P8: P4 with Predictable Packet Processing Performance

Hasanin Harkous*†, Michael Jarschel†, Mu He*, Rastin Pries†, Wolfgang Kellerer*
*Technical University of Munich, firstname.lastname@tum.de
†Nokia Bell Labs, firstname.lastname@nokia.com
Munich, Germany

***Abstract*—Data plane programmability brings network flexibility to a new level. However, it introduces the complexity of the data path's program as a new factor that influences packet forwarding latency and thus devices' performance. Accurate identification of the relation between data path complexity and packet forwarding latency enables the design and management of networks with predictable performance.**

In this paper, we leverage the characteristics of P4 programming language to provide a method for estimating the packet forwarding latency as a function of the data path program. We analyze the impact of different P4 constructs on packet processing latency for three state-of-the-art P4 devices: Netronome SmartNIC, NetFPGA-SUME, and T4P4S DPDK-based software switch. Besides comparing the performance of these three targets, we use the derived results to propose a method for estimating the average packet latency, at compilation time, of arbitrary P4-based network functions implemented using the surveyed P4 constructs. The proposed method is finally validated using a set of realistic network functions, which shows that our method estimates the average packet latency with sub-microsecond precision.

***Index Terms*—Performance Evaluation, Programmable Data Plane, P4, Software-Defined Networking, Modeling, Device Benchmarking.**

I. Introduction

THE introduction of programmability into networks has many advantages in terms of flexibility, operational cost, etc., compared to legacy network forwarding devices that work under the restriction of fixed packet processing pipelines [1], [2]. Software-Defined Networking (SDN) [3] enhanced network programmability, making it possible to customize packet forwarding based on specific header fields [2]. Network flexibility [1] was further increased with the introduction of data plane programmability [4], enabling the customization of packet processing pipelines in forwarding devices. There are multiple network devices [5]–[8] available in the market that support data plane programmability through different programming languages such as C, micro C, POF [8], and P4 [9], i.e., the language which we focus on in this paper.

Data plane programmability enables a large variety of new services such as in-network computation, load balancing, resilient and efficient forwarding, and telemetry. In-network computation offers the possibility to offload path aggregation, caching, or even AI/ML to the network. Efficient load balancing and forwarding not only need programmable data planes but also heavily rely on detailed measurements and monitoring. Telemetry helps here by offering deep insights into the network and partly into end hosts as well. Data plane programmability can thus not only be applied to pure data centers, but also offers new possibilities in cellular and campus networks.

There exists a trade-off between network performance and flexibility as well as a considerable cost factor. Performance and efficiency have always been key and in the past were achieved through fixed forwarding pipelines with limited flexibility. Over time, this lack of flexibility became a burden as deployed hardware could not be easily upgraded with new features or used for any other than its original purpose. Therefore, the idea to run network functions as software on commodity servers was introduced but faced its own set of challenges. While software is very flexible, the server platform underneath usually cannot offer the same performance level as purpose-built hardware or only at a significantly increased cost. Data plane programmability is positioned as a solution to augment these two extremes: Offering reliable high-performance networking with significant functional flexibility as is required in the hyper-dynamic networks of the future. However, this opens up a new problem space for future network operators that rely on programmable hardware as well as software solutions: placement and scheduling. The P4 language is envisioned to simplify this task by making network functions platform independent, i.e., the functionality remains the same, independent of where the function is run. Therefore, it becomes key to understand the performance of P4 and its building blocks (P4 constructs) on different platforms (P4 targets) to make a correct and ideally automated decision on which platform to execute a specific network function in a given scenario, e.g., when two platforms are available for two virtual networks used by different verticals in the same vicinity with non-complementary requirements. Choosing the right platform for the respective network functions and adapting the decision based on demand may be the difference between happy customers and an SLA-violation, particularly in resource-constrained edge locations with low-latency requirements on the packet processing pipeline.

