Empirical Predictability Study of SDN Switches

Amaury Van Bemten
Chair of Communication Networks
Technical University of Munich
Munich, Germany
amaury.van-bemten@tum.de

Nemanja Đerić
Chair of Communication Networks
Technical University of Munich
Munich, Germany
nemanja.deric@tum.de

Amir Varasteh
Chair of Communication Networks
Technical University of Munich
Munich, Germany
amir.varasteh@tum.de

Andreas Blenk
Chair of Communication Networks
Technical University of Munich
Munich, Germany
andreas.blenk@tum.de

Stefan Schmid
Faculty of Computer Science
University of Vienna
Vienna, Austria
stefan_schmid@univie.ac.at

Wolfgang Kellerer
Chair of Communication Networks
Technical University of Munich
Munich, Germany
wolfgang.kellerer@tum.de

***Abstract*—To meet their increasingly stringent dependability requirements, communication networks need to be predictable, both in terms of correctness and performance. In principle, Software-Defined Networks (SDN) enable such more predictable networks, however, these networks still depend the underlying switches. This paper presents an empirical study of the predictability of SDN switches. Our extensive benchmarking of seven hardware OpenFlow switches from four different manufacturers raises several concerns regarding the dependability of these switches.**

We uncover several incorrect and unpredictable behaviors and performance issues. In particular, we identify unpredictable behaviors related to the management of flows and buffers, and observe that existing quality-of-service mechanisms, such as priority queuing, introduce unexpected overheads. The latter, in turn, can lead to violations of latency guarantees. Based on our insights, we discuss first solutions toward more predictable architectures.

***Index Terms*—predictability, latency, guarantees, measurements, software-defined networking, programmable switches**

I. Introduction

Communication networks are a critical backbone of our digital society, and many emerging applications, e.g., related to entertainment, business, or health, introduce increasingly stringent dependability requirements. In particular, it is required that communication networks become highly *predictable*, both in terms of correctness as well as in terms of performance (e.g., end-to-end latency). In turn, predictability of networking devices enables operators to provide strict performance guarantees (e.g., latency guarantees) to their tenants — an indispensable feature in data centers or 5G networks [1], [2]. In this paper, we define predictability as the ability of a network device or a complete network to perform its operations with strict performance guarantees that can be predicted in advance.

Motivated by these requirements, over the last years, much progress has been made toward enabling more predictable communication networks. In particular, Software-Defined Networks (SDNs), with their principled approach to specify and operate networks, as well as the introduced direct control over network devices and the resulting management flexibilities [3], have the potential to greatly improve dependability and efficiency, at least from a control plane logic point of view. However, while these trends are promising, the dependability of a network, whether software-defined or not, is at most as good as the dependability of its data plane. In fact, even seemingly simple tasks, such as forwarding, involve many complex components, such as link buffers, hardware memory units, switch CPUs, queuing disciplines, etc. A deep understanding of the behavior of all these components is necessary for guaranteeing predictable network operations. Not surprisingly, already much research went into the study of solutions for predictable networks, e.g., related to latency issues: researchers derived many end-to-end mathematical models for per-packet latency guarantees [4]–[7]. However, these approaches still rely on an expected standard behavior, i.e., model, of forwarding hardware, whose correctness cannot be verified by the packet-level simulations or end-to-end measurements performed in these studies.

This paper is motivated by the following fact: the predictability of communication networks, both in terms of correctness and performance, critically depends on the underlying hardware, and especially the network devices used to process and forward packets. In order to shed light on the predictability of these network devices, we present an extensive measurement study of the behavior of SDN switches. In particular, we systematically benchmark seven state-of-the-art SDN switches from four vendors in order to analyze the predictability of their behavior with respect to important metrics such as processing time and throughput, as well as with respect to the mechanisms used to ensure quality-of-service (such as priority queuing). We also examine management aspects, e.g., related to flow tables and queues (packet buffers).

Our findings are rather negative: none of the examined switches can directly be used with the aforementioned models in order to provide predictable latency (Tab. I). In particular, we show that basic, indispensible and most light-weight QoS mechanisms, such as priority queueing, already introduce significant overheads. When determining the behavior of switches in terms of their flow table and buffer management, we

Switch	PT *II-A*	PQ *II-B*	TP *II-C*	FM *III-A*	BM *III-B*
HP E3800	+	-	-	-	
HP 2920	+	-	-	-	
Dell S3048-ON	+	∼	+	-	-
Dell S4048-ON	+	∼	+	-	-
Pica8 P3290	+	∼	+	-	-
Pica8 P3297	+	∼	+	-	-
NEC PF5240	+	∼	+	-	∼

TABLE I: Five predictability dimensions (here: regarding latency) of forwarding devices and whether they are verified for the seven switches. Green means a switch behaves as expected/predicted, orange means a switch partially behaves as expected, red means a switch does not behave as expected and gray means that it is not applicable because of another unmet requirement. None of the switches are predictable along all the five selected metrics.

find that most operations are not predictable or too slow for production deployments. At the same time, we show that once aware of these issues, solutions may exist, and hope that our study can help contribute toward more predictable network architectures.

Our contributions. The main contribution of this paper is an extensive measurement study of different SDN switches and their impact on network predictability. We examine different dimensions, including throughput, buffering architectures, control-plane-data-plane interference and consistency, and identify several shortcomings which render predictable behavior challenging: for example, we find that not all switches support line rate forwarding as promised, we identify *aging effects* (some switches suffer from reduced table size over time), we observe that some switches blindly drop forwarding rules, that priority queuing comes with a so far neglected processing time overhead, or that packet buffers are not isolated per-port or -queue, as assumed by traditional mathematical models (e.g., network calculus) for buffer dimensioning. We also contribute a new measurement methodology for determining the throughput of programmable devices, and initiate the discussion of architectural modifications that can overcome the observed shortcomings.

As a contribution to the research community and to ensure reproducibility, all the data sets, source code and configuration files associated to the results presented in this article are publicly available online [8].

Organization. The remainder of this paper is organized as follows. The performance and management predictability measurements and results are presented in Sec. II and Sec. III, respectively. We give some insights and discussions over these results in Sec. IV. Thereafter, we present the related work in Sec. V, followed by concluding remarks in Sec. VI.

II. Performance Predictability

This section (Sec. II) and the next section (Sec. III) report on our predictability analysis of different switches. Whereas this section focuses on forwarding performance, the correctness of assumptions with respect to management tasks is analyzed in Sec. III.

Tab. II lists the seven investigated switches, representing a wide range of devices: 4 different vendors, different switches per vendors; both SDN-tailored (e.g., Pica8) and general (e.g., HP) switches; both 1G and 10G devices and both high-end (e.g., Dell) and lower-end switches (e.g., NEC and Pica8). The challenge in benchmarking switches is that information about their internal functioning, i.e., what exact components are traversed by packets when they are forwarded, is not publicly available. Manufacturers are reluctant to open their architecture, as illustrated by the fact that we sometimes do not even know the ASIC or CPU model of a switch (Tab. II). Besides, when manufacturers actually describe internals, we will see that such documentation can be erroneous or outdated (Sec. III-B). This suggests that switches have to be considered blackboxes for our study.

The results of the empirical predictability study of this paper are summarized in Tab. I, structured into the different requirements and assumptions made: whereas some switches like the NEC support more assumptions, *none of the investigated switches exhibit predictable behavior along all the investigated dimensions*. The requirements, discussed and justified in separate subsections within this (Sec. II) and the next section (Sec. III), are selected based on the assumptions made by state-of-the-art end-to-end strict latency models regarding the behavior of forwarding devices [4]–[7].

Throughout the paper, we use OpenFlow v1.0 for configuring the switches because it supports all the features required to deploy the state-of-the-art latency models [4]–[7], a major focus of this paper.

A. PT — *Processing Time*

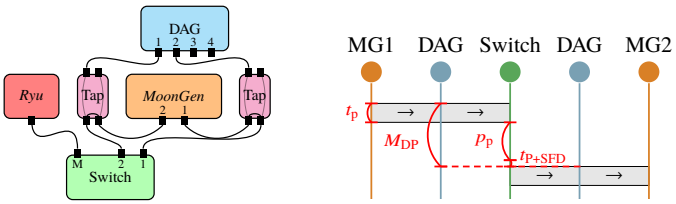
End-to-end delay is the sum of propagation, transmission, processing, and queuing delay. The propagation and transmission delays are physical values directly computed from the physical link properties. The processing and queuing delay are the critical components: they are determined by switch-internal functions. While queuing delay is computed using mathematical models (see Sec. II-B), state-of-the-art approaches assume that the processing time of the switches is deterministically bounded by a *constant* value [5], [7].

