Computer Communications 225 (2024) 44–53

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

Exploring Data Plane Updates on P4 Switches with P4Runtime

Henning Stubbe, Sebastian Gallenmüller *, Manuel Simon, Eric Hauser, Dominik Scholz, Georg Carle

TUM School of Computation, Information and Technology, Technical University of Munich, Boltzmannstrasse 3, 85748, Garching near Munich, Germany

ARTICLE INFO

Keywords:
Reproducibility
Network experiments
P4
P4Runtime
Control plane

ABSTRACT

The development and roll-out of new Ethernet standards increase the available bandwidths in computer networks. This growth presents significant advantages, enabling novel applications. At the same time, the increase introduces new challenges; higher data rates reduce the available time budget to process each packet. This development also impacts software-defined networks. Their data planes need to keep up with the increased traffic rates. Nevertheless, the control plane must not be ignored; fast reaction times are necessary to handle the increased rates handled by data planes efficiently.

In our work, we analyze the interaction of a high-performance data plane and different implementations for the control plane. We selected a P4 switching ASIC as our data plane. For the control plane, we investigate vendor-specific implementations and a standardized implementation called P4Runtime. To determine the performance of the control plane, we introduce a novel measurement methodology. This methodology allows measuring the delay between the initiation of rule updates on the control plane and their application on the data plane. We investigate the behavior of the data plane, its performance and non-atomicity of updates. Based on our findings, we apply different optimization strategies to improve control plane performance. Our measurements show that neglecting the control plane performance may impact network behavior due to delayed updates, but we also show how to minimize this delay and, thereby, its impact. We have released the experiment artifacts of our study including experiment scripts and measurement data.

This publication is an extension of the paper *"Keeping up to Date with P4Runtime: An Analysis of Data Plane Updates on P4 Switches"* originally published at the IFIP Networking Conference 2023 [1].

1. Introduction

Two ongoing trends in the development of computer networks are the continued growth of bandwidth and the increase of flexibility for networks enabled by software-defined networking (SDN). The programmability and increased bandwidth provide the foundation to run novel applications on these networks. Examples of such applications include distributed control processes in the area of transportation, industry, and medicine [2]. Using the improved programmability of the network, e.g., through OpenFlow [3] or P4 [4], we can dynamically adapt networks to the requirements of a specific application. These requirements include the application-specific protocol and the application's operational demands, such as the minimal bandwidth or the maximum latency. Consequently, new applications with custom protocols and the popularity of programmable network devices thrive.

A key component that enables network programmability in OpenFlow and P4 is the match–action table. Information extracted from protocol headers is compared to the patterns stored in this table structure (i.e., the match). Based on the detected pattern, specific tasks are executed (i.e., the action). The behavior of a particular protocol is realized by filling this table with the required match–action pairs. The contents of this table can be modified at the network device's runtime to adapt the behavior. However, this transition from an old table state to a new one can cause harmful delay or packet loss [5–7]. Higher data rates exacerbate this problem, as more flows or packets can be affected by these transient states of network devices.

A second topic essential for the implementation of new applications is reliable execution time. Consider a situation where the roll-out of table updates involves several devices. In the past, multiple different strategies were proposed to perform this roll-out avoiding unwanted behavior [8]. One possible mitigation strategy is a two-phase commit, where table updates must be confirmed by participating switches before the updates are applied to the network. Another possibility is

* Corresponding author.
E-mail addresses: stubbe@net.in.tum.de (H. Stubbe), gallenmu@net.in.tum.de (S. Gallenmüller), simonm@net.in.tum.de (M. Simon), hauser@net.in.tum.de (E. Hauser), scholz@net.in.tum.de (D. Scholz), carle@net.in.tum.de (G. Carle).

https://doi.org/10.1016/j.comcom.2024.06.020
Received 28 December 2023; Received in revised form 28 April 2024; Accepted 30 June 2024
Available online 3 July 2024

the generation of a specific sequence of table updates. These updates are designed and scheduled in a way that avoids or minimizes the harmful effects of table updates on the network behavior. However, to orchestrate the previously mentioned strategies, the update behavior of individual devices must be known. A crucial component of the update behavior is the control plane handling the rule application on network devices. The control plane is typically realized in software. Therefore, table updates are subject to random interrupts or short-time overload, impacting the update behavior of network devices.

In this work, we focus on high-performance P4 switches and present a measurement methodology to investigate their update behavior. Due to the importance of table updates across devices, we further want to determine the table update latency quantitatively and qualitatively. Therefore, we introduce a setup that allows the hardware timestamping of the entire data and control plane traffic using the same hardware reference clock. Based on this investigation, we apply optimization methods to create a more deterministic update behavior. We focus our investigations on P4Runtime, which was introduced to configure, among others, the mentioned match–action tables in P4 switches. Aiming to unify device-specific implementations of similar control plane interfaces, P4Runtime plays a key role in enabling programmable network devices across different devices or vendors. The contributions of this work include:

- the creation of an accurate measurement setup to monitor data and control plane simultaneously,
- the evaluation of reconfiguration behavior and latency of a programmable switching ASIC,
- the comparison of different control plane implementations, and
- the optimization of Linux that hosts the control plane to reduce latency and jitter.

The remainder of this work is structured as follows. Section 2 introduces an overview of the architecture of P4 switches and presents a short case study demonstrating the potential impact of table updates. Related work is covered in Section 3, which investigates the state of the art for table updates in software-defined networks with a focus on P4. Afterward, Section 4 introduces the experiment setup used throughout this work. Subsequently, Section 5 evaluates the behavior of an exemplary P4-programmable ASIC. Section 6 briefly introduces the experiment artifacts that we provide to the community for experiment reproduction. Finally, Section 7 concludes this work.

2. Background

To better understand a system's behavior, deeper knowledge about its fundamental architecture can be useful. In this section, we present a brief overview of the typical components of today's switch architectures. Based on this architecture, we describe the process for pushing updates from the control to the data plane. We list the involved components and the way of the update messages from initiation to their final application. Our description focuses on the delay introduced by different components handling the update message. Afterward, we provide a case study on the impact of control plane latency.

2.1. Router architecture

The typical architecture of current networking devices is shown in Fig. 1. Instead of one single processing unit, the architecture of these devices features not only a traditional CPU but also a configurable ASIC. This fundamental architecture is present in the switches across multiple vendors, such as Intel [9], Cisco [10], and Arista [11], or the open hardware design of the Wedge400 [12]. While the switch's CPU is programmed to handle subtasks related, in particular, to the control plane, the ASIC focuses on the data plane. Thus, this architecture combines the benefits of CPU and ASIC. The CPU is versatile but comparatively slow, a good fit to handle the complex tasks in

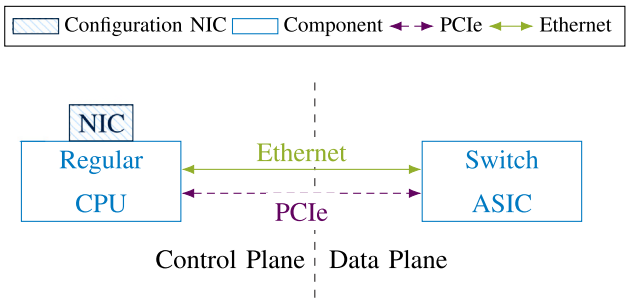

Fig. 1. High-level switch architecture.

Table 1
Optical Ethernet standards, transmission rates, and corresponding serialization delay of a minimum-sized Ethernet packet (60 B packet + 4 B frame check sequence + 12 B inter-packet gap + 7 B preamble + 1 B start-of-frame delimiter), number of packets impacted for 1 µs of control plane delay.