While data plane programmability is promising for the previously mentioned reasons, the performance of P4-enabled packet processors depends on the **complexity** of the programmed data plane pipeline. Specifically, the impact of programmable data plane complexity on the packet processing latency of different network functions, e.g., L3 forwarding and VxLAN tunneling, has to be understood.

Understanding the relation between packet forwarding latency and the pipeline complexity helps to characterize the forwarding performance of programmable networking devices. The latter enables operators to better manage their networking infrastructure and provide performance guarantees (e.g., latency guarantees), for a known traffic flow, to their tenants — an essential feature in data centers or 5G networks [10]. Moreover, knowing the forwarding latency of networking devices is the first building block for deriving analytical models of the performance of these devices as in queuing theory. Additionally, a priori knowledge of the packet latency of running different network functions on different programmable devices is critical to optimally schedule and provision network functions in a hybrid environment [11], [12].

Towards identifying the relation between P4 pipeline complexity and the forwarding latency of packet processors, we consider the following reasoning. Measuring the forwarding performance of running every possible network function, written as a P4 program, on every possible P4 device is clearly not an option. Hence, we consider the basic set of P4 constructs, such as parsing, header operations, and tables, as the **basis** that can be used to build arbitrary P4-based network functions. Therefore, by analyzing the impact of the basic set of P4 constructs on packet processing latency, we can estimate the latency of realistic network functions that span the measured P4 constructs.

The impact of the basic set of P4 constructs on packet latency was extensively measured on three state-of-the-art P4 programmable devices: NetFPGA-SUME card [5], Netronome SmartNIC [6], and T4P4S DPDK-based Software Switch [7]. More than 75 P4 pipelines, each written in three versions to be compatible with the investigated targets, were carefully designed and measured to quantify the impact of different P4 constructs on packet latency. This paper extends our previous work [13] which estimates the latency of a single P4 device based on a limited number of P4 constructs with a simple technique. The contributions of this paper are twofold:

1) Measure, compare, and analyze the impact of different P4 constructs on packet latency of three state-of-the-art P4 targets.
2) Propose a method for estimating the packet latency of realistic network functions based on the given P4 program using the results of the measurement campaign conducted on atomic P4 constructs.

In our evaluation, we focus on P4 as a programming language for being a domain-specific language designed to program packet processors, and for its increasing popularity in the research community. Moreover, the characteristics of the P4 language enable the estimation of the packet latency. These characteristics include *(i)* Abstraction of the programming of packet processing pipelines into a limited set of constructs. *(ii)* Prevention of loops with unknown iteration counts, and dynamic memory allocation to limit performance variation at run-time [14].

The paper is organized as follows. Section II contains relevant information on P4 language and its targets. In Section III, the measurement testbed, and the designed experiments are described. The measurement results are reported and analyzed in Section IV. Section V explains and validates the proposed packet latency estimation method. Related works are discussed in Section VI. Section VII concludes the paper.

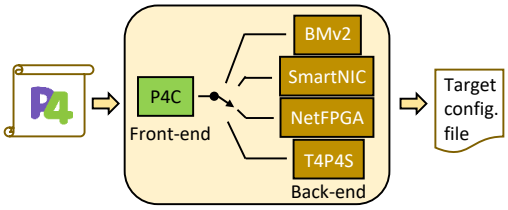


Fig. 1: P4 compiler design: All P4 targets share the same front-end compiler but have different back-end compiler.

II. Background

In this section, we provide relevant background information about the P4 language and its basic constructs, the P4 compilation process, and the available P4 programmable devices.

A. P4 Language

P4 [15] is a domain-specific language designed to program packet processors. This data plane program specifies which header information to parse, how to match on the parsed header, and what actions to take when a hit takes place. P4 adopts an abstract model for the packet processing pipeline which is made up of a basic set of constructs [9] that are described in the following.

Header type defines the format of each header in a packet as a set of fields and their respective sizes. **Parser** works as a Finite State Machine (FSM) and describes the sequence in which headers are extracted from a packet when it arrives into the packet processing pipeline. The FSM ensures that different packets can have different headers extracted and stored into the parsed-headers-stack.

