Are Slower than Additions and Deletions

We run the same experiments as described in previous subsections, but modifying existing rules instead. Because the results are very similar, we do not report them here in detail. All plots follow the same curves, but in general the update rate is between 0.5x and 0.75x of the rate for additions and deletions for P-3290 and 8132F. For 5406zl the difference is much smaller and stays within 12%.

## 5 Conclusions and Future Work

In this paper we try to shed light on the state of OpenFlow switches – an essential component of relatively new, but quickly developing Software Defined

Networks. The main takeaway is that despite a common interface, the switches are more diverse than one would expect, and this diversity has to be taken into account when building controllers. Because of the limited resources, we obtained sufficiently long access only to three switches. In the future, we plan to keep extending this study with additional devices to obtain the full picture.

**Acknowledgments** We thank Marco Canini, Dan Levin and Miguel Peón for helping us get access to the tested switches. We also thank Pica8 and Dell representatives for quick responses and explanations. We thank the reviewers, who provided excellent feedback. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement 259110.

## References

1. OpenFlow Switch Specification. `http://www.openflow.org/documents/openflow-spec-v1.0.0.pdf`.
2. Ethernet Switch Market: Who's Winning?, 2014. `http://www.networkcomputing.com/networking/d/d-id/1234913`.
3. A. Curtis, J. Mogul, J. Tourrilhes, and P. Yalagandula. DevoFlow: Scaling Flow Management for High-Performance Networks. In *SIGCOMM*, 2011.
4. D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity switch models for software-defined network emulation. In *HotSDN*, 2013.
5. S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-Deployed Software Defined WAN. In *SIGCOMM*, 2013.
6. N. P. Katta, J. Rexford, and D. Walker. Incremental Consistent Updates. In *HotSDN*, 2013.
7. P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking using Header Space Analysis. In *NSDI*, 2013.
8. P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *NSDI*, 2012.
9. A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *NSDI*, 2013.
10. M. Kuźniar, P. Perešíni, and D. Kostić. What you need to know about SDN control and data planes. Technical Report EPFL-REPORT-199497, EPFL, 2014.
11. A. Lazaris, D. Tahara, X. Huang, L. E. Li, A. Voellmy, Y. R. Yang, and M. Yu. Jive: Performance Driven Abstraction and Optimization for SDN. In *ONS*, 2014.
12. H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. A. Maltz. zUpdate: Updating Data Center Networks with Zero Loss. In *SIGCOMM*, 2013.
13. R. Mahajan and R. Wattenhofer. On Consistent Updates in Software Defined Networks. In *HotNets*, 2013.
14. P. Perešíni, M. Kuźniar, M. Canini, and D. Kostić. ESPRES: Transparent SDN Update Scheduling. In *HotSDN*, 2014.
15. M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *SIGCOMM*, 2012.
16. C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. Oflops: An open framework for openflow switch evaluation. In *PAM*, 2012.
17. M. Yu, A. Wundsam, and M. Raju. NOSIX: A Lightweight Portability Layer for the SDN OS. *ACM SIGCOMM Computer Communication Review*, 44(2), 2014.