IEEE standard	TX rate [Gbit/s]	Serialization delay [ns]	Impacted packets [#/µs]
802.3z [13]	1	672.0	2
802.3ae [14]	10	67.2	15
802.3bm [15]	100	6.7	150
802.3bs [16]	400	1.7	589
P802.3dj [17]	1600	0.4	2500

the control plane. The ASIC offers raw packet processing performance with limited complexity, which enables the data planes to handle billions of packets per second on a single ASIC. The communication between these two components is essential to ensure that CPU and ASIC work together as a single switch. Here, two popular options for communication channels are observable and often combined: Ethernet and PCIe. The former allows for the message exchange as envisioned in software-defined networking, e.g., for the data plane to forward unhandled packets to the control plane. PCIe, on the other hand, is a convenient interface to change the ASIC's state. Such state changes can include, e.g., reprogramming the ASIC to handle packets differently or updating its configuration, such as match–action table entries in P4.

Hence, in this switch architecture, a control plane update, issued externally by another party, is processed as follows: Initially, the other party sends its update message to the system in charge of the switch's control plane running on the traditional CPU. The control plane system often has a separate NIC (cf. Fig. 1) to receive such update messages. After reaching said NIC, the update message passes through the system's network stack until it reaches the application listening on the addressed port. The application then processes the message's information and translates it into a sequence of PCIe transactions involving the ASIC. In case of an update, these transactions ensure the addressed table and table entries exist, replacing the intended table entry value. Only after that, the data plane processes future packets according to the updated table. Therefore, a single control plane update message is passed over multiple layers of hardware and software on the control and data plane, impacting the observed delay for the update message.

2.2. Case study: Control plane latency

Our case study focuses on the impact of latency in the control plane. More precisely, we want to determine a lower bound for the latency in state updates caused by the processing of the update message inside the control plane. In view of the architecture in today's switches, the update process can be divided into several steps:

(1) the update process starts with the reception of a P4Runtime message on the control interface of the control plane;

(2) then, the information required for the update is transferred via PCIe from NIC to RAM (or directly into the CPU cache [18]);
(3) afterward, the update message is processed on the CPU itself, and,
(4) finally, the update process is completed with the PCIe transfer of the computed update from the CPU to the ASIC.

Gallenmüller et al. [19] measured a median latency of 3.3 µs between the ingress and egress interface of a simple packet forwarding application. Their latency measurements were performed on an Intel Xeon D-1518 CPU running Debian-based Linux. A similar system on a chip (SoC) and operating system (OS) is also used in the control plane of the switch that was investigated in this paper. Gallenmüller et al. used a simple forwarding application, introducing a CPU overhead of approx. 100 clock cycles per packet. The similar hardware and software architecture provides a realistic lower-bound estimate for the processing of packets in the control plane. For the control plane application, we expect higher latencies due to the more complex processing of update messages. Applying the update to the ASIC on the data plane involves an additional PCIe transfer of information, causing additional latency. Neugebauer et al. [20] measured a median round trip time of 800 ns for a minimum-sized 64-B packet across the PCI express bus. Their measurements do not involve any processing of the packet data on the CPU, e.g., in the driver. The underlying hardware (NIC, x86 CPU), control plane OS (Debian-based Linux), and the procedure (message reception, processing, and transfer via PCIe) are similar to a typical switch data plane. Based on these numbers, we expect a control plane latency in the order of several microseconds.

Table 1 shows the packet serialization delay for minimum-sized Ethernet packets for standardized bandwidths between 1 Gbit/s to 1.6 Tbit/s. To show the impact of delays, we added the number of impacted packets over a timespan of 1 µs for each rate. For illustration purposes, our table only lists the impact of a 1-µs delay. During a 1-µs timespan, two minimum-sized packets will pass a 1-Gbit/s link. A delay of 1 µs on the control plane would, therefore, impact two packets on a 1-Gbit/s link on the data plane. While a low number of impacted packets may be considered negligible, their number grows for higher bandwidths (cf. Table 1).

The numbers reported in Table 1 only consider a 1-µs delay. If we assume the higher control plane delay of 3.3 µs, the numbers are multiplied by a factor of 3.3. Worst-case latencies over 1 ms were observed by Gallenmüller et al. [19,21] and Neugebauer et al. [20]. With these ms-delays, the number of impacted packets grows into millions. These high, indicative numbers and the high variance of the previously mentioned delays justify a closer investigation, which we will present in the following.

3. Related work

Our work investigates *switches* executing *P4* programs managed by software-based *control planes*. In the following, we investigate related work from these three areas.

Switches. Updating the forwarding rules of running networks can cause unwanted side effects if partially old and new configurations are applied to specific packets [5–8]. To avoid these transient states between updates and their impact, Reitblatt et al. [22] have introduced a set of primitives to perform consistent updates on programmable switches. Their architecture guarantees per-packet consistency, i.e., at any point in time, there is a well-defined ruleset to be applied to a specific packet. Tycho-Förster et al. [8] provide a detailed survey investigating different strategies to solve the negative impacts of switch updates. Their survey mainly focuses on the algorithmic solution of network updates. Jin et al. [23] propose Dionysus, a mechanism to lower update times by optimizing the scheduling of individual switch updates. Due to the different impacts of individual updates on each other, an optimized schedule helped lower the update deployment by 61% in a real-world testbed. OFLOPS-SUME [24] is a framework that allows the measurement of OpenFlow data and control planes. A study on the software-based Open vSwitch and an Edgecore hardware switch uncovered inconsistent transient behavior during table modifications and modification delays of up to several hundred milliseconds. Han et al. [25] present BlueSwitch, a switching architecture that enforces per-packet consistency on a single switch. Their architecture solves the problems of inconsistent behavior on a hardware level, demonstrated on a NetFPGA-10G-based prototype.

P4. P4 [4] is a domain-specific language to program the data plane, which supports programming different types of data plane devices, such as switches. P4Runtime [26] standardizes the management of data planes utilizing a vendor-independent API. This API allows rule insertions, deletions, or updates of P4 data plane elements. P4Runtime relies on gRPC [27], a high-performance framework for remote procedure calls. Adoption of both is growing; e.g., Intel Tofino is a switching ASIC that supports P4 and P4Runtime natively [9]. Song et al. [28] measure an update performance between 30 000 and 80 000 entries per second for an Intel Tofino switching ASIC, depending on the insertion batch size and the utilization of the match–action table. Zeng et al. [29] observed similar limitations. They attribute the low performance to the slow control plane CPU, a limited PCIe interconnect between the CPU and switching ASIC, and the limited amount of memory on the ASIC, which requires expensive hashing computation and lookups.

Software-based control planes. Recalling the switch architecture (cf. Fig. 1) prevalent on P4 hardware, P4Runtime messages are typically processed in a software-based control plane. Therefore, we need to investigate Linux, the operating system (OS) used for control planes. Linux-based software packet processing systems are subject to delays in the millisecond range. Gallenmüller et al. [19,21] describe several effects causing that type of delay, such as the OS network stack or interrupts. Modern ASIC-based (P4) switches typically rely on a Linux-driven control plane that introduces the same delays to switches. Linux-based network stacks have been researched in the high-performance, low-latency networking community for decades. The Linux network stack employs a technique called NAPI [30] that allows dynamic switching between an interrupt-based and a polling-based packet reception. This adaptive mechanism improves throughput but introduces jitter and latency compared to a purely polling-based approach. Unsatisfied with this stack's performance, alternative user-space implementations were proposed, including DPDK [31]. DPDK relies exclusively on polling for packet reception, lowering jitter and latency. To make the benefits of DPDK easily accessible, DPDK features several examples, including a basic Layer 2 (Ethernet) forwarder called `l2fwd`. Naturally, bare-bone sample implementations such as this are in no direct comparison to full-fledged control plane implementations. The latter does not only handle packets but has to solve more advanced problems such as abiding by access policies or computing optimal routes. Nevertheless, achieving performant control planes benefits from performance improvements in any of its components. Thus, performant user-space implementations, such as DPDK, are of interest for such projects. Other projects also picked up on the idea of increasing DPDK's ease of use, e.g., the packet generator MoonGen [32]. The preemptive nature of the Linux kernel allows interrupting running processes, introducing jitter. A study by Reghenzani et al. [33] investigates real-time patches available for the Linux kernel that create a more stable and predictable behavior. The tickless Linux kernel (cf. Subsection 5.1 on the NOHZ kernel) [34] further improves predictability and low-latency behavior for applications by disabling scheduling interrupts (also called *ticks*) on specific CPU cores.

