Keeping up to Date with P4Runtime: An Analysis of Data Plane Updates on P4 Switches

Henning Stubbe, Sebastian Gallenmüller, Manuel Simon, Eric Hauser, Dominik Scholz, Georg Carle
TUM School of Computation, Information and Technology, *Technical University of Munich*
Garching near Munich, Germany
{ stubbe | gallenmu | simonm | hauser | scholz | carle }@net.in.tum.de

***Abstract*—The continuous increase of achievable data rates in computer networks is both blessing and curse. Increasing data rates enable novel applications through higher bandwidths. However, support for higher data rates requires devices to process packets reliably in an ever-decreasing amount of time per packet. In terms of software-defined networking: higher data rates call for a faster data plane. Nevertheless, the control plane must not be ignored; to faithfully react to data plane behavior, a high-performance control plane is essential. Otherwise, e.g., the data plane's state cannot be updated fast enough to cope with fast-paced traffic changes. In this case study, we investigate the control plane of a high-performance P4 switching ASIC. Moreover, we create a measurement methodology to track the delay between the reception of a rule update on the control plane and its actual application on the data plane of a P4 hardware switch. By applying the methodology to said ASIC, we can precisely describe its performance and non-atomicity in updates. Based on our findings, we apply multiple different approaches to optimize control plane latency. Our results highlight the need to consider latency on the control plane proportionate with the increase of achievable data rates.**

***Index Terms*—Reproducibility, Network Experiments, P4, P4-Runtime, Control Plane**

I. Introduction

Among the intriguing changes in networks in recent years is the continued growth of achievable data rates and the improved programmability of network devices. Combined with latency guarantees, the bandwidth growth and increased flexibility enable novel applications. Examples include distributed control processes in the area of transportation, industry, and medicine [1]. The improved programmability of the network, e.g., through OpenFlow [2] or P4 [3], can help tailor the network toward these novel applications. Consequently, such applications with custom protocols and the popularity of programmable network devices thrive.

As with OpenFlow, match-action tables are a crucial element of P4-programmed network devices. The table configuration defines a network device's behavior, matching patterns in packet headers with executed actions. Table entries can be modified at runtime to adapt the behavior of the device. This transition from an old table state to a new one can cause harmful delay or packet loss [4]–[6]. Higher data rates exacerbate this problem, as more flows or packets can be affected by these transient states. Similarly, execution time reliability is essential. The roll-out of table updates may involve several devices. Carefully scheduled and deterministically executed match-action table modification instructions may be used to counteract undesired but unavoidable delays. To orchestrate such distributed updates, the update behavior and latency of individual devices must be known. However, latency prediction is aggravated with table updates being processed in software that are subjected to random interrupts or short-time overload.

In this paper, we present a measurement methodology to investigate the behavior of P4 switches during table updates. 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 paper 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 paper is structured as follows. Section II gives background information on the architecture of P4 switches and describes the procedure for data plane updates on a single device. Section III investigates the state of the art for control planes, with a focus on performance in the context of P4. Afterward, Section IV introduces the experiment setup used throughout this work. Subsequently, Section V evaluates the behavior of an exemplary P4-programmable ASIC. Section VII concludes the paper.

II. Background

To understand a system's behavior, background knowledge of its underlying architecture is valuable. This section presents the typical structure of today's switch architecture. Based on this architecture, we describe the procedure of updating the

ISBN 978-3-903176-57-7

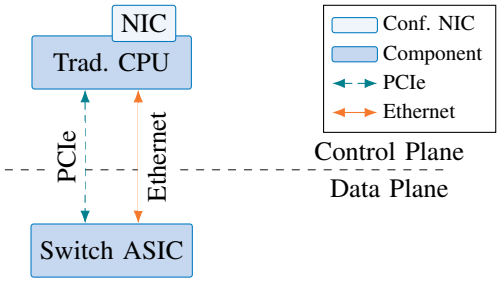

Figure 1: High-level switch architecture

data plane with a particular focus on latency. Afterward, we provide a case study on the impact of control plane latency.

A. Router Architecture

The typical architecture of current networking devices, cf. Figure 1, splits the device's tasks and assigns the resulting subtasks to different components. 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 of multiple vendors, such as Intel [7], Cisco [8], and Arista [9], or the open hardware design of the Wedge400 [10]. 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 a versatile but slow CPU with the power of a restricted but performant ASIC. The communication between these two components is essential, to ensure CPU and ASIC work together as a single switch. Here, two popular options for communication channels are observable, 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. Figure 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. Therefore, a single control plane update message results in a potentially complex process until it is applied.