Control flow specifies an invoke sequence of **match-action units** that operate on the extracted headers stored in the parsed-headers-stack. Each match-action unit does the following three steps: (1) builds lookup keys from packet header fields and run-time metadata, (2) performs lookup with the keys in the table and choose the associated action and input data, and (3) executes the chosen action. An **action** is made up of a set of operations that can modify fields of a header, or manipulate the parsed-headers-stack by copying, adding, and removing headers.

After traversing the control blocks, the packet reaches the **deparser** stage. The deparser reassembles the packet through attaching the manipulated parsed-headers-stack (including any newly inserted header) back to the payload and pushing it to an egress queue, if not dropping it. **Extern objects** are architecture-specific constructs (e.g., header checksum and encryption units) that can be invoked via pre-defined APIs. Their internal behavior is hard-coded and therefore cannot be programmed by P4 [9].

B. P4 Compilation

To enable target-independence and ease of future upgrades, P4 compilers are divided into a common *front-end* compiler and a target-specific *back-end* compiler as shown in Fig. 1.

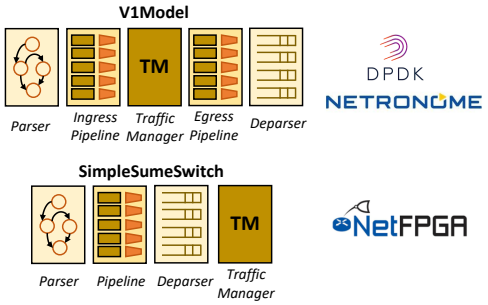


Fig. 2: Two architectures that are used by the studied P4 targets in this work. Their difference resides in the order of stages, i.e., parser, match-action units (ingress/egress pipeline), traffic manager and deparser.

The common open-source front-end compiler (i.e., P4C [16]) is provided by the P4 community which transforms a P4 program into an Intermediate Representation (IR) in the form of JSON consisting the definitions of parser, tables, and actions. P4C provides some basic implementations of optimizations (e.g., dead state elimination and constant propagation) [17].

The back-end compiler, on the other hand, is provided by the vendor who designs the target. It generates a configuration file to program the target's behavior, as well as the control plane APIs to update its state (e.g., table entries and register values) at run-time. Based on the properties of the target, the back-end compiler implements all atomic operations of P4 in a compatible way. Additionally, the back-end compiler may perform optimizations to enhance the packet processing performance of the target. In our analysis, we measure the packet forwarding latency of a P4 target considering all the optimizations of the back-end compiler as a part of the tested P4 target, i.e., the P4 target and its back-end compiler define the device under test.

C. P4 Targets

A P4 *target* is a system that can execute a P4 program and process data packets.

The architecture serves as a template of a pipeline structure for a P4 target. Fig. 2 illustrates the two most common architectures, i.e., *V1Model* and *SimpleSumeSwitch*. For the measurements, we create different versions of P4 programs with the same set of operations on parsers and match-action units following the respective architectures of different targets.

P4's target-independence feature enables different types of both software and hardware networking devices to be programmed via P4. The software family includes BMv2 [18] and T4P4S [7] targets, and the hardware family includes different targets with different hardware platforms.

As the first software prototype switch of P4, BMv2 is mainly used for proof-of-concept implementations with limited performance in terms of processing rate, i.e., around 1 Gbps. This work focuses on state-of-the-art P4 targets that can process packets at high rates, therefore does not consider BMv2. We evaluate the performance of three P4 targets that have different processing platforms. In the following we provide details about these investigated targets:

Netronome SmartNIC: Netronome SmartNIC is a Network Processor Unit (NPU) with tens of multi-threaded purpose-built cores that enable high parallelism. Hierarchical transactional memory and built-in accelerators in the device also boost the packet processing capability. The P4 architecture used by Netronome SmartNIC is the V1Model. The back-end compiler generates a C implementation of the datapath, which is then used to create the firmware for the SmartNIC. It takes a few minutes to build from a P4 source code and load the firmware. [19] presents further details about Netronome SmartNIC's hardware design and programmability.