We evaluate the processing time of our switches in different settings to assess whether it is indeed bounded by a predictable value for different modes of operation. We vary the *matching* and *actions* properties of rules, their *number*, their *priority*, the *matching rule*, the *rate* of (data plane) packets and the *packet size*. These considered dimensions and their respective values are shown in Tab. III. For the *matching* values, combination of fields are also considered: *five-tuple* includes L3/L4 source/destination and L3 protocol; *all* includes L2/L3/L4 source/destination, L3 ToS and L3 protocol. For each *action* type, an additional *output* action is included. The *all* action consists of *output*, *set-dl-src*, *push-vlan*, *set-vlan-id*, *set-vlan-pcp*, and *set-nw-tos* (as shown later, L3/L4 modifications are never realized in hardware).

1) Measurement Setup: A *Ryu*-based [9] controller generates a flow table according to the selected dimension values

TABLE II: Specifications of the investigated switches: names, ASIC, CPU, firmware and ports.

Switch	ASIC	CPU	Firmware *(release date)*	Ports
HP E3800	HPE ProVision	Freescale P2020	KA.16.04.0016 *(2018-06-22)*	48×1G-RJ45 + 4×10G-SFP+
HP 2920	HPE ProVision	Tri Core ARM1176	WB.16.08.0001 *(2018-11-28)*	24×1G-RJ45
Dell S3048-ON	Broadcom StrataXGS	*undisclosed*	DellOS 9.14 *(2018-07-13)*	48×1G-RJ45 + 4×10G-SFP+
Dell S4048-ON	*undisclosed*	*undisclosed*	DellOS 9.14 *(2018-07-13)*	48×10G-SFP+ + 6×40G-QSFP+
Pica8 P3290	Broadcom Firebolt 3	Freescale MPC8541CDS	PicOS 2.10.2 *(2018-01-19)*	48×1G-RJ45 + 4×10G-SFP+
Pica8 P3297	Broadcom Triumph 2	Freescale P2020	PicOS 2.11.19 *(2019-02-27)*	48×1G-RJ45 + 4×10G-SFP+
NEC PF5240	*undisclosed*	*undisclosed*	OS-F3PA 6.0.0.0 *(2014-06)*	48×1G-RJ45 + 4×10G-SFP+



(a) *PT* measurement setup. (b) *PT* sequence diagram.

Fig. 1: (a) Switch processing time measurement setup and (b) corresponding sequence diagram.

Dimension	Values
num. of entries	1, **100**, 200, 300, 400, 500
match type	*port*, *dl-dst*, *dl-vlan*, *dl-vlan-pcp*, *masked-nw-dst*, *tp-dst*, ***five-tuple***, *all*
action	***output***, *enqueue*, *set-dl-src*, *set-vlan-id*, *set-vlan-pcp*, *strip-vlan*, *push-vlan*, *set-nw-src*, *set-nw-tos*, *set-tp-src*, *all*
matching rule	*first*, ***last***
priorities	***increasing***, *decreasing*, *same*
packet size [bytes]	64, **306**, 548, 790, 1032, 1274, 1516
rate [Mbps]	5, **100**, 500, 750, 900, 950, 1000

TABLE III: Dimensions (and their values) considered for the processing time measurements. Bold values correspond to the default values.

(Fig. 1). The matching flow entry is configured to be forwarded to port 2 of the switch. We use *MoonGen* [10] to generate packets with the appropriate header fields, packet size and rate. Packets arriving (port 1) and leaving (port 2) the switch are mirrored using network taps to a nanosecond-precise Endace DAG 7.5G4 measurement card. The card timestamps packets upon arrival of the start frame delimiter (SFD) [11]. The processing time p_p of a packet p can be obtained by

$$p_p = M_{DP} - t_p - t_{P+SFD}, \quad (1)$$

where M_{DP} is the measured latency, t_p is the computed packet transmission time, and t_{P+SFD} is the computed transmission time of the Ethernet preamble and SFD (8 bytes) (Fig. 1b).

2) Results: While investigating a specific dimension, the other dimensions are kept constant with the default values in Tab. III. The boxplots in Fig. 2 show the measured processing times p_p for 100,000 packets per scenario. Note that, because of its lower processing time, a different scale is used for the Dell S4048-ON switch. We first consider only hardware rules and cover software rules later.

a) General: Fig. 2 shows that switches have similar processing times (around 4 μs), except the Dell S4048-ON (around 2 μs). Also, for a given case, the variance in processing time exhibited by the switches is always the same (around 0.2 μs for the Dell S4048-ON and 0.6 μs for the others).

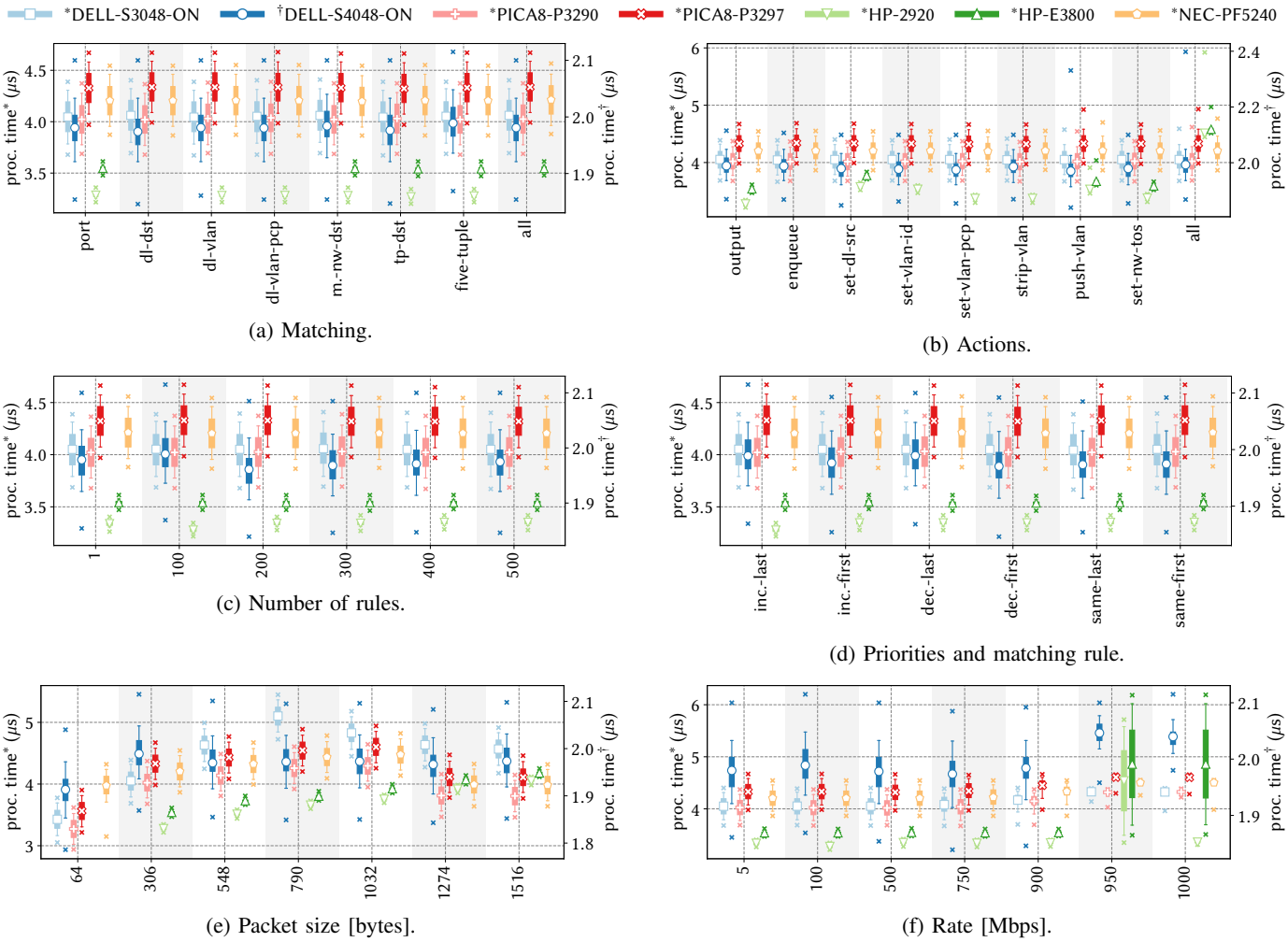
b) Matching: In Fig. 2a, it can be seen that the complexity of the matching structure does not affect the processing time of the switches. The HP 2920 does not support matching in hardware for *all* and *dl-dst*. Also, the HP E3800 does not support *dl-dst* in hardware, and interestingly, does not support *dl-vlan-pcp* and *dl-vlan* matchings at all[1].

c) Actions: Similar to the matching, the *action* types do not affect the processing time of the switches (see Fig. 2b). This behavior remains true, even when packet rewriting and checksum recomputations are involved. Again, we omit the unsupported actions (i.e., *strip-vlan*, *set-vlan-pcp* and *set-vlan-id* for the HP E3800 and *set-nw-src* and *set-tp-src* for all switches). For the *push-vlan* action case, though the processing time minimum, average and median values are the same as for the other actions, we observe that its maximum processing time is slightly higher. Similarly, this behavior is observed for the *all* action (as it includes *push-vlan*).

d) Number, Priority, and Order of Rules: Fig. 2c and 2d show that the number, priority and order of rules do not impact the switch processing time.

e) Packet Size: As it can be seen in Fig. 2e, the Pica8 P3290, Pica8 P3297, Dell S3048-ON, and NEC PF5240 switches show almost the same behavior: the switch processing time increases with the packet size up to a certain threshold (e.g., 790 bytes for the Dell S3048-ON), thereafter, it starts to decrease (e.g., from 1032–1516 for the Dell S3048-ON). This is surprising when compared to existing literature results, that additionally consider transmission latency [12] when measuring processing time. Due to missing insights on the exact details of the switch architecture and its used ASICs, we can only speculate on the reasons which looked most reasonable to us. As these switches mainly use Broadcom chips, we

[1]We note that the HP E3800 documentation states that *dl-vlan* and *dl-vlan-pcp* matchings are supported. However, with a configuration identical to the one used for HP 2920, the switch never successfully matched on these fields. We tried to contact the HP support, unfortunately, without any reply.