Table I: 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
P802.3df [17]	1600	0.4	2500

B. Case Study: Control Plane Latency

We want to determine a lower bound for the update latency caused by the control plane. This update process involves the reception of a P4Runtime message on the control interface of the control plane, the PCIe transfer from NIC to RAM, the processing of the message on the CPU itself, and, finally, the PCIe transfer of the update to the ASIC.

Neugebauer et al. [11] measured a median round trip time of 800 ns for a minimum-sized packet of 64 B across the PCI express bus. Their measurements do not involve any processing of the packet data on the CPU, e.g., in the driver. Gallenmüller et al. [12] measured a median latency of 3.3 µs between the ingress and egress interface of a simple packet forwarding application. This latency additionally includes the processing of data on the CPU (approx. 100 clock cycles) and additional latency for accessing the data. The underlying hardware (NIC, x86 CPU) 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 microseconds.

Table I shows the packet serialization delay for minimum-sized Ethernet packets for 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, we assume a control plane with a a 1-µs delay. In such a control plane, two minimum-sized packets will pass a 1-Gbit/s data plane before a received control plane update is delivered to the switching ASIC. While a low number of impacted packets may be considered negligible, their number grows for higher bandwidths (cf. Table I).

The numbers reported in Table I only consider the best-case scenario. If we assume the higher control plane delay of 3.3 µs, the numbers are multiplied by a factor of 3.3. With reported worst-case latencies of 1 ms and more (cf. Gallenmüller et al. [12], [18]), the number of impacted packets can grow 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.

III. Related Work

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

a) Switches: Updating the forwarding rules of running networks can cause unwanted side effects if partially old and new configurations are applied to specific packets [4]–[6]. To avoid these transient states between updates and their impact, Reitblatt et al. [19] 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. OFLOPS-SUME [20] 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. [21] 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.

b) P4: P4 [3] is a domain-specific language to program the data plane, which supports programming different types of data plane devices, such as switches. P4Runtime [22] 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 [23], 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 [7]. Song et al. [24] 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. [25] 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.

c) Software-based Control Planes: Recalling the prevalent switch architecture (cf. Figure 1) 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. [12], [18] 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 [26] 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 [27]. DPDK relies exclusively on polling

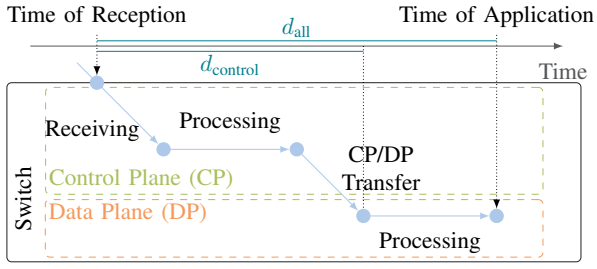

Figure 2: Switch update delay (d) for forwarding and processing of control plane messages

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`. Other projects also picked up on the idea of increasing DPDK's ease of use, e.g., the packet generator MoonGen [28]. The preemptive nature of the Linux kernel allows interrupting running processes introducing jitter. A study by Reghenzani et al. [29] investigates real-time patches available for the Linux kernel that create a more stable and predictable behavior. The tickless Linux kernel [30] further improves predictability and low-latency behavior for applications by disabling scheduling interrupts (also called *ticks*) on specific CPU cores.

Data plane updates on switches may cause unwanted effects for packets processed during transient states. When offloading applications to P4 data plane elements, these effects 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 to also investigate individual data plane updates and their impact. In addition, we consider delays that may be introduced by the control plane OS.

IV. Measurement Methodology

Determining the impact of table updates on the control plane for packet processing on the data plane requires a specific measurement approach observing both planes. In this section, we introduce the challenges of synchronizing and measuring the test traffic between data and control plane. To perform experiments with expressive results, we further present a suitable measurement setup.