T4P4S DPDK-based Software Switch: Data Plane Development Kit (DPDK) [20] is an open-source framework for accelerating packet processing on x86, ARM and PowerPC processors. It accelerates the packet processing of software switches through the following features. First, the incoming packets of all flows are distributed to all CPU cores that are reserved for the switch. Second, the data path between the interfaces and the processing threads only lies in user space, thus, saving unnecessary copying of packets between user and kernel spaces. Third, to better use the CPU's cache, it processes packets in batches.

T4P4S [7] is a compiler that turns a P4 code into a target-independent C core program that can run on top of the DPDK framework to execute the designed P4 datapath. The compilation process usually takes a few minutes. By default, two CPU cores (one as master and the other as slave) are reserved for packet processing with NUMA mode enabled. T4P4S also uses the V1Model architecture.

NetFPGA-SUME: With the increasing specialization of packet processors, Field Programmable Gate Array (FPGA) is a promising alternative that satisfies the requirements of low-latency, high-throughput concurrent packet processing. NetFPGA-SUME [5] enables easier programmability on FPGAs by leveraging the P4→NetFPGA workflow [21].

Hardware resources on an FPGA are represented mainly as its number of slices, the number of on-chip memories (Block RAMs). Memory resources are used for different types of matching: TCAM for ternary matches, and SRAM for hash-based exact matches. The back-end compiler generates a Register-Transfer Level (RTL) implementation of the datapath, which is then synthesized to a bitstream to program the FPGA chip. The whole process takes about one hour. The P4 architecture of NetFPGA is SimpleSumeSwitch with only one control flow, whereas the V1Model consists of two control flows abstracted as ingress and egress stages.

III. Testbed & Experiment Design

In this section, we describe the measurement testbed setup, shown in Fig. 3, and the experiments designed to evaluate the latency cost of different P4 constructs.

We perform the experiments using two Nokia NDCS16RM AirFrame Compute Nodes with 16 cores (dual-socket Intel Xeon CPU E5-2630 v3 @ 2.40GHz) and 64GB of 2133 MHz DDR4 memory. 82599ES 10-Gigabit Ethernet network interface card is installed to each server. Three different P4 targets,

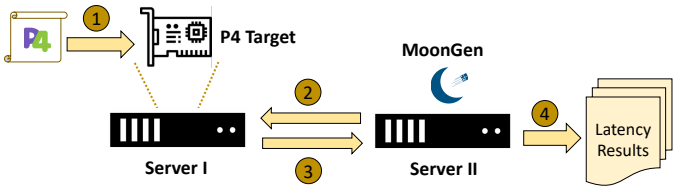


Fig. 3: Testbed setup consisting of two servers connected by two cables. The P4 device is installed in Server I, and the packet generator (MoonGen) runs in Server II. The measurement traffic is generated in Server II, sent to and processed in Server I with the respective P4 device, and sent back to Server II for latency analysis.

belonging to different processor families, are evaluated: *(i)* An Agilio CX 2x10GbE SmartNIC from Netronome [6], *(ii)* A NetFPGA-SUME board with Xilinx Virtex-7 XC7V690T FFG1761-3 FPGA [22], and *(iii)* An open-source T4P4S DPDK-based P4 software switch [20]. Hardware targets are attached to Server I's PCI bus, whereas the T4P4S switch runs as software in this server. The two physical ports of the P4 target under test are connected to two interfaces of Server II running MoonGen packet generator [23]. Each measurement case corresponds to its respective P4 program that is loaded into the P4 target. Packets are generated by MoonGen and sent over one link to the P4 targets at 10 Gbps rate, except for T4P4S software switch where the generated rate is set equal to the maximum rate where no packet drop was observed, which is equal to 9.7 Gbps. After being processed in the P4 target, packets are sent back to MoonGen for measuring and reporting the packet latency.