(a) Matching.

(b) Actions.

(c) Number of rules.

(d) Priorities and matching rule.

(e) Packet size [bytes].

(f) Rate [Mbps].

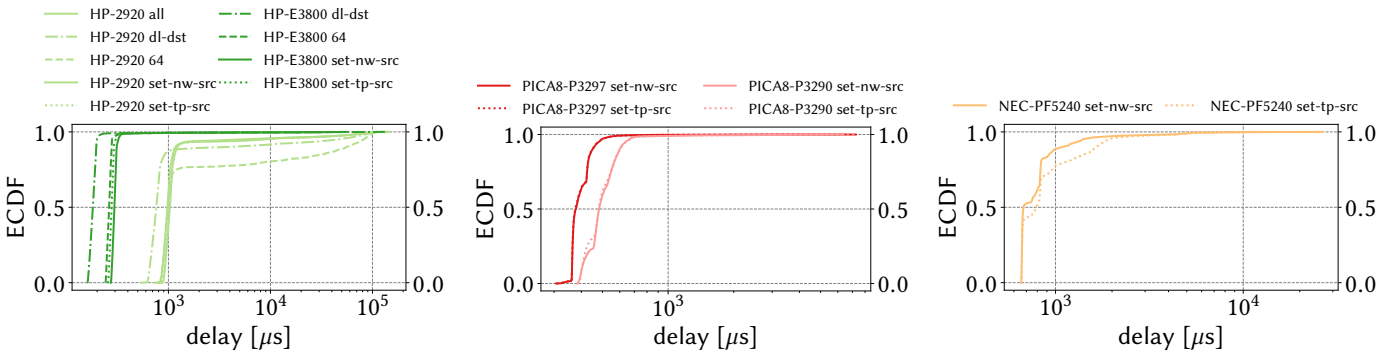
Fig. 2: Impact of the different considered dimensions on the hardware processing time of the different switches. The number of rules, their priority, the matching rule and its matching structure do not impact the processing time. The packet size is the main influential factor, while rate and action type impact only a subset of the investigated switches. Overall, the processing time of switches is predictable and can be deterministically bounded by a constant.

guess that this is indeed an ASIC dependent behavior[2]. Such a behavior can be due to the traffic manager implementation, since this ASIC module is usually responsible for buffering packets before sending them. Specifically, to achieve high utilization and efficiency, the traffic manager typically waits until a certain number of cells of data are filled to continue processing. While this can present some explanations for the increasing trend, the decreasing trend remains unclear. The two HP switches show a mostly (actually piecewise) linear behavior. We suspect this is because of the specific way bytes are buffered in the HPE ASIC. In contrast, the processing time of the Dell S4048-ON is mostly constant except for smaller packets, where it is smaller.

f) Rate: Fig. 2f indicates that the rate also does not influence processing time of the switches, except for very high rates. In these cases, both HP switches see their overall processing time increasing, while other switches only see their minimum processing time increased.

g) Software Rules: In this part, we present the results of measuring the processing time of software rules (excluding Dell, because they do not support any software processing capabilities). Because packet loss is observed for higher rates, we use a rate of 0.1 Mbps for this experiment. We point out three main observations (Fig. 3). First, none of the switches support L3/L4 rewriting in hardware. The HP switches additionally do not support L2 matching in hardware. More surprisingly, the HP switches cannot forward 64 bytes packets in hardware: even if the rule is stored in hardware and operates properly for bigger packets, we observed that 64 and 65 bytes packets are always processed in software. Interestingly, this does not happen when the switch runs in legacy mode or with OpenFlow rules matching on the physical input port. We suspect that this is due to the fact that the HP switches use a different TCAM table for OpenFlow processing with complex matching, which might not be able to process packets of these sizes. Second, software processing time is up to 5

[2]We have tried to contact Broadcom to obtain explanations on this behavior, however without any success.



(a) HP. (b) Pica8. (c) NEC.

Fig. 3: Processing time of software rules for different vendors. Note that Dell does not support any software rules. We observe that software processing is much less predictable than hardware processing. Software processing is also orders of magnitude slower.

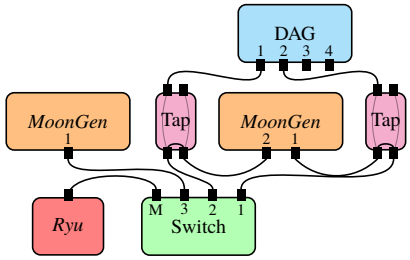


Fig. 4: Measurement setup for priority queuing investigation.

orders of magnitude higher than in hardware. Third, processing time in software is much less predictable: it varies by nearly two orders of magnitude comparing to the hardware case (see Fig. 2). This is due to the fact that the switch CPU is also performing other interfering tasks (e.g., running its OS and the OpenFlow agent).

3) Outcomes: The processing time of switches mostly depends on packet size. Moreover, the *action*, *matching*, *priorities*, and the *number* of rules do not influence processing time. The processing time of switches can be considered predictable and bounded only for hardware rules.

B. PQ — *Priority Queuing Overhead*

Predictable latency works consider the availability of priority queuing [4]–[7]. Queuing delay is then computed using mathematical models, e.g., network calculus [13], [14].

We explore whether our switches support priority queuing and if their behavior can be verified by network calculus models. Regarding the first question, we found that only the two HP switches do not support priority queuing in OpenFlow mode (i.e., the *enqueue* action).

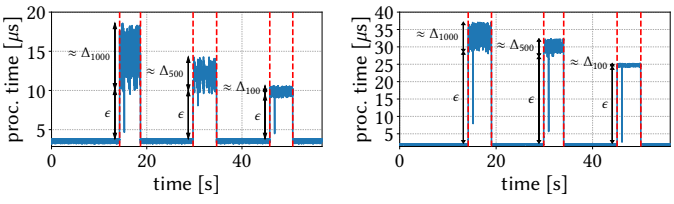
1) Measurement Setup: Two *MoonGen* instances send low and high priority flows to the switch through ports 1 and 3, respectively (Fig. 4). The flows are forwarded to port 2. We start a high priority flow sending bursts of 100 bytes packets. Then, in order to quantify the overhead of priority queuing, we subsequently send three low priority flows with different packet sizes (1000, 500, and 100 bytes) also at line rate. The switch processing time is then measured using the measurement card and a setup identical to Sec. II-A.

2) Network Calculus Prediction: Network calculus states that processing of a high priority (H) packet can be delayed by a lower priority (L) packet by at most the transmission time of the largest packet in lower priority queues (thereby considering the non-preemptive property of priority schedulers). Therefore, network calculus calculates the worst-case delay bound D_{H} experienced by a high priority flow as

$$D_{\mathrm{H}} = p_{\mathrm{p}} + \Delta_{l_{\mathrm{L}}} = p_{\mathrm{p}} + l_{\mathrm{L}}/R, \tag{2}$$

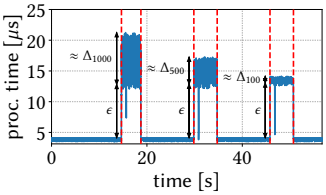
where p_{p} is the pure processing time of packet p (as measured in Sec. II-A), the parameter $\Delta_{l_{\mathrm{L}}}$ is the transmission time of the largest packet in lower priority queues, l_{L} is the size of this packet, and R is the link rate [13], [14]. In the following, we will show that surprisingly, in practice, processing time can be higher than this mathematical prediction.