A. Challenges

Figure 2 shows the delay of a table update beginning with the time of reception on the control plane and the actual time of application for the table update on the data plane. We refer to the total delay as d_{all}. Initially, a table update message is received on the control plane, processed, and transferred to the data plane. We refer to this control plane delay as d_{control}. During d_{control}, the message is handled in software and, hence, subjected to the jitter and delay caused by the OS, i.e., the Linux kernel. After the reception on the data plane, the message is applied to the switching ASIC. We consider the entire delay d_{all} from reception on the control plane to

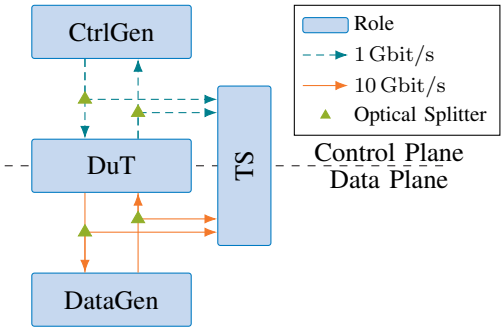

Figure 3: Measurement setup overview

application on the data plane as the time of transition between the old and the new state. The main focus of our investigation is delay and jitter during this time of transition. Additionally, we are interested in the events on the data plane during this timeframe, such as packet drops or partially applied updates.

Typically, data plane and control plane traffic is received on separate interface ports. Control plane ports are directly connected to the CPU hosting the control plane software (cf. Figure 1). The data plane utilizes ports attached to the switching ASIC. To accurately determine the delay between the control and data plane ports, we need a method to synchronize the control and data plane traffic. Given the continued downward trend in serialization times (cf. Table I), a highly precise and reliable measurement setup is essential.

Jitter introduced by software interrupts, or PCI express transfers, hamper accurate and precise latency measurements in software. Therefore, we prefer timestamping hardware that timestamps received packets early in the processing paths, completely preempting elements causing jitter-introducing effects. To avoid additional inaccuracies, introduced by synchronizing different clocks between the control and data plane, we use the same measurement device, i.e., the same clock, for both traffic streams.

B. Setup

The setup (cf. Figure 3) used in this work 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 a 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.

Table II: Hardware components of our measurement setup

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

Both load generators employ MoonGen [28] as packet generator. They run on an Intel Xeon E5-1650 equipped with 128 GB memory and an Intel X710 NIC. The DuT consists of a P4-programmable ASIC combined with an Intel Xeon D-1548 equipped with 32 GB memory and an Intel I350 NIC on the control plane. 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 [31] 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 a line rate of 10 Gbit/s with a resolution of 4 ns. Table II 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.

C. Experiment Description

We expect a significant performance impact of the control plane, the control plane application, and its host configuration. Therefore, we discriminate our experiments based on the P4-Runtime implementation used and the DuT host configuration. Options for the former are discussed later in this section, starting with Paragraph b). The latter is either the vendor's default configuration or a configuration optimized for low latency. Here, the low-latency optimized configurations build on experiences from previous work [12], [18].

Akin to all experiments is their 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, processing consists of consulting a P4 match-action table, which specifies how to update the packet's Ethernet source address. Throughout the experiment, the value of the Ethernet source address is 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 in- and outgoing packets for both load generators.

Based on this observation of in- and outgoing packets, the processing delay 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 evaluate the processing latency induced by the DuT, during an experiment, the CtrlGen transmits instructions at a constant rate. Minus the initial warm-up phase, all observed processing delays are recorded by the TS for later evaluation.

a) 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., Implementations (F2) and (F3), have an advantage compared to the interpreted Python implementation (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.

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

Given that both Implementations (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., Implementations (I2) and (I3), compared to the interpreted implementation is expected.

c) P4Runtime Table Manipulation Operation: Following the P4Runtime specification [22], a limited number of operations can be performed on tables: 1) new entries can be added via *insert*; 2) present entries may be *remove*d; 3) alternatively, *modify* allows to replace existing entries. 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, this work focuses on the modify operation.

V. 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.

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.

A. 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) 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 that 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.