Processing performance. Data plane updates on switches may cause unwanted effects for packets processed during transient states. Although solutions exist to avoid the negative impacts of these transient states, we did not find a detailed study that investigates the update process for

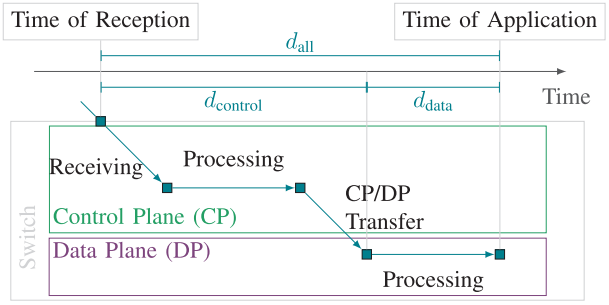

Fig. 2. Switch update delay (d) for forwarding and processing of control plane messages.

a single, modern P4 switch. When offloading applications to P4 data plane elements, table updates may impact a wide range of different applications. In this work, we want to create a methodology to measure not only the maximum number of possible updates for a given interval but also to investigate individual data plane updates and their impact. In addition, we consider delays that may be introduced by the control plane and how to lower the control plane impact.

4. Measurement methodology

For 100-Gbit/s-networks, a new packet can be received in less than 10 ns (cf. Table 1). Our measurement methodology and setup need to consider this challenging requirement to perform effective measurements. For these measurements, we need to correlate control plane and data plane traffic to determine the time between the reception of a table update in the control plane of a P4 switch and its actual application on the data plane of this P4 switch. In the following, we introduce our general approach to measuring and the specific setup used for our measurements.

4.1. Scenario description — A table update

Fig. 2 shows the process of a table update for a single P4 switch. The table update is received (*Time of Reception*) and processed on the control plane, transferred to and processed on the data plane before the table update is applied to the packet on the data plane (*Time of Application*). We refer to the total delay between reception and application as d_{all}, which can be split into two components: *(i)* The time the update is handled on the control plane, which we call $d_{control}$, and *(ii)* the time the update is handled on the data plane, which we call d_{data}. Despite being part of the same switch, the control plane and data plane can be seen as two interconnected but separate systems (cf. Fig. 1). On the **control plane**, the table update is received on a NIC port that is physically connected to the CPU of the control plane. The control plane software runs on a Linux system that runs on the previously mentioned CPU. During $d_{control}$, the message is handled in software and, hence, subject to the jitter and delay caused by the OS, i.e., the Linux kernel. The **data plane** handles the traffic received via the actual switching ports of the P4 switch. These ports are physically connected to the switching ASIC and not directly connected to the control plane CPU. As soon as the table updates are handed over to the data plane, they are no longer subjected to the previously mentioned effects on the control plane. The main focus of our investigation is the measurement of delay and jitter during the time of a table update. Additionally, we are interested in the events on the data plane during this timeframe. Events of interest are, e.g., packet drops or partially applied updates.

4.2. Measurement challenges

The control plane side of the switch uses well-known system components, i.e., CPU, NIC, or OS. With access to the control plane OS and due to the well-understood architecture, we can perform white-box tests. Such tests allow the investigation of table updates from within the switch during $d_{control}$. For Linux, standard tools are available such as the profiler perf, that can be used to investigate a system. The data plane side is a different kind of system. It runs on a proprietary ASIC that limits our measurement capabilities. With limited access to the inner structure of the switching ASIC, we cannot perform white-box tests and cannot directly measure the time of application, i.e., d_{data}.

Despite the ability to run white-box measurements on the control plane, these measurements come with attached restrictions. Running an additional measurement application on the control plane may impact the behavior of the control plane itself. For instance, jitter can be introduced by software interrupts for profiling, which hamper accurate and precise latency measurements in software, especially considering the ns-delays that we expect to measure.

The switch architecture with the separate control and data plane systems impedes delay measurements. Both, the CPU and the P4 switching ASIC, utilize separate system clocks. To determine d_{all}, both clocks need to be synchronized. Control and data plane possess clocks that provide the necessary timer resolution in the ns-range. For x86-CPUs, there is the TSC that offers a high timer resolution based on the CPU clock, and our switching ASIC also offers timestamps with a ns-resolution. Due to the proprietary nature of the switching ASIC, synchronizing via standardized timing protocols such as NTP or PTP is not possible.

4.3. Solution

Due to the impacts of white-box measurements on the control plane and the impossibility of such measurements on the data plane, we opted for a black-box measurement approach. In the black-box approach, measurements are performed externally. Here, packets are timestamped just before they arrive on the switch or shortly after they have left the switch. For the control plane, the black-box approach does not require the deployment of measurement software on the switch, i.e., there is no performance impact on its behavior. The black-box approach also works for the data plane as measurements are created externally.

To perform the black-box measurements, we use a setup already described by Gallenmüller et al. [21]. This setup uses passive optical splitters that create a passive clone of each packet that is received and sent by the switch. With passive splitters, we avoid the impact of any active component on the delay of packets. The packets are precisely timestamped in hardware, which avoids any impact of OS or other software on measurements. At the same time, hardware timestamping ensures precise timestamps in the ns-range.

The black-box approach requires the observation of multiple ports at the same time. One port is attached to the control plane; the other ports are attached to the data plane. Timestamping the time of reception on the control plane port can be done by recording the timestamp of the table update message. The time of application cannot be determined directly as the data plane does not create an explicit update message at the time of application. However, we can observe the impacts of the table update indirectly. First, we timestamp all ingress and egress traffic from a specific observed data plane port. Second, we detect the effects of the update message on the data plane traffic (i.e., the change of a specific header field). Finally, we can correlate the update message on the control plane with the first egress packet that contains the specified update. As both packets are timestamped, we can calculate d_{all}.

The black-box measurement does not require precisely synchronized clocks between control and data plane. We use a single device for timestamping all packets. This timestamping device uses a shared clock for all of its ports, therefore, all timestamps use a common reference clock, rendering synchronization unnecessary. The lack of synchronization and the lack of software deployed on the device under test lowers the complexity of the measurement setup.

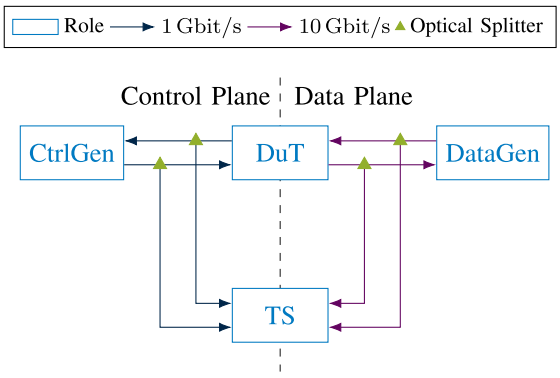

Fig. 3. Measurement setup overview.

Table 2
Hardware components of our measurement setup.