3) Results: In the worst-case, we observe a total delay increase of $\Delta_{l_{\mathrm{L}}} + \epsilon$ (Fig. 5). That is, there is a delay increase of ϵ in addition to the increase predicted by network calculus, independent of the size of the interfering packets. In fact, ϵ is the overhead of the priority queuing implementation. The reason is that the switch is not able to determine the next queue without spending a minor processing overhead. We confirmed this observation by reducing the rate of the interfering flows. In this case (not shown in this paper due to lack of space), the maximum processing time of the high priority flow looks exactly the same but we observe all the intermediate values. This is because, when the rate is lower, it can happen that a high priority packet arrives just before the next check of the switch, hence not having to wait. In our plots, these values are not visible because cross-traffic is sent at line rate; hence, the scheduler always switches back to low priority flows. All switches exhibit this behavior. Further measurements, not shown here, show that the ϵ value depends on the switch model but stays constant for different scenarios (different high priority packet sizes, additional concurrent lower priority flows). From our measurements, we have 9 μs for both Pica8 switches and the NEC PF5240, 6 μs for the Dell S3048-ON, and 27 μs for the Dell S4048-ON. The relatively low overhead

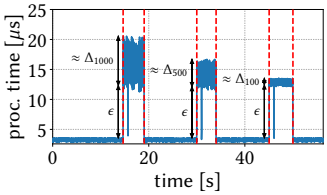


(a) Dell S3048-ON.

(b) Dell S4048-ON.



(c) NEC PF5240.



(d) Pica8 P3290.

Fig. 5: Processing time of a high priority flow (bursts of 100 bytes packets) with 3 subsequent interfering low priority flows of, respectively, 1000, 500 and 100 bytes packets. We observe that the processing time increases more (by ϵ) than what network calculus predicts. We note that the behavior of Pica8 3297 switch is very similar to Pica8 3290 and hence not shown.

and its stability tells us that the scheduling operation is not performed by the central CPU of the switch but by a specific component responsible for this, e.g., a micro-controller.

4) Outcomes: While the two HP switches do not support priority queuing, all the other switches implement priority queuing by updating the queue to send from only every ϵ seconds. This invalidates network calculus predictions (Eqn. 2). However, once the modeling correction is done, the performance of the switches is stable and predictable.

C. TP — Line Rate Throughput

Queuing models used by state-of-the-art approaches [4]–[7] assume that queuing does not happen at the ingress of switches but at their egress. Accordingly, switches need to be fast enough to process packets at line rate.

We verify that switches indeed can process packets at line rate in both directions on all their ports simultaneously without any loss. Interestingly, most existing work measure only the throughput of a single port of a switch [12], [15]–[19]; the complete saturation of a switch backplane has not yet been targeted in the literature.

1) Setup: The Shoelace Measurement: The traditional approach to measure throughput is quite simple: send a high load to a switch and measure the output load from it. In practice, however, to saturate the switch is a challenging task: a 48-port switch demands producing a rate of up to 48 Gbps, which, with a simple approach of connecting each port to a traffic source, would require 48 servers or networking cards. As a result, researchers then simply fallback to measuring the throughput of a single pair of ports [12], [15]–[19]. However, such a measurement does not verify that a switch is able to process packets at line rate if several ports are used simultaneously, hence not guaranteeing predictability.

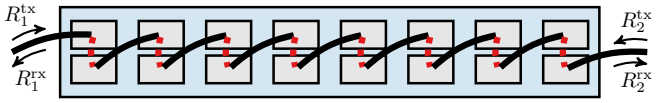


Fig. 6: The shoelace measurement setup for the measurement of the throughput of a programmable switch. Black thick lines represent cables and red thick dashed lines represent internal forwarding rules. This setup allows to saturate all the ports of a switch with only two traffic sources.

To circumvent this issue, we propose the *shoelace measurement*[3] setup (Fig. 6), a methodology that makes use of the programmability of switches to saturate a switch with only *two traffic sources* — only two physical network connections instead of, e.g., 48, are needed.

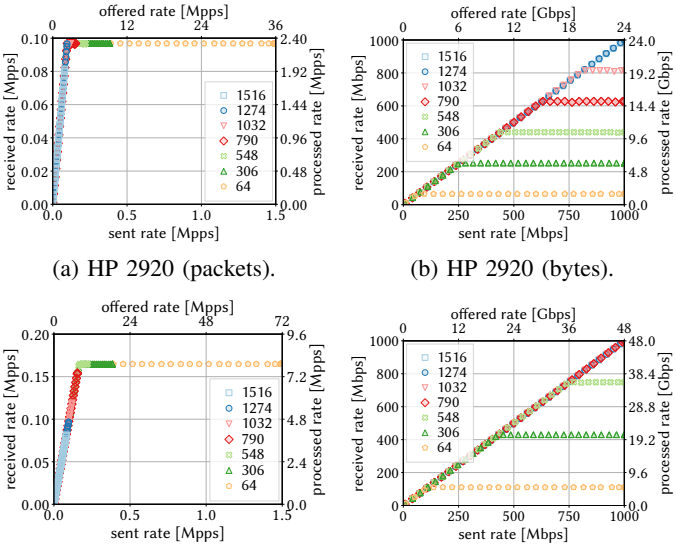
The first source saturates the first port of the switch. The switch is configured to forward all traffic entering this first port to its second one. The second port of the switch is then connected back to its third port. This port configuration and connection then goes on until it reaches the last port of the switch. In this way, all the traffic sent to the first port has to be processed n times, where n is the number of ports of the switch. Further, by having the second source sending traffic to the last port of the switch, and configuring backward forwarding rules, we effectively saturate the n ports of the switch in both directions. We then can compute the (minimum) throughput of the switch as $n \times (R_1^{rx} + R_2^{rx})$, where R_1^{rx} and R_2^{rx} are the rates received at the first and last ports, respectively. If these values are equal to R_2^{tx} and R_1^{tx}, we could not reach the throughput limit of the switch and the switch is able to process bi-directional line rate on all its ports simultaneously. For a given input rate $R_1^{tx} = R_2^{tx}$, we run the experiment five seconds and consider that a switch is able to handle a given rate only if no single packet is lost.

We use the shoelace measurement setup to investigate the throughput of all our switches for all the packet sizes in Tab. III. Using *MoonGen* [10], we generate line rate traffic on both the first and last port of the switch. We use the statistics of the two interfaces to detect packet loss. We consider rates from 8 to 1000 Mbps by steps of 32 Mbps. For the Dell S4048-ON, we also run the experiment with 10 Gbps links, ranging from 80 Mbps to 10 Gbps by steps of 320 Mbps.

2) Results: Among all the switches, only the HP switches lost packets. For the smallest packet size (64 bytes), sending 1 Gbps (resp. 10 Gbps) corresponds to 1.5 Mpps (resp. 15 Mpps). That is, for our 48-port switches all but the HP switches can process at least 48 $\times$ 1.5 = 72 Mpps (resp. 720 Mpps) simultaneously. This is confirmed by the datasheets of the switches, which all announce values higher than this.

For the HP switches, the datasheets also announce that the switches can process line rate on all their ports. However, this is not the case (see Fig. 7). We observe that, independently of the packet size, the switches can process up to 2.32 Mpps and 7.91 Mpps. Depending on the packet size, this leads to different data rates. The HP 2920 can process line rate for

[3]Taking its name from how the cabling of the switch looks like in this setup.



(a) HP 2920 (packets). (b) HP 2920 (bytes).

(c) HP E3800 (packets). (d) HP E3800 (bytes).

Fig. 7: Throughput of the HP switches. Lower (resp. left) axes correspond to the traffic sent (resp. received) on the first and last ports, i.e., R_1^{tx} and R_2^{tx} (resp. R_1^{rx} and R_2^{rx}). The upper (resp. right) axes correspond to the values scaled by the number n of ports to represent the total data rate actually processed by the switch.

packets of at least 1274 bytes and the HP 3800 for packets of at least 790 bytes. This stands in contradiction to the datasheets of the switches. In order to investigate this further, we conduct a measurement run with the HP E3800 in legacy mode. As the shoelace measurement setup cannot be used for legacy switches, we only measure the throughput on two ports. We use the 10G SFP+ ports to be able to reach more than 7.91 Mpps. In such a setup, the HP E3800 switch processes correctly 10 Gbps at line rate, i.e., from 0.8 Mpps to 14.8 Mpps depending on the packet size. Using a bidirectional measurement, the switch is also able to process all packets, but the network card (Intel 82599ES) used was only able to generate up to 9.2 $\times$ 2 = 18.4 Mpps for 64 bytes packets. We conclude that the throughput that the switch can handle depends on whether it operates in OpenFlow or legacy mode.

Our explanation here is as follows: the L2/L3 TCAM tables used for legacy switching/routing cannot store arbitrary matching fields, which means they cannot support the OpenFlow features. To support OpenFlow, the HP switches use their traditional so-called "ACL" TCAM tables, which provide higher flexibility. Hence, in OpenFlow mode, we measure the throughput of the "ACL" table, while in legacy mode we measure the throughput of the L2/L3 tables. It turns out that the "ACL" table of the switches was not dimensioned for handling line rate, which hence impacts the performance of the switch when used in the OpenFlow mode[4].