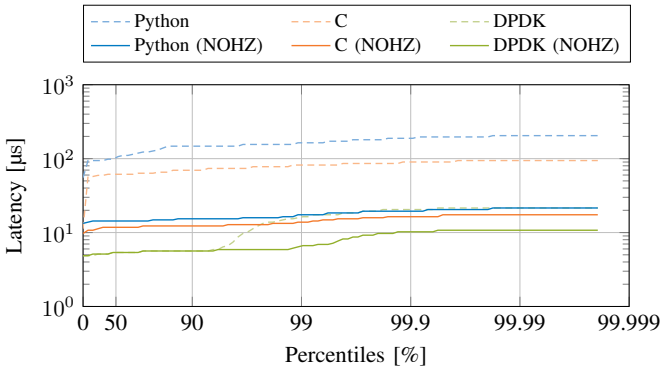

Figure 4: Baseline measurements for the processing latency of the control plane

a) Impact of the Programming Language: We have chosen two forwarders, written in Python and C, to demonstrate the impact 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. Figure 4 shows the results of the Python and C forwarder as dashed lines. We use a percentile distribution graph to visualize the measured latency [32]. 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 significantly less for the high percentiles of the C forwarder.

b) 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 Figure 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.

c) 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 [30], i.e., with this option enabled, and relevant DuT tasks pinned to isolated CPUs, as previously suggested [12], [18]. To utilize the features of this kernel, we compiled a new Linux kernel with the flag `CONFIG_NO_HZ_FULL` enabled. In this paper, we refer to that kernel as the *NOHZ* environment. The NOHZ kernel runs processes with a lower jitter on dedicated cores by disabling typical OS interrupts, such as the scheduling interrupts. With scheduling interrupts disabled, 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. Figure 4 visualizes these latencies using solid lines. We see significant improvements for all three forwarders. For the Python (median: 14 µs, 99.99th percentile: 17 µs) and C (median: 12 µs, 99.99th percentile: 22 µs) forwarders, 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.

B. 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 Figure 5 as a percentile distribution. The figure depicts the recorded receive time differences, i.e., the DuT's processing delay of the three investigated P4Runtime implementations, emphasizing higher percentiles.

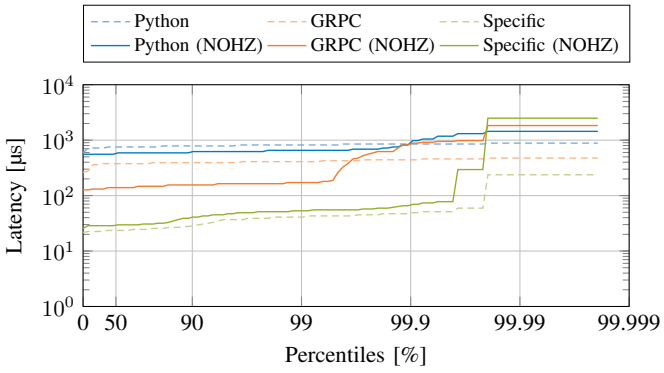

Figure 5: Percentile distribution of processing latency

a) Python-based Implementation (I1): From the results shown in Figure 5 follows 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.

b) 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) 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.

d) 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 I shows that for 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. But, as discussed, this only partially holds 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.

C. 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.

Figure 6 compares the observed processing latency and the number of times the default action was applied to a packet throughout of 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 flier 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.

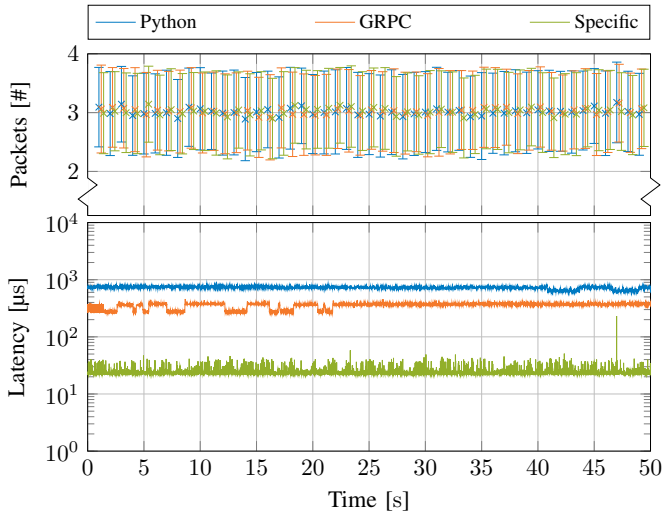

Figure 6: Per-second mean packet sequence length subjected to default action (outliers show standard error; top) vs. processing latency over time (bottom)

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 Figure 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 cases.

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 [33] 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.