Device	CPU	Memory	NIC
CtrlGen	Intel Xeon E5-1650	128 GB	Intel X710
DataGen	Intel Xeon E5-1650	128 GB	Intel X710
DuT	Intel Xeon D-1548	32 GB	Intel I350
TS	Intel Xeon D-1548	32 GB	Endace DAG 10X4-S

4.4. Setup

The setup (cf. Fig. 3) implements the previously described solution. It consists of four roles: two load generators (CtrlGen & DataGen), one device-under-test (DuT), and one time-stamper (TS). Both load generators supply the DuT with constant bitrate traffic, differing mainly in the targeted component of the DuT. While one load generator applies the load to the control plane interface of the DuT (CtrlGen), the second one targets the DuT's data plane (DataGen). In our experiments, both load generator roles are assumed by the same physical host. The TS monitors the information exchanged between the load generators and DuT via optical splitters. The splitter-based setup allows the TS to timestamp events on the control and data plane with the same shared reference clock. The optical splitters are passive; hence, the measurement system does not introduce additional jitter. All used cables have the same length of 3 m between CtrlGen, DataGen, TS, and the DuT, i.e., the delay is not skewed by different cable lengths.

Both load generators employ MoonGen [32] as packet generator. They run on an Intel Xeon E5-1650 CPU equipped with 128 GB memory and an Intel X710 NIC. The DuT, the Stordis BF2556X-1T-A1F switch, consists of a P4-programmable ASIC combined with an Intel Xeon D-1548 CPU equipped with 32 GB memory and an Intel I350 NIC on the control plane [35]. Notably, this DuT features an optical control plane interface that enables it to be attached to the passive optical splitters. We use different applications on the control plane of the DuT to investigate the impact of techniques, such as NAPI or DPDK, on latency and jitter. Lastly, to timestamp packets, we used Endace's DAG 10X4-S [36] on a host whose properties match the DuT's. The quad-port Endace NIC supports hardware timestamping the entire traffic with the same clock on all its ports at 10 Gbit/s line rate with a resolution of 4 ns. Table 2 lists the hardware components of our setup. While the DuT runs on Ubuntu 20.04 LTS and the TS relies on Ubuntu 18.04.1 LTS, Debian buster is used as the load generator's OS. The different OS distributions and versions were used due to the different requirements of the software frameworks for the switching ASIC, the Endace timestamping framework, and MoonGen.

4.5. Experiment description

Based on our case study (Section 2.2), we expect a significant impact on d_{all} on the control plane. We plan to investigate the control plane application itself and additional parameters for the control plane OS. Therefore, we discriminate our experiments based on the P4Runtime implementation used and the DuT host configuration. Section 4.6 discusses the different types of control plane applications that we investigate. Additionally, we measure the impact of either the OS default configuration or a configuration optimized for low latency. Here, the low-latency optimized configurations build on experiences from previous work [19,21].

All experiments share the same fundamental structure. Following an initial configuration phase, the setup behaves as follows: The *DataGen* emits packets with a size of 64 B at a constant bit rate. Each of these packets is processed by the DuT and afterward sent back to the DataGen. On the data plane of the DuT, every received packet is matched against a P4 match–action table. This table is configured as an exact match with a MAC address as the key value. Due to the exact match, the program uses SRAM exclusively. TCAM is not used because the exact match does not rely on any ternary or longest prefix matching. Moreover, this data plane program, including the tables, is very lightweight. We only require one of the several available pipelines and allocate less than 10% of its resources. The P4 program writes the value of this table to the source address field of the Ethernet header for the egress traffic. Throughout the experiment, the value of the MAC address in the P4 table is constantly modified via the control plane. The *CtrlGen* sends a stream of P4Runtime modification messages at a constant rate. Each of these messages contains a new value for the Ethernet source address. We use a packet forwarder on the DuT that receives the P4Runtime modification messages and applies the modification utilizing one of the respective P4Runtime implementations. After processing the P4Runtime messages, the forwarder sends the packets back to the CtrlGen. Hence, enabling the TS to capture ingoing and outgoing packets for both load generators.

Based on this observation of received and sent packets, the processing delay (d_{all}) of the DuT can be determined, i.e., the time required for a modify instruction received via the control plane to be visible on the data plane. To detect the changes on the data plane in a timely manner, high packet rates with a short inter-packet gap are required. Therefore, the data rate on the DataGen impacts the accuracy of our measurements and should be maximized. The packet rate on the CtrlGen does not impact measurement accuracy. A goal of our measurements is to show a use case during normal operation, not during phases of overload with unnaturally high latency. Thus, we choose conservatively low packet rates for the CtrlGen to avoid overloading the control plane. To evaluate the processing latency induced by the DuT, during an experiment, the CtrlGen transmits instructions at a constant rate. All observed processing delays are recorded by the TS for later evaluation. To avoid latency artifacts caused by newly started applications due to an empty CPU cache, we exclude the first seconds of each measurement for our latency calculation.

4.6. Control plane applications

Our measurements consider different impact factors for the control plane application. First, we investigate a forwarder application where packets are exclusively handled on the control plane. This measurement acts as a baseline measurement to determine the impact of the control plane on the forwarding process without any data plane involvement. Second, we measure the impact of the actual control plane implementations. We consider aspects such as the used programming languages or vendor-specific implementations. Third, we determine the impact of the table update messages. Therefore, we generate different kinds of table update messages.

Forwarder implementation. We expect that the choice of the packet forwarder handling the modification messages significantly impacts processing latency. Therefore, we investigate three different kinds of forwarding applications:

(F1) a Python-based implementation using the Linux network stack;

(F2) a C-based implementation relying on the Linux network stack;
(F3) a C-based DPDK implementation (`l2fwd`).

Past experience suggests that compiled applications, i.e., (F2) and (F3), have an advantage compared to the interpreted Python (F1). We expect further performance benefits for the DPDK-based implementation (F3) that relies on the optimized DPDK stack entirely bypassing the Linux network stack.

P4Runtime implementation. As mentioned, we discriminate our experiments based on the used P4Runtime implementation with the goal of measuring the impact of the different P4Runtime implementations. Therefore, as said, the DuT's forwarding application will execute the respective control plane implementation's callback for each received packet. In this work, we consider three different implementations available on our switch:

(I1) a Python-based implementation supplied by the vendor of the ASIC, targeting the DuT's ASIC;
(I2) a C++-based gRPC implementation designed to conform with the P4Runtime specification;
(I3) a C++-based implementation, also relying on the vendor-specific, and, thus, device-specific, interface.

Given that both (I1) and (I3) were written with a particular switching ASIC in mind, a performance benefit due to device-specific optimizations is likely. At the same time, as mentioned before, an advantage of the compiled C++ applications, i.e., (I2) and (I3), compared to the interpreted (I1), is expected. Both the vendor-specific controller implementation and the P4Runtime implementation are built in software. While the latter is a generic API used for different switches, the vendor-specific API can be specifically optimized for the underlying hardware.

P4Runtime table manipulation operation. Following the P4Runtime specification [26], a limited number of operations can be performed on tables:

1. new entries, non-existing keys with their value, can be added via *insert*;
2. present or predefined entries may be *removed*;
3. alternatively, *modify* allows to replace existing entries retaining the key while changing the associated value.

Among these, the modify operation is the most expressive. The modify operation can be used to implement the add and remove operations. An exemplary implementation could rely on pre-populated tables and a noop action, such as P4's `NoAction`. Then, an add would update the entry to replace the noop action with the desired one. Conversely, a delete would modify the entry to use the noop action instead. While this modify-only approach relies on large tables, the availability of content-addressable memory in modern network devices compensates for the overhead of an increased table size. Following this argument, our work focuses on the modify operation.

5. Evaluation

The center of our investigation is the delay and its jitter observed between control plane table modification and the modification's manifestation on the data plane. This investigation requires an in-depth analysis of the DuT behavior during the application of modify operations.

During each experiment, a single 10 Gbit/s port of the DuT's data plane was subjected to load traffic with a constant rate of 6 Gbit/s at a packet size of 64 B (approx. 8.9 Mpkt/s). Simultaneously, on the control plane, table modifications were triggered with a rate of 100 Hz by the CtrlGen. The control plane traffic was sent to the DuT's 1 Gbit/s NIC port directly attached to the control plane.