3) Outcomes: Using our proposed shoelace setup, we observed that only the HP switches do not behave as expected and are hence not predictable, as they occasionally can lose packets. They cannot process packets at line rate (Tab. IV), even though they can in legacy mode: the TCAM table used for OpenFlow and legacy modes are different and exhibit different throughput.

[4]Note that, here, 64 byte packets are still processed in hardware, while they were processed in software in Sec. II-A. This is because we are here matching on physical port while we used *five-tuple* matching in Sec. II-A.

TABLE IV: Measured throughput for the different switches. Values in bps are given for the smallest and biggest packets. *line rate* means that the switch handles line rate on all its ports simultaneously.
*measured at 10 Gbps.

Switch	**[pps]**	**[bps]** (64 – 1516 bytes)
HP E3800	7.91 Mpps	5.31 Gbps – *line rate*
HP 2920	2.32 Mpps	1.56 Gbps – *line rate*
Dell S3048-ON	$\geq$ 72 Mpps	*line rate* – *line rate*
*Dell S4048-ON**	$\geq$ 720 Mpps	*line rate* – *line rate*
Pica8 P3290	$\geq$ 72 Mpps	*line rate* – *line rate*
Pica8 P3297	$\geq$ 72 Mpps	*line rate* – *line rate*
NEC PF5240	$\geq$ 72 Mpps	*line rate* – *line rate*

III. Management Predictability

Next, we consider the management predictability. We analyze two aspects which are relevant for state-of-the-art approaches concerning predictable latency: the flow management (FM), in Sec. III-A, and the buffer management (BM), in Sec. III-B, of switches.

A. FM — Flow Management

State-of-the-art approaches rely on fine-grained traffic engineering (one rule per single flow) in order to provide their strict guarantees, e.g., deterministic latency [6], [7].

Therefore, the number of flow rules on a single switch in a network can grow up to several thousands of flows [7]. Hence, the first requirement is a sufficient flow table capacity. As flow requests arrive during runtime, they have to be inserted live in the corresponding hardware tables by the controller. Hence, each switch should have synchronized data and control planes, e.g., it should not state that certain flow rules are embedded if they are actually not. Note that it is known that adding a rule into the hardware table of a switch can cause a wide variety of issues [20], [21]. For instance, the state of the data plane can lag behind the state of the control plane for a certain amount of time [21]. However, we here only require the switch to add the rule in its hardware table *at some moment*, and we do not consider delay as an issue.

Note that in this section, the OpenFlow version in use may have an impact. We, however, stick to OpenFlow 1.0 as it provides the necessary features for the state-of-the-art end-to-end latency models and it is fully supported by all switches.

1) Measurement Setup: We connect the target switch to a *Ryu*-based controller and connect a dual-port data plane host running *MoonGen* [10] to the switch. First, we install rules at a given rate on the switch until the latter returns an OpenFlow *Error* message indicating that its flow table is full. We consider the same match and action parameters as listed in Tab. III. The rules' *output* actions direct to the second *MoonGen* interface. Second, the controller queries the

state of the switch with *TableStatsRequest*/*FlowStatsRequest* messages. Finally, the *MoonGen* host generates one packet per each rule on its first interface and checks if it is received on the second interface, i.e., if it is correctly forwarded.

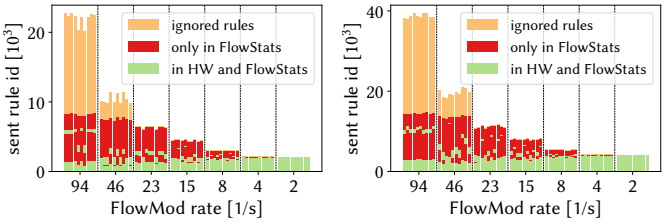
2) Results: Generally, the results per switch vary; hence, we report on each manufacturer separately.

a) Pica8 switches: We consider *five-tuple*/*output* flow rules. The following behavior is the same for other combinations of matching and actions. Fig. 8 shows the total number of sent rules before receiving the first *Error* message for different *FlowModAdd* rates and different runs per rate (one bar corresponds to one run). The different colors identify rules that were correctly added in the hardware table, those that were only in the stats reply of the switch (and hence are reported to be added but are actually not) and those that were simply ignored. The total number of rules in hardware is always the same: the Pica8 P3290 stores 2046 rules and the P3297 stores 4094. However, for increasing *FlowModAdd* rates, we observe that the number of ignored and incorrectly added rules increase. While ignored rules are not too problematic, as the controller can react to it, rules that are reported to be added but are not are critical: the controller assumes that packets will be forwarded, while they will not be.

Although both switches can store enough rules, they fail to report a correct state to the controller, which can be dramatic for predictability: packets of a theoretically accepted flow can never be forwarded. We believe that the reason for this is due to synchronization issues between the Open vSwich (OVS) [22] instance, which realizes OpenFlow on both switches, and the ASIC managing the hardware table. OVS might not be fast enough to insert all the rules, although it previously confirmed them to the controller. This leads to an inconsistency between OVS and what is actually inserted into the hardware table. It is important here to make sense of two aspects. First, the considered rates: inconsistencies appear already for as low as four new flows per second. Second, the correctly installed rules are not the first ones and vary depending on the run. That means that the switches show here an completely unpredictable behavior: we cannot know in advance which rules will be correctly added, except if we reduce the addition rate to a single flow per second, which is not feasible. We believe that such observations are not only relevant for operators, but also researchers when experimenting with these switches.

b) HP switches: Fig. 10a shows the hardware table size of the HP E3800 switch for the different match/action combinations. A size of zero (white) indicates that a certain match and action combination is not supported. Depending on the match and action type of the embedded rules, the table size varies from 372 to 4085. Due to space reasons, we omit showing the detailed results for the HP 2920; however, it exhibits a similar trend as the HP E3800 but supports only around 100 to 500 flows.

We additionally observed a remarkable behavior: the switches showed aging effects during our measurements. Fig. 9a shows the total number of rules in the *FlowStats*



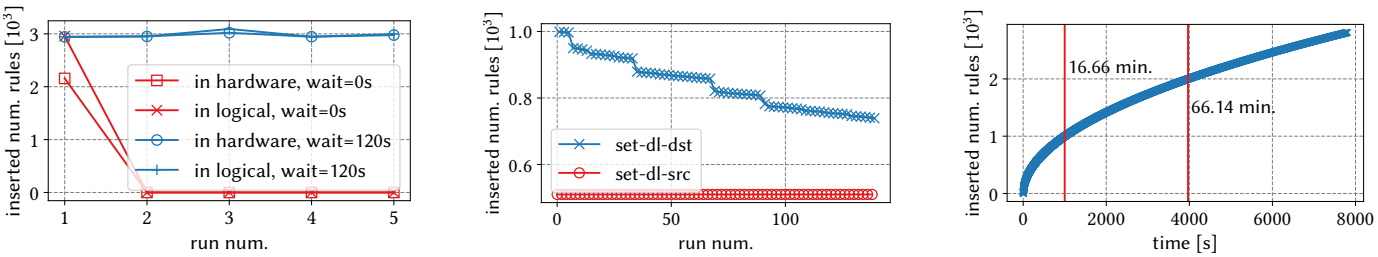
(a) Pica8 P3290. (b) Pica8 P3297.

Fig. 8: Divergence of the flow table state of the Pica8 (a) P3290 and (b) P3297 switches for different *FlowModAdd* rates. For each rate, each stacked bar corresponds to a distinct run. Rules that are only in the *FlowStats* response of the switch are critical: while the controller thinks the packets will be forwarded, they will actually not be. The switch is then unpredictable.

answer of the HP E3800 switch and its hardware table over five consecutive runs with 85 *FlowModAdd* messages per second (*five-tuple*/*output* flow rules). For the first set (red lines) of runs, the data plane test is done directly after receiving the *FlowStats* response. For the second set (blue lines), we introduce an additional waiting time of 120 seconds before the start of the date plane test. We notice that sending the *FlowModAdd* messages with a high rate triggers the switch to send the *Error* message earlier, i.e., before the hardware table is actually full. Indeed, the table size for *five-tuple*/*output* with lower rate was 4085 (see Fig. 10a) while it is now around 3000 (run 1 in Fig. 9a). Furthermore, we observe that the amount of forwarded packets is lower (around 2100 packets) than the number of rules in the logical table. Doing other consecutive runs without waiting, we observe that the switch then rejects any new rule. On the other hand, we observe that, if we wait 120 seconds before the data plane tests, the switch does not show some aging effects (blue lines). This aging leads to an unpredictable behavior from the switch: the controller can never be sure whether it is able to use the complete hardware table space.