Fewer surprises were observed when looking at the control plane delays (d_{control} in Figure 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., $d_{\text{all}} - d_{\text{control}}$ in Figure 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.

VI. Reproducibility

We consider reproducibility to be a key element of research, enabling others to verify and extend our results. Therefore, we publish our measurement scripts and the results of their execution. [34] For the measurements, we used pos and its methodology. [35] Further information on the experiment execution and evaluation are provided as well.

VII. Conclusion

P4-programmable switches tap into the domain of terabit networks. These higher packet rates on the data plane imply the rising importance of fast control actions. To unify the control of P4 devices, P4Runtime was established—a control plane API with growing popularity. This work looked at the control plane performance of three implementations on a P4-programmable ASIC, with a special focus on the Linux environment hosting the control plane applications.

Results indicate that a conscious implementation choice is required to minimize overall system delay. 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 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.

Bottom line is, a fast data plane needs a fast control plane. We argue that the development of control planes, especially the P4Runtime implementations, needs closer attention focusing on crucial latency and jitter performance. Our results show that control plane performance can be significantly improved by accelerating the packet reception and choosing an optimized control plane application. Optimizing the control application offers further untapped potential for improvement.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation programme (grant agreement no. 101008468 and 101079774). Additionally, we received funding by 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 Germany Federal Ministry of Education and Research (BMBF) under the projects 6G-life (16KISK001K) and 6G-ANNA (16KISK107) as well as the German Research Foundation (HyperNIC, grant no. CA595/13-1).

References

[1] X. Ge, F. Yang, and Q. Han, "Distributed networked control systems: A brief overview," *Inf. Sci.*, vol. 380, pp. 117–131, 2017. DOI: 10.1016/j.ins.2015.07.047.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008. DOI: 10.1145/1355734.1355746.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014. DOI: 10.1145/2656877.2656890.

[4] S. Raza, Y. Zhu, and C. Chuah, "Graceful network state migrations," *IEEE/ACM Trans. Netw.*, vol. 19, no. 4, pp. 1097–1110, 2011. DOI: 10.1109/TNET.2010.2097604.

[5] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. E. Anderson, and A. Venkataramani, "Consensus routing: The internet as a distributed system. (best paper)," ser. USENIX NSDI '08, USENIX Association, 2008, pp. 351–364.

[6] L. Vanbever, S. Vissicchio, C. Pelsser, P. François, and O. Bonaventure, "Seamless network-wide IGP migrations," ser. SIGCOMM '11, ACM, 2011, pp. 314–325. DOI: 10.1145/2018436.2018473.

[7] Intel, *Intel® Tofino™ Series Programmable Ethernet Switch ASIC*, en, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html (visited on 02/13/2023).

[8] Cisco, *Cisco Catalyst 9400 Series Architecture White Paper*, Mar. 2022. [Online]. Available: https://www.cisco.com/c/en/us/products/collateral/switches/catalyst-9400-series-switches/nb-06-cat9400-architecture-cte-en.html (visited on 02/13/2023).

[9] Arista, *Arista 7050X Switch Architecture ('A day in the life of a packet')*, 2020. [Online]. Available: https://www.arista.com/assets/data/pdf/Whitepapers/Arista_7050X_Switch_Architecture.pdf (visited on 02/13/2023).

[10] G. Kurio, L. Wu, I. Wu, and V. Vijayanath, *Open Compute Project Wedge 400C Design Specification V1.1*, Jan. 2022. [Online]. Available: https : / / www . opencompute . org / documents / wedge400c - ocp - specification-2-pdf (visited on 02/13/2023).

[11] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding PCIe performance for end host networking," ser. SIGCOMM '18, New York, NY, USA: Association for Computing Machinery, Aug. 2018, pp. 327–341, ISBN: 978-1-4503-5567-4. DOI: 10.1145/3230543.3230560.

[12] S. Gallenmüller, J. Naab, I. Adam, and G. Carle, "5G URLLC: A Case Study on Low-Latency Intrusion Prevention," *IEEE Communications Magazine*, vol. 58, no. 10, pp. 35–41, Oct. 2020, Conference Name: IEEE Communications Magazine, ISSN: 1558-1896. DOI: 10.1109/MCOM.001.2000467.