Three different packet sizes are considered, namely 256, 1000, and 1500 Bytes. For each measurement case, we collect more than 100,000 data points of latency values. MoonGen packet generator is a DPDK-based packet generator that generates 10 Gbps Ethernet traffic and beyond. It uses hardware timestamping capabilities of modern commodity NICs to deliver accurate and precise latency measurements with sub-microsecond precision [23].

Seven experiments are designed to study the effect of the basic elements (i.e., constructs) of a P4 pipeline on packet latency. The examined P4 constructs cover parsing headers, executing different header operations, and applying match-action tables. The P4 pipelines in these experiments are designed with different initial parsing cases to make sure that only one P4 construct is varied at a time, and accordingly, any measured variation in packet latency is due to this varying P4 construct. All the pipelines also includes a single table that forwards packets back to MoonGen by modifying the egress port while matching on the ingress port based on a fixed rule. Table I summarizes the different parameters and cases evaluated in our measurement campaign. The results of these experiments are used in Section V to derive a method for estimating the packet latency of an arbitrary network function, which is made up of the examined constructs, by analyzing its P4 program. In the following, we describe the objective of the performed experiments and the P4 pipelines used in every test case.

TABLE I: Cases and parameters considered while measuring the packet latency.

Variable	Value
P4 Targets	Netronome SmartNIC, NetFPGA-SUME, T4P4S DPDK-based Software Switch
P4 Constructs	Parsing Headers, Modifying Header Fields, Modifying Headers, Copying Headers, Removing Headers, Adding Headers, Adding Tables
Packet Sizes	256 Bytes, 1000 Bytes, 1500 Bytes
Rate	9 to 10 Gbps

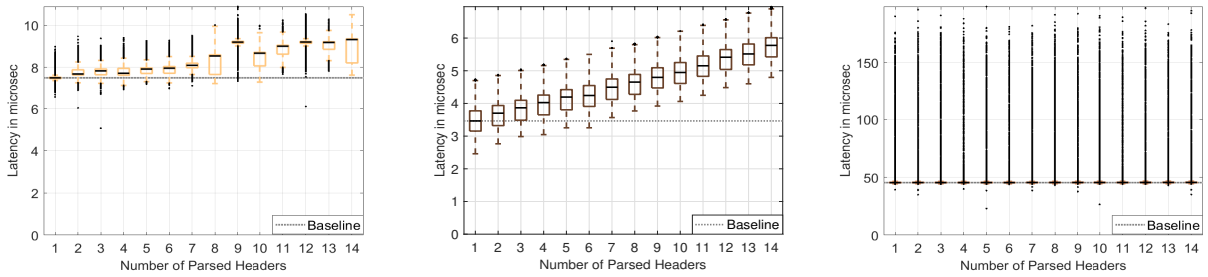
1) Parsing Headers: In this experiment, we quantify the latency cost of parsing headers in a P4 pipeline. For this purpose, we start with a baseline P4 pipeline, denoted by **Base_1**, which parses Ethernet header. Then, we increase the number of parsed headers in the parser stage until 14 and measure the packet latency of each P4 pipeline. The parsed-headers-stack is made up of Ethernet, IPv4, UDP, PTP, and 10 dummy headers each of size 16 Bytes. The first 4 headers should be maintained to ensure that the used packet generator, MoonGen, measures latency properly [23].

2) Modifying Fields within a Header: The objective of this experiment is to find the relation between the number of modified fields within a header and packet latency. This answers the question: *What is the difference in packet processing latency of a P4 pipeline when a single Ethernet header field, such as source MAC address, is modified versus the case when multiple Ethernet header fields, such as source and destination MAC addresses, are modified?* To investigate this question, we prepare and measure two sets of P4 programs that differ in the following: In the first set, a single field of three different headers: Ethernet, IPv4, and UDP is modified, while in the second set, multiple fields of these headers are modified.