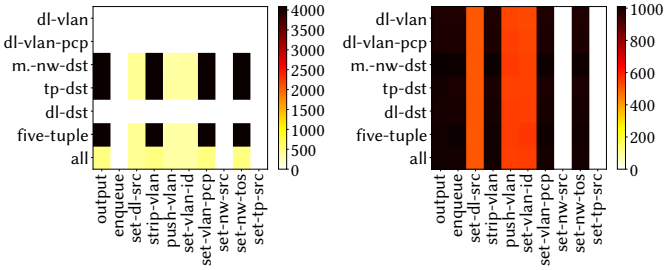
c) Dell switches: Similar to the HP switches, the number of rules which can be stored in the hardware flow table of the Dell switches varies based on the *match*/*action* combination: from 510 rules to the maximum of 1000 rules. Fig. 10b shows this for the Dell S4048-ON, the Dell S3048-ON behaving exactly the same. While the Dell devices are able to handle higher *FlowModAdd* rates (e.g., more than 85 *FlowModAdd* messages per second) than the other switches, their hardware tables are the smallest.

However, we noticed again flow table aging effects (shown for the Dell S3048-ON in Fig. 9b). We perform 140 consecutive runs measuring the available flow table size. All even runs have *five-tuple* matchings and *set-dl-src* actions, while the odd runs have *five-tuple* matchings and *set-dl-dst* actions. We observe that, for each new iteration, the number of flows that can be added with *five-tuple*/*set-dl-src* combinations stays the same, while it reduces for the runs with *five-tuple*/*set-dl-dst* combinations. The reduction is non-negligible, as the capacity of the flow table reduces from 1000 to 739 rules.



(a) HP E3800. (b) Dell S3048-ON. (c) NEC PF5240.

Fig. 9: (a) HP E3800 switch, *five-tuple-output* as the match-action, with rate of 85 messages per second. (b) Aging of the Dell switches. The measurement procedure is as follows, we use *five-tuple* matching and we perform 200 runs, 100 runs with *set-dl-src* and 100 runs with *set-dl-dst* actions. The runs are performed consecutively, first one runs with *set-dl-src* and then with *set-dl-dst*. We notice that the maximum number of rules for *set-dl-dst* reduces at each iteration. (c) The addition of rules to the NEC PF5240 switch takes a big amount of time.



(a) HP E3800. (b) Dell S4048-ON.

Fig. 10: Flow table size of the (a) HP E3800 and (b) Dell S4048-ON switches for the different match/action combinations. A size of 0 identifies a combination not supported in hardware. We observe that the table size depends on both the matching type and the action and can heavily influence the number of rules that can be inserted.

d) NEC PF5240: The NEC PF5240 switch is the only switch that controls the control plane message rate by throttling the TCP connection. As we cannot control the TCP behavior with Ryu, we modify our procedure. Instead of sending *FlowModAdd* messages with a certain rate, we now send a *BarrierRequest* message after each *FlowModAdd* and wait for the response from the switch. Upon the reception of the *BarrierResponse*, a new *FlowModAdd* message is dispatched after 10 milliseconds. Fig. 9c shows the number of inserted rules by the switch during one measurement run. By using the modified procedure, adding the rules to NEC PF5240 switch takes a very large amount of time. For instance, adding 1000 rules takes around 16 minutes, and adding 2000 rules takes more than one hour. In total, the switch accepts 2809 rules. Although the size of the table is acceptable, the rules are not inserted in a *timely* manner.

3) Outcomes: Unfortunately, based on our analysis, none of the switches exhibit a predictable behavior with a sufficiently sized hardware flow table. Both Dell and HP switches suffer from unpredictable aging, i.e., the flow table size can reduce with each consecutive run. The Pica8 switches can unpredictably and silently ignore rules. For the NEC switch, reaching a high number of rules requires too much time.

B. BM — *Buffer Management*

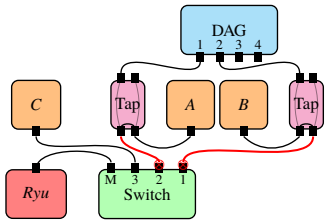
In order to ensure no packet loss, state-of-the-art approaches for predictable latency rely on mathematical models, e.g., network calculus, to bound the amount of backlog flows generate at each individual queue on their way to their destination [4]–[7]. To this end, all these approaches assume that each queue of each port of each switch is equipped with its own physical buffer, and that these buffers are managed independently. To which extent this assumption is true is still an open question. More precisely, there are several challenges that are still open to be tackled: *i)* how do existing switches actually manage their buffers? and *ii)* are these buffers actually isolated? In this part, we intend to study these questions in detail by measuring the buffer capacity of a particular queue in different overload scenarios and assess whether the way the switches manage their buffer is predictable, and, more importantly, as predicted by state-of-the-art models.

1) Measurement Setup: We have already seen that the output of switches is not always trustworthy (Sec. III-A). Therefore, we design a setup relying only on data plane measurements to measure buffer capacities. Similar to [12], we infer the buffer size N of a particular queue based on observed packet delays. The total delay observed (through a measurement setup identical to Fig. 1a) by a packet from a high priority queue is given by

$$D_{\mathrm{H}} = p_{\mathrm{p}} + \epsilon + q_{\mathrm{p}}, \tag{3}$$

where, in addition to the processing time p_{p} (measured in Sec. II-A), ϵ corresponds to the priority queuing overhead (measured in Sec. II-B), and q_{p} corresponds to queuing delay. We note that both ϵ and q_{p} parameters were not present in Sec. II-A. This is because these parameters are larger than zero only when queuing happens, which we made sure is not the case for our measurements in Sec. II-A. The values of p_{p} and ϵ in Eqn. 3 are known from previous sections. The queuing delay q_{p} can then be obtained by measuring the total delay D_{H} and used to calculate N. Indeed, the queuing delay q_{p} of a given high priority packet can be decomposed as

$$q_{\mathrm{p}} = \sum_{i \in P} l_{\mathrm{i}}/R, \tag{4}$$



(a) Physical setup.

From	To	Priority
A	B	High
B	A	High
C	B	High
C	A	High
C	B	Low
C	A	Low

(b) Flows sent.

Fig. 11: Setup for evaluating buffer size with 2 congested ports and 2 congested queues per port. The congested ports are marked with a red cross.

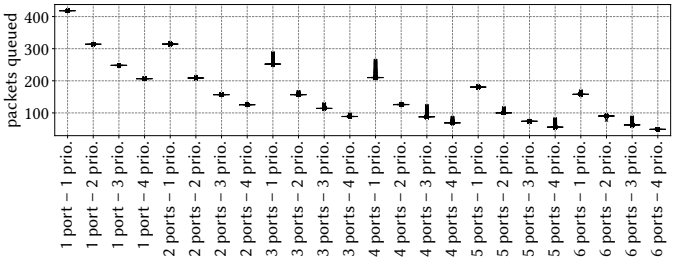
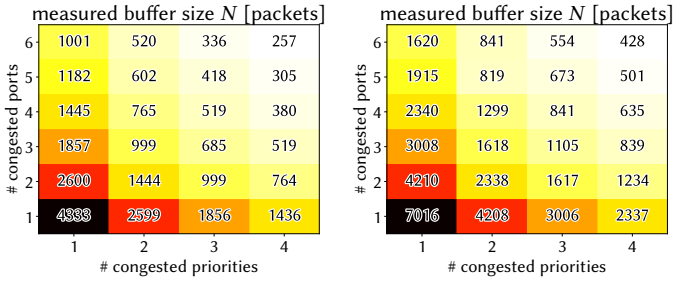


Fig. 12: Buffer capacity made available to a given queue of the Dell S3048-ON switch for different numbers of congested ports and queues (1516 bytes packets): the more ports and queues are overloaded, the less buffer is made available to a given queue.



(a) 1516 bytes packets.

(b) 790 bytes packets.

Fig. 13: Median buffer capacity made available to a given queue of the Dell S4048-ON switch for different numbers of congested ports and queues for different packet sizes. The Dell S4048-ON can store more packets than the Dell S3048-ON and the number of packets that can be buffered, as expected, increases with smaller packets.

where P is the set of packets scheduled before the considered packet and R is the link rate. If all packets have the same size l, we have

$$q_{\text{p}} = |P|l/R, \tag{5}$$

where $|P|$ is the number of packets in the high priority queue when the considered packet arrived. If a port or queue is overloaded[5], it will start queuing packets and eventually drop some of them. When we observe a packet loss, we can compute the queuing delay d_{p} of the previous non-dropped packet, from which we can obtain $|P|$. As this packet was the last one to be buffered before dropping packets, we have $N = |P| + 1$, i.e., the buffer capacity corresponds to the number of packets queued before p plus one for p itself. This measurement procedure allows us to monitor only the buffer size available to high priority queues.