[13] H. Frazier, "The 802.3z Gigabit Ethernet Standard," *IEEE Netw.*, vol. 12, no. 3, pp. 6–7, 1998. DOI: 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)*, pp. 1–544, 2002. DOI: 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*, pp. 1–172, 2015. DOI: 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)*, pp. 1–372, 2017. DOI: 10.1109/IEEESTD.2017.8207825.

[17] *IEEE P802.3df 200 Gb/s, 400 Gb/s, 800 Gb/s, and 1.6 Tb/s Ethernet Task Force. Tools and Channel Data Area*. [Online]. Available: https://www.ieee802.org/3/df/public/tools/index.html (visited on 02/13/2023).

[18] S. Gallenmüller, F. Wiedner, J. Naab, and G. Carle, "How Low Can You Go? A Limbo Dance for Low-Latency Network Functions," *Journal of Network and Systems Management*, vol. 31, no. 20, Dec. 2022, ISSN: 1573-7705. DOI: 10.1007/s10922-022-09710-3.

[19] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," ser. SIGCOMM '12, ACM, 2012, pp. 323–334. DOI: 10.1145/2342356.2342427.

[20] R. Oudin, G. Antichi, C. Rotsos, A. W. Moore, and S. Uhlig, "OFLOPS-SUME and the art of switch characterization," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 12, pp. 2612–2620, 2018. DOI: 10.1109/JSAC.2018.2871235.

[21] J. H. Han, P. Mundkur, C. Rotsos, G. Antichi, N. H. Dave, A. W. Moore, and P. G. Neumann, "Blueswitch: Enabling provably consistent configuration of network switches," ser. ANCS '15, IEEE Computer Society, 2015, pp. 17–27. DOI: 10.1109/ANCS.2015.7110117.

[22] *P4runtime specification*, P4.org API Working Group, Dec. 2020. [Online]. Available: https://p4.org/p4runtime/spec/v1.3.0/P4Runtime-Spec.html.

[23] gRPC Authors, *gRPC A high performance, open source universal RPC framework*. [Online]. Available: https://grpc.io (visited on 02/13/2023).

[24] C. H. Song, X. Z. Khooi, D. M. Divakaran, and 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. DOI: 10.23919/IFIPNetworking55013.2022.9829799.

[25] 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, and C. Guo, "Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing," ser. USENIX NSDI '22, USENIX Association, 2022, pp. 1345–1358.

[26] J. Salim, "When NAPI Comes to Town," in *Proceedings of Linux 2005 Conference, UK*, 2005.

[27] DPDK Project, *Data Plane Development Kit*, 2022. [Online]. Available: https://www.dpdk.org/.

[28] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," ser. IMC '15, New York, NY, USA: Association for Computing Machinery, Oct. 2015, pp. 275–287, ISBN: 978-1-4503-3848-6. DOI: 10.1145/2815675.2815692.

[29] F. Reghenzani, G. Massari, and W. Fornaciari, "The Real-Time Linux Kernel: A Survey on PREEMPT_RT," *ACM Comput. Surv.*, vol. 52, no. 1, 18:1–18:36, 2019. DOI: 10.1145/3297714.

[30] n.a., *NO_HZ: Reducing Scheduling-Clock Ticks*. [Online]. Available: https://www.kernel.org/doc/Documentation/timers/NO%7B%5C_%7DHZ.txt (visited on 02/13/2023).

[31] endace, *Datasheet Endace DAG 10X4-S*. [Online]. Available: https://web.archive.org/web/20180905043442/https://www.endace.com/dag-10x4-s-datasheet.pdf (visited on 02/13/2023).

[32] G. Tene, *HdrHistogram: A High Dynamic Range Histogram*. [Online]. Available: http://hdrhistogram.org/ (visited on 02/13/2023).

[33] *P4_16 language specification*, P4.org API Working Group, Jul. 2022. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v-1.2.3.html.

[34] H. Stubbe, S. Gallenmüller, D. Scholz, M. Simon, E. Hauser, and G. Carle, *Measurement artifacts*, Zenodo, Jun. 2023. DOI: 10.5281/zenodo.7871012.

[35] S. Gallenmüller, D. Scholz, H. Stubbe, and G. Carle, "The pos framework: A methodology and toolchain for reproducible network experiments," ser. CoNEXT '21, New York, NY, USA: Association for Computing Machinery, Dec. 2021, pp. 259–266, ISBN: 978-1-4503-9098-9. DOI: 10.1145/3485983.3494841.