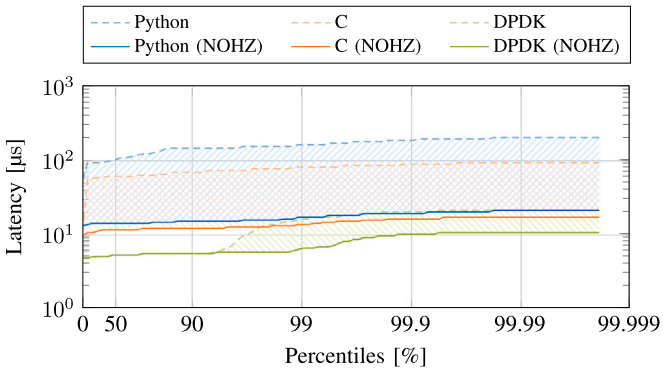

Fig. 4. Baseline measurements for the processing latency of the control plane.

Overwhelming a device with packets fills up buffers, leading to high latencies. To determine the latency of the non-overloaded DuT, we want to avoid buffering of any packets on both the control and data plane. Both planes' rates are chosen such that an increase of either results in a degradation of observed latencies, i.e., preceding analyses indicated a non-overload state of the DuT for these parameters. For low data plane rates, we did not observe a correlation between data plane utilization and processing latency. For higher rates, however, latency on the data plane increased linearly.

We excluded a warm-up phase of 10 s at the beginning of each measurement. This was done to avoid measuring latency caused by ramp-up effects of the control plane application, such as first-time cache misses. The measurement time for our presented results lasted for 50 s.

5.1. Table modification delay—Packet reception

We assume the software-based control plane to contribute significantly to the overall delay of table modifications. Therefore, we want to investigate the major steps in the processing chain of table modification messages. The first step in this processing chain is the reception of the modify messages on the control plane. For the reception, we want to investigate relevant aspects that we previously identified as factors contributing to delay: (a) the programming language of an application, (b) the software interface used to access packets, and (c) the interrupts introducing jitter to a running application.

To measure the delay of the message reception procedure, we use a simple Layer 2 forwarding program. To avoid any impact of the controller application on the message reception, we only run the forwarder on the control plane system. The forwarder receives packets on the NIC port of the control plane and sends them out on the same port without any further processing. We use this forwarder to measure the latency in our setup, which relies on hardware-timestamped packets. Compared to the control plane implementation, a forwarder includes additional tasks, such as sending the packet. Therefore, our measurements of the forwarder overestimate the latency of the investigated control plane implementations to some extent. Assuming symmetric receive and transmit paths, the measured roundtrip times can effectively be halved to determine the message reception times.

Impact of the programming language. We have chosen two forwarders, written in Python and C, to demonstrate the effect of the programming language on the packet processing delay. Python is an interpreted scripting language relying on an automated garbage collector. The interpretation of the code and the execution of the garbage collector may introduce unwanted jitter into a controller application that we expect to run continuously. The C language is compiled and does not use an automated garbage collector; therefore, we expect a lower jitter. Fig. 4 shows the results of the Python and C forwarder as dashed lines. We use a percentile distribution graph to visualize the measured latency [37].

This type of graph highlights the latencies at high percentiles (>99.x), characterizing not only latency but also the jitter in a more expressive way. This is a clear benefit over traditional representations of latencies in histograms or CDFs that hide high but rare latency events in long, hard-to-read tails. For the Python forwarder, we observed a median latency of 106 µs that rises up to 205 µs for the 99.99th percentile. The C forwarder has a median latency of 61 µs that increased to 94 µs for the 99.99th percentile. These numbers indicate a significant advantage for the C language. The latency is significantly lower for the C forwarder, but also the jitter, as numbers increase substantially less for the high percentiles of the C forwarder.

Impact of the packet reception API. Linux offers a flexible but complex network stack supporting various protocols or dynamic mechanisms such as NAPI. Frameworks, such as DPDK, allow bypassing the Linux network stack and provide their own simpler stack that offers higher performance. For this work, we investigated whether DPDK provides lower latency or jitter for the control plane. To measure the latency of DPDK, we use the DPDK `l2fwd`. Its latency is shown in Fig. 4, with a median latency of 5 µs and a latency of 22 µs for the 99.99th percentile. These figures demonstrate that DPDK can significantly improve latency and jitter compared to the previously measured C-based forwarder utilizing the Linux network stack. However, the figures also indicate that even with DPDK, outliers do occur. We attribute these outliers to interrupts issued by the Linux kernel briefly stopping packet processing. We observed similar outliers for DPDK applications in previous work [21].

Impact of the Linux kernel. The Linux kernel utilizes interrupts for specific tasks such as scheduling. The execution of these interrupts can, therefore, introduce jitter to currently running applications. To mitigate these problems, a low-latency kernel was developed [34]. These low-latency features have to be compiled into the Linux kernel. As an additional measure, critical processes on the DuT can be pinned to isolated CPU cores (cf. Gallenmüller et al. [19,21]). To utilize the features of this kernel, we compiled a new Linux kernel with the flag `CONFIG_NO_HZ_FULL` enabled. In this work, we refer to that kernel as the *NOHZ* environment. The Linux scheduler needs to regularly check if a CPU core needs to be yielded to another process. This check is performed during regular interrupts or so-called *ticks*. The NOHZ kernel allows to disable these interrupts on dedicated cores. With scheduling interrupts disabled, processes running on these dedicated cores are no longer interrupted, lowering the observed latency and jitter. However, it also means that these CPU cores cannot be shared between applications. Therefore, only a single process can run on a NOHZ core. If two or more processes are executed on such a core, interrupts are re-enabled, and the kernel behaves like a regular Linux kernel. By comparing these two environments with each other, the possible impact of a non-optimized standard configuration on the DuT's performance can be estimated; thus, possibilities to shape the DuT behavior for high-performance scenarios become apparent. To ensure comparability, both environments rely on the same Linux kernel version: `5.4.0-105.119`.

We repeated our measurements of the three different forwarders on a NOHZ kernel, to measure the potential for improvement. Fig. 4 visualizes these latencies using solid lines. We see significant improvements for all three forwarders. For the Python forwarders (median: 14 µs, 99.99th percentile: 17 µs) and C (median: 12 µs, 99.99th percentile: 22 µs), latency and jitter were significantly improved. For the DPDK forwarder, we measured the same median latency of 5 µs as for the regular Linux kernel. At the 99.99th percentile, latency was reduced to only 11 µs. We observed that the latency increase for the DPDK forwarder, running on a NOHZ kernel, happens at a higher percentile, e.g., the 99th percentile.

Looking at the results, we claim to have achieved our initial goals. We have shown that the latency and jitter can be significantly improved. The best results were achieved using a compiled language based on the DPDK framework running on a NOHZ Linux kernel, improving latency as well as jitter significantly. OS interrupts will still impact the latency of DPDK applications, therefore, the tickless kernel also improves the jitter of DPDK applications. However, further optimizations of the kernel network stack will be irrelevant as packets bypass the OS network stack when using DPDK.

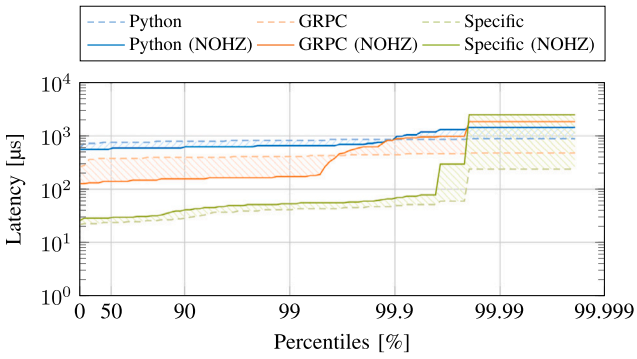

Fig. 5. Percentile distribution of processing latency.