3) Modifying Headers: In this experiment, we target quantifying the latency cost of modifying a different number of headers, defined in the header stack, of a P4 pipeline. For this purpose, we start with a P4 pipeline, denoted by **Base_14**, that parses 14 headers. Then, we modify the 14 parsed headers incrementally by changing one field of every modified header. The selection of **Base_14** to be a common pipeline in all the cases is done to avoid varying the number of parsed and modified headers at the same time, as headers cannot be modified without being parsed. The selection of the baseline pipeline in the remaining experiments is also done to fulfill the same single variable requirement.

4) Copying Headers: The objective of this experiment is to quantify the latency cost of applying copy-header operations, where one parsed header is copied into another parsed header in a P4 pipeline. In this experiment, we start with the **Base_14** P4 pipeline, and then we copy one of the parsed headers to another header. The latency results of copying 1 to 10 headers were recorded.

5) Removing Headers: The latency cost of removing headers from a received packet was measured in this experiment. The **Base_14** P4 pipeline was again used in this experiment as a baseline pipeline. Afterwards, we measure the latency of 10 P4 pipelines that only vary in the number of removed headers.



(a) Netronome SmartNIC. (b) NetFPGA-SUME. (c) T4P4S Switch.

Fig. 4: Forwarding latency given number of parsed headers for 1500 Bytes packets.

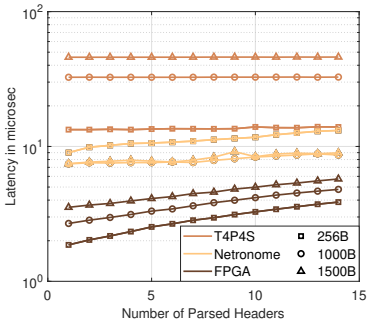


Fig. 5: Average forwarding latency given number of parsed headers for different targets and packet sizes.

6) Adding Headers: In this experiment, we evaluate the latency cost of adding headers into a packet in a P4 pipeline. We start with a baseline pipeline, denoted by **Base_4**, which parses Ethernet, IPv4, UDP, PTP headers. Then, we add 1 to 10 headers after the four parsed headers in the baseline pipeline, while measuring the latency of each add-header pipeline. In this experiment, the maximum packet size is set to 1300 Bytes instead of 1500 Bytes to guarantee that the received packets, with the added headers, do not exceed the MTU size. Additionally, we decrease the rate at which the packets are transmitted from 10G as more headers are added. This is important to make sure that the rate of the packets, with the added headers, sent back from the P4 target does not exceed the line rate.

7) Adding Tables: The objective of this experiment is to quantify the latency cost of adding tables into a P4 pipeline. We start with the **Base_4** pipeline, which already includes a single table matching on the ingress port and writing the egress port based on a fixed rule. Then, we add to the ingress stage of the P4 pipeline 1 to 14 additional tables, while measuring the packet latency. All added tables performs exact matching on a single field and executes the same forwarding action.

IV. Measurement Results

In this section, the results of the experiments described in Section III are presented.