We use up to six different ports sending flows to each other at line rate. Each port sends and receives one flow. These flows are forwarded to the highest priority queues. We then use an additional port to send "overload" traffic. For each sending port, the overload port sends additional packets with the same headers, hence overloading the corresponding receiving queue. In order to overload low priority queues, the overload port simply sends additional flows which are forwarded to the corresponding low priority queues. This is sufficient to overload them, as they will never be served, since we send line rate of high priority flows at the same time. We use this setup to overload from 1 to 6 ports and from 1 to 4 or 8 priority queues per port (depending on how many queues the switch under test supports). An example setup to congest 2 ports and 2 queues per port is shown in Fig. 11. We monitor packet loss and packet delays of one of the high priority flows using our measurement card. We configure forwarding rules on the switch matching on IP destination and enqueuing in a specific queue. Further, we use different source MAC addresses for each packet to uniquely identify them and easily detect packet loss and compute q_{p}.

2) Measurement Results: We detail our results for the different manufacturers separately.

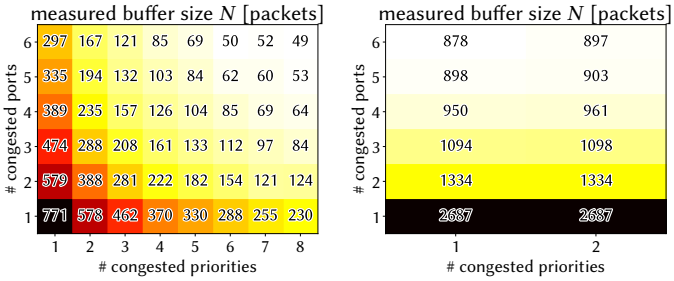
a) HP switches: As the two HP switches do not support priority queuing (no separate queues), this measurement does not apply to them.

[5]*overloaded* and *congested* terms are used interchangeably.

b) Dell switches: The Dell S3048-ON and S4048-ON switches support up to 4 priority queues. Fig. 12 shows the inferred buffer sizes over runs of at least three seconds with 1516 bytes packets with the Dell S3048-ON switch, for different number of congested ports and priorities. In contrast to what other related works such as QJump [5], Silo [4] and DetServ [7] assume, we observe that the buffer made available to our monitored flow depends on the buffer needed by other ports and queues. More specifically, the more ports and queues are congested, the less buffer is made available to our monitored flow. We note that the available buffer size ranges from around 420 packets to around 50 packets.

Fig. 12 shows that for each run, the inferred buffer size (one value per packet lost) is stable. Hence, in Fig. 13, we plot heatmaps of the observed median values for the Dell S4048-ON switch. The results indicate that the available buffer for the S3048-ON switch is bigger than for the S4048-ON switch: from around 10 times bigger for a single congested queue to around 5 times bigger for 6 ports with 4 congested queues. Fig. 13b shows that, as expected, reducing the size of transferred packets allows to increase the total number of packets in the queue. The overall behavior however stays the same.

c) Pica8 switches: Both Pica8 switches presented the exact same behavior, we here show only results for the Pica8 P3297 switch. Fig. 14a shows that the behavior is comparable to the Dell switches: the more queues and ports are congested,



(a) Pica8 P3297 (the P3290 behaves exactly the same).

(b) NEC PF5240.

Fig. 14: Pica8 and NEC behave similarly to the Dell switches: buffers are not isolated per-queue (1516 bytes packets).

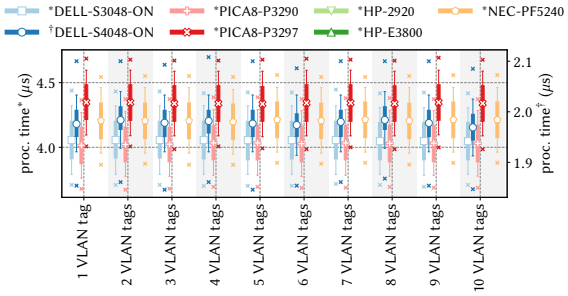


Fig. 15: Processing time of the different switches for matching on the outermost VLAN tag, popping it and forwarding to a particular queue based on different VLAN tag stacks heights. We observe that the switches can predictably support source routing solutions based on VLAN tags.

the less buffer becomes available to the monitored queue. Interestingly, our results invalidate the Pica8 documentation regarding buffer management. Indeed, while they indeed say the queues are based on a shared buffer, the numbers they provide do not correspond to our results.

d) NEC PF5240: The NEC switch supports up to 8 priority queues. The same experiment with the NEC PF5240 switch exhibits a constant buffer size for each scenario: 63 packets. Hence, the NEC switch seems to have isolated buffers for each queue. However, the NEC switch provides a global *limit-queue-length* option, 64 by default, that limits the maximum packets a queue can buffer. We set this option to its maximum value and rerun the experiment. Unfortunately, this option applies only to the two lower priority queues. As our setup can only measure the buffer size available for the highest priority, we reduce the maximum number of priority levels that can be congested to 2, the two lowest priority queues. Fig. 14b shows that the buffer size made available to a queue then depends on the number of congested ports. On the contrary to the Dell and Pica8 switches, it does not seem to depend on the number of congested priorities. Note however that the results here are much less stable than those from Dell in Fig. 12. Hence, the numbers should not be taken as exact, but as giving insights on the buffer management strategy of the switch.

3) Outcomes: We observe that, while predictable latency solutions do assume that switches provide isolated per-buffer queues, the reality is the opposite: all our switches are based on a shared memory for implementing queue buffers. This does actually make sense: sharing buffers among all queues and ports enable work-conservation: if a queue does not use its space, it can be used by another one. However, from a predictability point of view, there is a major issue: a burst in a low priority queue, or even on another port, can suddenly reduce the buffer space available to a given queue and hence potentially lead to unpredicted packet loss.

IV. Insights and Discussions

Let us recap some of our insights on the predictability of SDN switches so far. On the bright side, the observed processing times of the switches, at least when done in hardware, is fairly predictable. We also observed that most switches successfully process packets at line rate.

On the dark side, we identified several challenges. First, we found that priority queuing adds non-negligible overhead to processing time. While this is not what is predicted by state-of-the-art models (e.g., network calculus), we observe that this overhead is actually predictable: the observed ϵ values, which are static values for each switch, can simply be added as constants to the modeling formulas of network calculus. Second, we observed that switches have some very unpredictable behavior regarding their flow management. For example, the Pica8 switches can unpredictably drop rules without leaving any means to the controller to be aware of it. Would there be a way to avoid sending new rules to the switches at runtime? That would prevent such unpredictable artifacts to happen.

A potential solution is to use source routing, e.g., based on stack of VLAN tags. In this way, the forwarding table of the switches has to be programmed only once and embedding additional flows does not require to interact with the flow table of switches but only requires to implement the tagging on the source end host. As a small prototype, Fig. 15 shows that all our switches (but the HP switches) indeed support such as setup: forwarding to a particular queue based on the outermost VLAN tag and pop it.

Finally, we observe that the switches are based on a shared buffer infrastructure, in opposition to what traditional latency models assume [4]–[7]. We indeed observe that a given queue gets a different buffer capacity based on the current congestion state of the switch. While it may seem like that the buffer capacity made available to a queue is unpredictable, thereby making packet loss prevention complicated, our measurements give some hope. Indeed, we observe that the buffer capacity of a port, though variable, depends only on the number of congested ports and queues. Usually, one does not use all the ports of a switch: for example, in a $k = 4$ fat-tree topology, one only needs 4 ports. We can then use the values resulting from our measurements as worst-case buffer capacity, instead of the very pessimistic strategy of dividing the total buffer memory by the total number of queues (number of priority queues times the number of ports).

Generally, while our results refer to the investigated specific SDN switches, we expect that many of our results to

also be valid for P4 programmable devices. Indeed, in both cases, forwarding is done by TCAM tables and a firmware (independent of the P4 code) which implements buffer and flow management. The need for predictability for P4 devices might, however, be even more stringent, as a P4 programmer expects to have full control over its device, and hence expects a strictly predictable behavior based on its implemented logic.

V. Related Work

Our work builds on a rich literature on switch performance measurements. From the management and control plane point of view, studies in the recent years [20], [21], [23]–[29] have already shown that the states of control and data plane of certain switches can diverge. For instance, inserting a rule is not atomic, i.e., it might still take time for a rule to be inserted in hardware, even after having received a confirmation of the insertion from the switch. Other studies have shown that the flow table capacity of switches varies drastically among vendors [12]. Although we also cover similar flow management aspects, we provide new insights with a focus on predictability. For instance we show that besides not being atomic, rule insertion can even be ignored by some switches, thereby leaving the data plane configuration permanently inconsistent with the control plane. We further show that certain switches exhibit aging effects reducing their table size over time and thereby making it unpredictable. Similarly, while some studies have measured switch buffer sizes [12] or revealed the importance of buffer management strategies for latency-sensitive applications, our work sheds light on how switches actually manage their buffer: most architectures are based on a shared buffer dynamically allocated to queues or ports.