5.2. Table modification delay—Packet processing

The previous section investigated the impact of the packet reception process on latency and jitter. In this section, we quantify the latency and jitter impact of the actual control application. Based on the previously identified benefits of DPDK, we adapted the three control applications to use DPDK. For our investigation, we compare the observed receive time differences between a forwarded table modify instruction and the first data plane packet affected by this modification at the TS. We introduced this timespan as d_{all}. The results of this comparison are summarized in Fig. 5 as a percentile distribution. The figure depicts the recorded receive time differences, i.e., the DuT's processing delay of the three investigated implementations, emphasizing higher percentiles.

Python-based implementation (I1). The results in Fig. 5 show a median latency of approx. 590 µs and 557 µs for the generic and NOHZ Python implementation, respectively. However, starting around the 99.9th percentile, the NOHZ variant performs worse, i.e., during the experiments, it has a higher tendency for outliers. The highest processing latencies for generic and NOHZ variants were approx. 1442 µs and 885 µs, respectively.

Despite profiting from DPDK, the Python implementation has a comparatively high mean processing latency. We attribute this to Python being an interpreted language with a garbage collector and its global interpreter lock preventing concurrent execution of Python byte-code. While implementations other than the tested CPython might yield improved performance, this is not considered in this work.

C++-based gRPC implementation (I2). Similar to the Python-based implementation, the observed latencies of the NOHZ variant surpass the generic variant but for about 1% of the cases. With 262 µs to 475 µs and 126 µs to 1835 µs, generic and NOHZ variants offer lower minimum processing delay than the Python-based implementation. However, outliers with poor worst-case latency overshadow this benefit.

Arguably, the best-case delay, compared to the Python-based approach, can partly be attributed to the availability of compile-time optimizations. Additionally, the benefit of avoiding interrupts positively influences induced latencies in the NOHZ case. However, this benefit turns into a disadvantage when considering percentiles above 99%. As low-latency optimization is a delicate undertaking, an inconvenient combination of interrupts and modify instruction arrival is likely the cause of this behavior. Plus, the implementation's use of GRPC, a library relying on threading, conflicts with the need to have at most one runnable process or thread per CPU when using NOHZ.

C++-based vendor-implementation (I3). The third investigated implementation is the vendor-provided ASIC-specific implementation. With best-case processing latencies around 22 µs to 26 µs rising to 237 µs to 2490 µs in the worst-case, this implementation provides the most promising processing delay, apart from few outliers. In contrast to the other implementations, this ASIC-specific implementation noticeably suffers from NOHZ optimization. Given that NOHZ optimizations shine when processes are pinned to individual cores and with the architecture of the DuT's non-ASIC component in mind, a probable cause for this negative correlation stems from the inability to pin processes appropriately. On the other hand, one major benefit of this ASIC-specific implementation and architecture shows when comparing this implementation's performance with the other two: the ASIC-specific implementation outperforms both.

Discussion. Our measurements show a clear difference between the three implementations. We observed the highest median latency for the Python-based P4Runtime Implementation (I1) (590 µs) and the lowest latency for the vendor-specific C++-based Implementation (I3) (22 µs). The C++-based gRPC Implementation (I2) (262 µs) provides a middle ground with a latency between the two other implementations. These numbers show that ease of use and increased flexibility come at a price. In this case, the price can be up to several hundreds of microseconds. Table 1 shows that data plane bandwidths of 100 Gbit/s up to 150 pkt/µs are impacted. The choice of the control plane application can, therefore, affect several ten thousand packets for just a single modification. For percentiles of >99.9, latency rises significantly for all three implementations by 200 to 300 µs. This worst-case latency must be respected if we want to assume that a table modification is applied with a high probability. Therefore, even more packets may be affected during the modification period d_{all}. Intuitively, the move toward a tickless kernel is associated with reduced jitter. However, as discussed, this only partially holds true in these experiments. Results were counter-intuitive; we saw a positive impact on mean latency; however, jitter is significantly increased when looking at higher percentiles. We suspect the root cause to be the complexity of the software architectures used on the DuT; an effective optimization depends on the well-tuned interplay of all components. Arguably, the complexity of the software architecture and the specialization of the hardware architecture suggest a limited portability of these findings. However, there are shared elements across typical data center switches—a control plane based on commercial off-the-shelf hardware running Linux, cf. Fig. 1. Effects caused by this common platform will be observed on all platforms sharing these elements. Other effects that are caused by device-specific hardware or software components must be measured on a per-device basis. However, Gallenmüller et al. [21] indicate that the impact of the common components, Linux and the used software stack, are prone to dominate overall latency. Other factors such as the NIC have only a limited impact on the system performance. To optimize the latency of switches, one should focus on the factors with the highest impact on latency, i.e., the OS and the software stack, which are likely to be similar on all switches.

5.3. Table modification behavior

Moving away from the latency-focused discussion, another observation is noteworthy. For all investigated implementations, an intermediate state of the DuT was observed for some modify instructions. In these cases, neither the match–action table entry before the modification nor the one after was applied to processed packets on the data plane. Instead, the table's default action was applied to up to three consecutive packets. In other words, the modify instruction is not applied atomically. Instead, we assume a table entry modification is realized as a delete followed by an insert. Unfortunately, neither a validation nor a more thorough analysis of this assumption is feasible due to the driver distribution policy. Regardless, we attribute the non-atomicity not to the PCIe bus as the mechanism used to transfer the update to the ASIC. The transfer mechanism does not influence the atomicity. Other approaches, such as Direct Memory Access (DMA), show similar behavior regarding atomicity. Mitigation through software seems feasible but is not within the scope of this work.

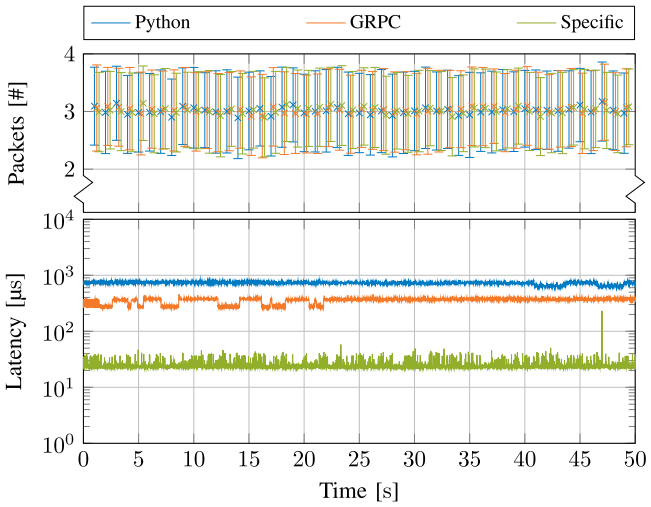

Fig. 6. Per-second mean packet sequence length subjected to default action (outliers show standard error; top) vs. processing latency over time (bottom).

Fig. 6 compares the observed processing latency and the number of times the default action was applied to a packet throughout an experiment. The upper subfigure shows the per-second mean number of consecutive packets processed according to the default table entry of the DuT. Further, the upper and lower fliers in the top subfigure represent the standard error. The depicted number of packets is obtained by dividing the difference in receive timestamps by the serialization delay. For these timestamp differences, we considered the first packet subjected to the default action as well as the first packet matched to any other action.

The lower subfigure depicts the observed processing delays over time, shown as a percentile distribution in Fig. 5. Here, the generic experiments were chosen due to their higher variations in the processing delay when considering lower percentiles. The reasoning behind this argument is as follows. A higher variation in the processing delay allows for increased uniqueness of the observed latencies over time. Consequently, making it easier to spot a possible correlation between the two compared metrics. And, even though the NOHZ cases feature more pronounced jitter, these events are significantly rarer than in the generic instances.