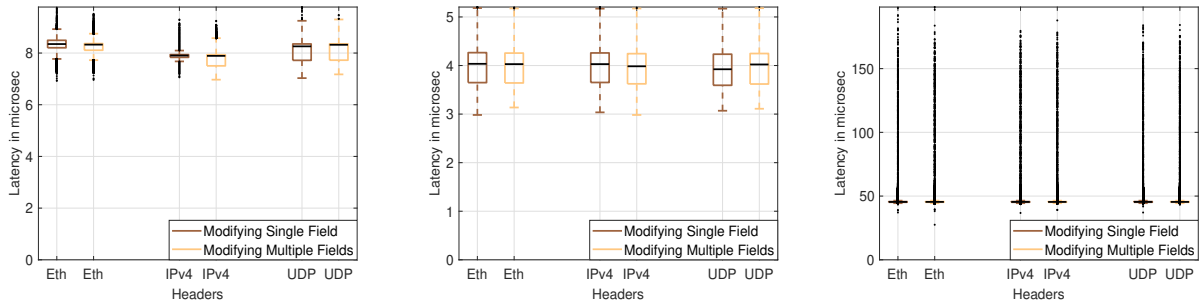
A. Parsing

Fig. 4 shows the box plots, which include the minimum, first quartile, median, third quartile, and the maximum of the measured packet latency in μs, as a function of the number of parsed headers for Maximum Transfer Unit (MTU) sized packets. The measurement results corresponding to the Netronome SmartNIC, presented in Fig. 4a, show that the median of the forwarding latency of the **Base_1** pipeline where only one header is parsed is around 7.5 μs. Then, the latency increases slightly as the number of parsed headers increases.

Similarly, the results corresponding to the NetFPGA-SUME card, illustrated in Fig. 4b, also show that parsing more headers leads to a linear increase in latency, as the measured latency increased by 2.3 μs when the number of parsed headers increases from 1 to 14. Parser states are translated into a Finite State Machine on the FPGA board which appears to have a high impact on processing latency [24]. We can observe a stable performance of NetFPGA-SUME as the distribution of measured packet latencies is very similar for the different parsing cases.

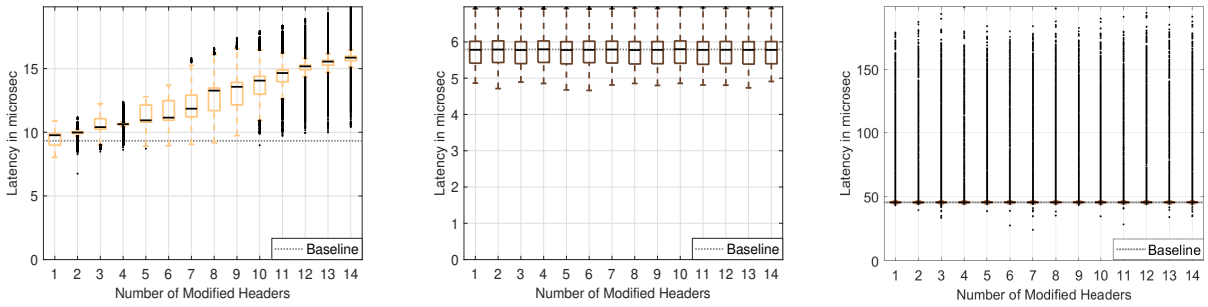
The measurement results corresponding to the T4P4S DPDK-based software switch, displayed in Fig. 4c, show that the median of the measured latency is constant at around 45 μs, as the number of parsed headers increases. This software switch implements header-parsing by storing meta-information related to the parsed header's first bit and size. Then, all parsed headers are emitted back through one memory copy operation, where the main latency cost is in the memory access, while the size of the copied headers has a negligible effect [7], [20]. We can observe that most measured packet latencies are centered around the median except for some outliers that could reach 200 μs. In our measurements, we observe that the packets with high latency values appear periodically, and this may be due to batch processing in DPDK.

Fig. 5 shows the average measured latency, in μs and in logarithmic scale, for the three investigated targets and packet sizes equal to 256 Bytes, 1000 Bytes, and 1500 Bytes as a function of the number of parsed headers. We can observe that for all targets, the packet latency variation is equally small as in Fig. 4. For every target, the packet size has an effect of shifting the curves by almost a constant latency factor, while the slopes are almost the same. While smaller packets are forwarded faster than larger packets on NetFPGA-SUME card and T4P4S, it is not the case on the Netronome SmartNIC,



(a) Netronome SmartNIC. (b) NetFPGA-SUME. (c) T4P4S Software Switch.

Fig. 6: Forwarding latency given number of modified fields for 1500 Bytes packets.