Numerous works have also provided insights into the data plane performance of programmable switches [12], [15]–[17], [19], [27], [30]–[33]. For instance, [12], [27], [33] revealed important latency, throughput and buffer size metrics in particular scenarios. Our work focuses more on predictability by investigating the same metrics but evaluating them in variable scenarios. Loko [19] also focuses on predictability but derives a completely new model for a low-cost switch for which the state-of-the-art models investigated here are not valid [4]–[7]. Software implementations have also been investigated [26], [34]–[37]. However, as suggested by these works, our measurements confirm that software processing using OS-based CPUs is not a viable solution for predictable performance.

Regarding priority queuing, Durner et al. [38] conducted an interesting measurement study on its impact on network performance, however, with a focus on flow-level aspects while our analysis focuses on per-packet delays for assessing the predictability of priority schedulers with respect to the latency of individual data plane packets. Some of our presented results also show that previous studies [12] contain even incorrect data, mostly due to device misconfiguration.

VI. Conclusion

This paper was motivated by the increasingly stringent dependability requirements of communication networks and the observation that a predictable network behavior critically depends on the underlying hardware. We presented a methodology and reported on our measurement study using different switches from different vendors, and identified several shortcomings, in terms of performance but also in terms of correctness.

We understand our work as a first step and believe that it opens several interesting avenues for future research. In particular, we only initiated the discussion of possible solutions, and more research is needed on how to design and model network components toward more predictable and deterministic network architectures meeting the requirements of future applications. We additionally hope that our results can serve as a motivation for manufacturers to avoid such unpredictable behaviors, especially for P4 devices, where the programmer expects to have complete control over its device.

Acknowledgment

This work has been performed in part in the framework of the EU project FlexNets funded by the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement No 647158 - FlexNets), and in part in the DFG ModaNet Project (grant No KE 1863/8-1). This work reflects only the authors' view and the funding agency is not responsible for any use that may be made of the information it contains.

References

[1] M. Shafi, A. F. Molisch, P. J. Smith, T. Haustein, P. Zhu, P. De Silva, F. Tufvesson, A. Benjebbour, and G. Wunder, "5G: A tutorial overview of standards, trials, challenges, deployment, and practice," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 6, pp. 1201–1221, 2017.

[2] ITU-R: Radiocommunication Sector of ITU, "IMT vision–framework and overall objectives of the future development of IMT for 2020 and beyond," *Rec. ITU-R M.2083-0*, 2015.

[3] W. Kellerer, P. Kalmbach, A. Blenk, A. Basta, M. Reisslein, and S. Schmid, "Adaptable and data-driven softwarized networks: Review, opportunities, and challenges," *Proceedings of the IEEE*, vol. 107, no. 4, pp. 711–731, 2019.

[4] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable message latency in the cloud," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 435–448, 2015.

[5] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues don't matter when you can jump them!" in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015, pp. 1–14.

[6] A. L. King, S. Chen, and I. Lee, "The middleware assurance substrate: Enabling strong real-time guarantees in open systems with openflow," in *Proceedings of the 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE, 2014, pp. 133–140.

[7] J. W. Guck, A. Van Bemten, and W. Kellerer, "DetServ: Network models for real-time QoS provisioning in SDN-based industrial environments," in *IEEE Transactions on Network and Service Management (TNSM)*, vol. 14, no. 4. IEEE, 2017, pp. 1003–1017.

[8] "Source code, configuration files and data sets associated to this paper." https://sdn-predictability.lkn.ei.tum.de, 2019.

[9] Ryu, SDN, "Framework community: Ryu SDN framework," http://osrg.github.io/ryu, 2015.

[10] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proceedings of the 2015 Internet Measurement Conference*. ACM, 2015, pp. 275–287.

[11] S. F. Donnelly, "High precision timing in passive measurements of data networks," *PhD thesis, University of Waikato*, 2002.

[12] S. Bauer, D. Raumer, P. Emmerich, and G. Carle, "Behind the scenes: what device benchmarks can tell us," in *Proceedings of the Applied Networking Research Workshop (ANRW)*. ACM, 2018, pp. 58–65.

[13] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, April 2012.

[14] A. Van Bemten and W. Kellerer, "Network calculus: A comprehensive guide," *Technical University of Munich, Chair of Communication Networks, Technical Report No. 201603*, October 2016.

[15] G. Pongrácz, L. Molnár, and Z. L. Kis, "Removing roadblocks from SDN: Openflow software switch performance on Intel DPDK," in *Proceedings of the Second European Workshop on Software Defined Networks (EWSDN)*. IEEE, 2013, pp. 62–67.

[16] A. Bianco, R. Birke, L. Giraudo, and M. Palacin, "Openflow switching: Data plane performance," in *Proceedings of the 2010 IEEE International Conference on Communications (ICC)*. IEEE, 2010, pp. 1–5.

[17] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "Performance characteristics of virtual switching," in *Proceedings of the 3rd International Conference on Cloud Networking (CloudNet)*. IEEE, 2014, pp. 120–125.

[18] D. Raumer, S. Gallenmüller, F. Wohlfart, P. Emmerich, P. Werneck, and G. Carle, "Revisiting benchmarking methodology for interconnect devices," in *Proceedings of the 2016 Applied Networking Research Workshop (ANRW)*. ACM, 2016, pp. 55–61.

[19] A. Van Bemten, N. Đerić, J. Zerwas, A. Blenk, S. Schmid, and W. Kellerer, "Loko: Predictable latency in small networks," in *Proceedings of the 15th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. ACM, 2019, to appear.

[20] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about sdn control and data planes," Tech. Rep., 2014.

[21] ——, "What you need to know about sdn flow tables," in *Proceedings of the International Conference on Passive and Active Network Measurement*. Springer, 2015, pp. 347–359.

[22] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The design and implementation of open vswitch," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015, pp. 117–130.

[23] M. Kuźniar, P. Perešíni, D. Kostić, and M. Canini, "Methodology, measurement and analysis of flow table update characteristics in hardware openflow switches," *Computer Networks*, vol. 136, pp. 22–36, 2018.

[24] K. He, J. Khalid, S. Das, A. Gember-Jacobson, C. Prakash, A. Akella, L. E. Li, and M. Thottan, "Latency in software defined networks: Measurements and mitigation techniques," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 1. ACM, 2015, pp. 435–436.

[25] Z. Bozakov and A. Rizk, "Taming SDN controllers in heterogeneous hardware environments," in *Proceedings of the Second European Workshop on Software Defined Networks (EWSDN)*. IEEE, 2013, pp. 50–55.

[26] A. Lazaris, D. Tahara, X. Huang, E. Li, A. Voellmy, Y. R. Yang, and M. Yu, "Tango: Simplifying SDN control with automatic switch property inference, abstraction, and optimization," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. ACM, 2014, pp. 199–212.

[27] D. Y. Huang, K. Yocum, and A. C. Snoeren, "High-fidelity switch models for software-defined network emulation," in *Proceedings of the 2nd ACM SIGCOMM workshop on Hot Topics in Software Defined Networking*. ACM, 2013, pp. 43–48.

[28] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "Oflops: An open framework for openflow switch evaluation," in *Proceedings of the International Conference on Passive and Active Network Measurement*. Springer, 2012, pp. 85–95.

[29] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. E. Li, and M. Thottan, "Measuring control plane latency in sdn-enabled switches," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*. ACM, 2015, pp. 25:1–25:6.

[30] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, "Modeling and performance evaluation of an openflow architecture," in *Proceedings of the 23rd International Teletraffic Congress*. International Teletraffic Congress, 2011, pp. 1–7.

[31] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown, "Implementing an openflow switch on the netfpga platform," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. ACM/IEEE, November 2008, pp. 1–9.

[32] A. Gelberger, N. Yemini, and R. Giladi, "Performance analysis of software-defined networking (SDN)," in *Proceedings of the 21st International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2013, pp. 389–393.

[33] Y.-D. Lin, Y.-K. Lai, C.-Y. Wang, and Y.-C. Lai, "Ofbench: Performance test suite on openflow switches," *IEEE Systems Journal*, vol. 12, no. 3, pp. 2949–2959, 2018.

[34] A. Drescher, J. DeHart, and P. Crowley, "Bayesian factor analysis and performance measurement of the linux forwarding architecture," in *Proceedings of the 14th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. ACM/IEEE, 2018, pp. 28–40.

[35] R. McGuinness and G. Porter, "Evaluating the performance of software nics for 100-gb/s datacenter traffic control," in *Proceedings of the 14th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. ACM/IEEE, 2018, pp. 74–88.

[36] R. Rahimi, M. Veeraraghavan, Y. Nakajima, H. Takahashi, S. Okamoto, and N. Yamanaka, "A high-performance openflow software switch," in *Proceedings of the 17th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2016, pp. 93–99.

[37] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of frameworks for high-performance packet io," in *Proceedings of the 11th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. ACM/IEEE, 2015, pp. 29–38.

[38] R. Durner, A. Blenk, and W. Kellerer, "Performance study of dynamic qos management for openflow-enabled sdn switches," in *Proceedings of the 23rd International Symposium on Quality of Service (IWQoS)*. IEEE, 2015, pp. 177–182.