The measurements in the figure suggest that no correlation exists between implementation, processing delay, or experiment time. Instead, the impression arises that this behavior appertains to the DuT. For each of the performed experiments, the described intermediate state was observed 9500 times. The number of consecutive packets processed in this state varied and amounted to about 1%, 45%, and 54% for one to three packets, respectively. The P4 specification [38] demands that the P4 program contains a default rule for every table. If no rule is present in the source code, the compiler sets `NoAction` as its default rule. While falling back to the default action is a sensible course of action to take, seeing it used when switching between values can be quite surprising. Such behavior raises security concerns. Active default rules may leak traffic to unexpected destinations. This behavior needs to be taken into consideration during updates involving several switches that may otherwise receive or transmit traffic to or from unexpected destinations and sources. There are two strategies to mitigate the concerns: *(1)* Updating a table's default action via P4Runtime, is a potential mitigation strategy. Following the specification [26], updating

the default entry is permitted if the default action is not marked as constant by the program. *(2)* Another strategy is possible for tables that use longest prefix, ternary, or range matches. Adding a new wildcard entry, which matches all potential keys, can supersede the default action. In this case, only the new wildcard entry will be used instead of the original default action.

Fewer surprises were observed when looking at the control plane delays (d_{control} in Fig. 2) caused by the different implementations. Therefore, we timestamped the reception of a P4Runtime message and its acknowledgment sent after the P4Runtime message was processed. We calculated d_{control} by subtracting the first from the second timestamp. d_{control} does not include the time required by the DuT to apply the modification, i.e., d_{data} in Fig. 2. We observe similar behavior for d_{control} and d_{all}. This similarity is likely rooted in the fact that each implementation's underlying function call blocks until the modify instruction is performed.

6. Reproducibility

We consider reproducibility a key element of research, enabling others to verify and extend our results. Therefore, we publish our experiment artifacts [39] on Zenodo [40], ensuring long-term public availability of our artifacts. The artifacts contain the scripts to create and process the measurement data, including the sample measurement data used for Figs. 4, 5, and 6 in this paper. The experiments were conducted using pos and its methodology [41]. This methodology ensures that experiments can be reproduced automatically on a testbed implementing this methodology.

7. Conclusion

P4-programmable hardware data planes support bandwidths in the Tbit/s-range. The control planes need to keep up with these novel data planes to ensure an effective operation of these high-performance data plane devices. Different flavors of control plane implementations are available: vendor-specific implementations tailored to a specific P4 data plane and the standardized, vendor-independent P4Runtime-based implementations. In this work, we investigated three implementations of P4 control planes for P4 hardware data planes: two vendor-specific and a generic P4Runtime-based implementation. Besides the type of implementation, we investigate the Linux environment hosting the control plane implementations.

Our measurements indicate that a conscious implementation choice is required to optimize performance. To improve latency and jitter of the control plane, the client implementations were ported to DPDK. Out of the three investigated applications, the C++-based vendor-specific API offered significantly lower latency and jitter. We observed differences of several hundred microseconds for P4Runtime-enabled data planes. Hoping to further improve latency and jitter, experiments were repeated on a low-latency Linux kernel. The results of this implementation were mixed: simple packet forwarding was rewarded with reduced jitter, complex packet processing was penalized with higher jitter. We identified the complex architectures of the control plane applications and their dependency on threading as the root cause. We further noticed the reappearance of default rules during table modifications. From this, we conclude that the modify implementation on this device is not implemented in an atomic fashion. The length of the default rule period varies, but in our experiments, it took up to about 200 ns. This kind of non-atomic behavior during table modifications must be considered when rolling out updates across entire networks.

Our results show that high-performance data planes can profit from an optimized control plane. Latency and jitter can significantly impact the traffic processed by the data plane. The P4Runtime implementation should be further improved to achieve performance similar to the vendor-specific implementations. Porting the control plane application to DPDK can help accelerate packet reception and reduce latency and jitter. Initial results indicate that performance could be further improved when switching to a simpler application architecture that can benefit from the low-latency features of the Linux kernel.

CRediT authorship contribution statement

Henning Stubbe: Writing – review & editing, Writing – original draft, Visualization, Validation, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Sebastian Gallenmüller:** Writing – review & editing, Writing – original draft, Methodology, Investigation, Conceptualization. **Manuel Simon:** Writing – review & editing, Writing – original draft, Methodology, Investigation, Conceptualization. **Eric Hauser:** Writing – review & editing, Writing – original draft, Methodology, Investigation, Conceptualization. **Dominik Scholz:** Writing – review & editing, Writing – original draft, Methodology, Conceptualization. **Georg Carle:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation programme as part of the projects SLICES-SC and SLICES-PP (grant agreement no. 101008468 and 101079774). Additionally, we received funding from the Bavarian Ministry of Economic Affairs, Regional Development and Energy as part of the project 6G Future Lab Bavaria. Moreover, this work is partially funded by the German Federal Ministry of Education and Research (BMBF) under the projects 6G-life (16KISK002) and 6G-ANNA (16KISK107) as well as the German Research Foundation (HyperNIC, grant no. CA595/13-1).

References

[1] H. Stubbe, S. Gallenmüller, M. Simon, E. Hauser, D. Scholz, G. Carle, Keeping up to date with P4Runtime: An analysis of data plane updates on P4 switches, in: IFIP Networking Conference, IFIP Networking 2023, Barcelona, Spain, June 12-15, 2023, IEEE, 2023, pp. 1–9, http://dx.doi.org/10.23919/IFIPNETWORKING57963.2023.10186439.

[2] X. Ge, F. Yang, Q. Han, Distributed networked control systems: A brief overview, Inform. Sci. 380 (2017) 117–131, http://dx.doi.org/10.1016/j.ins.2015.07.047.

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, OpenFlow: enabling innovation in campus networks, SIGCOMM Comput. Commun. Rev. 38 (2) (2008) 69–74, http://dx.doi.org/10.1145/1355734.1355746.

[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, D. Walker, P4: programming protocol-independent packet processors, SIGCOMM Comput. Commun. Rev. 44 (3) (2014) 87–95, http://dx.doi.org/10.1145/2656877.2656890.

[5] S. Raza, Y. Zhu, C. Chuah, Graceful network state migrations, IEEE/ACM Trans. Netw. 19 (4) (2011) 1097–1110, http://dx.doi.org/10.1109/TNET.2010.2097604.

[6] J.P. John, E. Katz-Bassett, A. Krishnamurthy, T.E. Anderson, A. Venkataramani, Consensus Routing: The Internet as a Distributed System. (Best Paper), in: USENIX NSDI '08, USENIX Association, 2008, pp. 351–364.

[7] L. Vanbever, S. Vissicchio, C. Pelsser, P. François, O. Bonaventure, Seamless Network-Wide IGP Migrations, SIGCOMM '11, ACM, 2011, pp. 314–325, http://dx.doi.org/10.1145/2018436.2018473.

[8] K. Foerster, S. Schmid, S. Vissicchio, Survey of consistent software-defined network updates, IEEE Commun. Surv. Tutorials 21 (2) (2019) 1435–1461, http://dx.doi.org/10.1109/COMST.2018.2876749.

[9] Intel, Intel® Tofino™ series programmable ethernet switch ASIC, 2022, URL https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html. (Accessed 30 June 2024).

[10] Cisco, Cisco catalyst 9400 series architecture white paper, 2022, URL https://www.cisco.com/c/en/us/products/collateral/switches/catalyst-9400-series-switches/nb-06-cat9400-architecture-cte-en.html. (Accessed 30 June 2024).
[11] Arista, Arista 7050X switch architecture ('a day in the life of a packet'), 2020, URL https://www.arista.com/assets/data/pdf/Whitepapers/Arista_7050X_Switch_Architecture.pdf. (Accessed 30 June 2024).
[12] G. Kurio, L. Wu, I. Wu, V. Vijayanath, Open compute project wedge 400C design specification V1.1, 2022, URL https://www.opencompute.org/documents/wedge400c-ocp-specification-2-pdf. (Accessed 30 June 2024).
[13] H. Frazier, The 802.3z gigabit ethernet standard, IEEE Netw. 12 (3) (1998) 6–7, http://dx.doi.org/10.1109/65.690946.
[14] IEEE Standard for Information technology - Local and Metropolitan Area Networks - Part 3: CSMA/CD Access Method and Physical Layer Specifications - Media Access Control (MAC) Parameters, Physical Layer, and Management Parameters for 10 Gb/s Operation, IEEE Std 802.3ae-2002 (Amendment to IEEE Std 802.3-2002), 2002, pp. 1–544, http://dx.doi.org/10.1109/IEEESTD.2002.94131.
[15] IEEE Standard for Ethernet - Amendment 3: Physical Layer Specifications and Management Parameters for 40 Gb/s and 100 Gb/s Operation over Fiber Optic Cables, IEEE Std 802.3bm-2015, 2015, pp. 1–172, http://dx.doi.org/10.1109/IEEESTD.2015.7069180.
[16] IEEE Standard for Ethernet - Amendment 10: Media Access Control Parameters, Physical Layers, and Management Parameters for 200 Gb/s and 400 Gb/s Operation, IEEE Std 802.3bs-2017 (Amendment to IEEE 802.3-2015 as Amended by IEEE's 802.3bw-2015, 802.3by-2016, 802.3bq-2016, 802.3bp-2016, 802.3br-2016, 802.3bn-2016, 802.3bz-2016, 802.3bu-2016, 802.3bv-2017, and IEEE 802.3-2015/Cor1-2017), 2017, pp. 1–372, http://dx.doi.org/10.1109/IEEESTD.2017.8207825.
[17] IEEE P802.3dj 200 Gb/s, 400 Gb/s, 800 Gb/s, and 1.6 Tb/s ethernet task Force Public Area Channel Data Area, 2023, URL https://www.ieee802.org/3/dj/public/index.html. (Accessed 30 June 2024).
[18] R. Huggahalli, R.R. Iyer, S. Tetrick, Direct cache access for high bandwidth network I/O, in: 32st International Symposium on Computer Architecture (ISCA 2005), 4-8 June 2005, Madison, Wisconsin, USA, IEEE Computer Society, 2005, pp. 50–59, http://dx.doi.org/10.1109/ISCA.2005.23.
[19] S. Gallenmüller, J. Naab, I. Adam, G. Carle, 5G URLLC: A case study on low-latency intrusion prevention, IEEE Commun. Mag. 58 (10) (2020) 35–41, http://dx.doi.org/10.1109/MCOM.001.2000467, Conference Name: IEEE Communications Magazine.
[20] R. Neugebauer, G. Antichi, J.F. Zazo, Y. Audzevich, S. López-Buedo, A.W. Moore, Understanding PCIe Performance for End Host Networking, SIGCOMM '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 327–341, http://dx.doi.org/10.1145/3230543.3230560.
[21] S. Gallenmüller, F. Wiedner, J. Naab, G. Carle, How low can you go? A limbo dance for low-latency network functions, J. Netw. Syst. Manage. 31 (20) (2022) http://dx.doi.org/10.1007/s10922-022-09710-3.
[22] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, D. Walker, Abstractions for Network Update, SIGCOMM '12, ACM, 2012, pp. 323–334, http://dx.doi.org/10.1145/2342356.2342427.
[23] X. Jin, H.H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, R. Wattenhofer, Dynamic scheduling of network updates, in: F.E. Bustamante, Y.C. Hu, A. Krishnamurthy, S. Ratnasamy (Eds.), ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014, ACM, 2014, pp. 539–550, http://dx.doi.org/10.1145/2619239.2626307.
[24] R. Oudin, G. Antichi, C. Rotsos, A.W. Moore, S. Uhlig, OFLOPS-SUME and the art of switch characterization, IEEE J. Sel. Areas Commun. 36 (12) (2018) 2612–2620, http://dx.doi.org/10.1109/JSAC.2018.2871235.
[25] J.H. Han, P. Mundkur, C. Rotsos, G. Antichi, N.H. Dave, A.W. Moore, P.G. Neumann, Blueswitch: Enabling Provably Consistent Configuration of Network Switches, ANCS '15, IEEE Computer Society, 2015, pp. 17–27, http://dx.doi.org/10.1109/ANCS.2015.7110117.
[26] P4runtime specification, 2020, URL https://p4.org/p4runtime/spec/v1.3.0/P4Runtime-Spec.html.
[27] gRPC Authors, gRPC a high performance, open source universal RPC framework, 2023, URL https://grpc.io. (Accessed 29 December 2023).
[28] C.H. Song, X.Z. Khooi, D.M. Divakaran, M.C. Chan, Revisiting application offloads on programmable switches, in: IFIP Networking Conference, IFIP Networking 2022, Catania, Italy, June 13-16, 2022, IEEE, 2022, pp. 1–9, http://dx.doi.org/10.23919/IFIPNetworking55013.2022.9829799.
[29] C. Zeng, L. Luo, T. Zhang, Z. Wang, L. Li, W. Han, N. Chen, L. Wan, L. Liu, Z. Ding, X. Geng, T. Feng, F. Ning, K. Chen, C. Guo, Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing, in: USENIX NSDI '22, USENIX Association, 2022, pp. 1345–1358.
[30] J. Salim, When NAPI comes to town, in: Proceedings of Linux 2005 Conference, UK, 2005.
[31] DPDK Project, Data plane development kit, 2022, URL https://www.dpdk.org/. (Accessed 30 June 2024).
[32] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, G. Carle, MoonGen: A Scriptable High-Speed Packet Generator, IMC '15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 275–287, http://dx.doi.org/10.1145/2815675.2815692.
[33] F. Reghenzani, G. Massari, W. Fornaciari, The real-time linux kernel: A survey on preempt_rt, ACM Comput. Surv. 52 (1) (2019) 18:1–18:36, http://dx.doi.org/10.1145/3297714.
[34] NO_HZ: Reducing scheduling-clock ticks, 2023, URL https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt. (Accessed 30 June 2024).
[35] STORDIS GmbH, Technical specifications of the stordis BF2556X-1T-A1F, 2019, URL https://www.stordis.com/wp-content/uploads/2019/12/STORDIS_BF2556X-1T-A1F.pdf.
[36] endace, Datasheet endace DAG 10x4-S, 2023, URL https://web.archive.org/web/20180905043442/https://www.endace.com/dag-10x4-s-datasheet.pdf. (Accessed 30 June 2024).
[37] G. Tene, HdrHistogram: A high dynamic range histogram, 2023, URL http://hdrhistogram.org/. (Accessed 30 June 2024).
[38] P4_16 language specification, 2022, URL https://p4.org/p4-spec/docs/P4-16-v-1.2.3.html.
[39] H. Stubbe, S. Gallenmüller, D. Scholz, M. Simon, E. Hauser, G. Carle, Measurement artifacts, 2023, http://dx.doi.org/10.5281/zenodo.7871012.
[40] European Organization For Nuclear Research, OpenAIRE, Zenodo, 2013, http://dx.doi.org/10.25495/7GXK-RD71, URL https://www.zenodo.org/. (Accessed 30 June 2024).
[41] S. Gallenmüller, D. Scholz, H. Stubbe, G. Carle, The Pos Framework: a Methodology and Toolchain for Reproducible Network Experiments, CoNEXT '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 259–266, http://dx.doi.org/10.1145/3485983.3494841.