(a) Netronome SmartNIC. (b) NetFPGA-SUME. (c) T4P4S Switch.

Fig. 7: Forwarding latency given number of modified headers for 1500 Bytes packets.

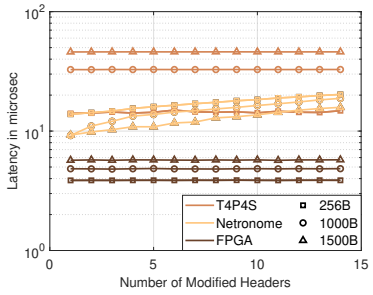


Fig. 8: Average forwarding latency given number of modified headers for different targets and packet sizes.

where we observe that the average forwarding latency of 256 Byte packets is larger than that for 1000 Byte and 1500 Byte packets. This is due to the frequent memory access, especially at small packet sizes, i.e., higher packet rates [25]. On average, the packet forwarding is performed on the NetFPGA-SUME card faster than that on the Netronome SmartNIC, which is faster than that on the T4P4S DPDK-based switch.

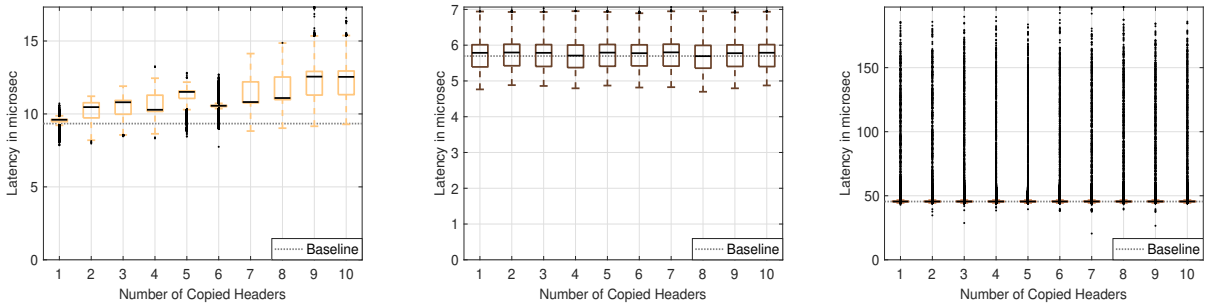
B. Modifying Fields within a Header

Fig. 6 shows the box plots of the measured packet latency, in μs, when a single field of Ethernet, IPv4, and UDP headers is modified versus the case when multiple fields of these headers are modified. The figure shows the results collected from the three investigated targets when the packet size is equal to 1500 Bytes. For all considered headers, and for all targets, we can observe that the latency results when modifying a single header field are very similar to the results when multiple header fields are modified. This is explained by the fact that all considered targets write the complete new header when a single field of the header is modified. This is also consistent with the P4 language deparsing syntax, where complete headers are *emitted* to reconstruct the packet. As a result, we infer that the packet latency of a P4 pipeline is independent of the number of fields modified within one header.

C. Modifying Headers

The results of modifying a different number of headers within a P4 pipeline is analyzed in this subsection. Note that the number of fields modified within each header is irrelevant as revealed in the previous experiment. The box plots of the measured packet latency, in μs, as a function of the number of modified headers for MTU sized packets is shown in Fig. 7. We recall that the pipelines of this experiment are built on top of the **Base_14** pipeline where 14 headers are parsed and whose median latency, for every target, is also plotted in this figure as baselines.

Fig. 7a shows that the median of the measured latency on the Netronome SmartNIC increases almost linearly with the number of modified headers from 9.8 μs to 15.9 μs. On the contrary, Fig. 7b and Fig. 7c show that there is no extra latency cost when headers are modified in the case of NetFPGA-SUME and T4P4S where the median of the measured latency is equal to 5.8 μs and 45 μs respectively. The reason for



(a) Netronome SmartNIC. (b) NetFPGA-SUME. (c) T4P4S Switch.

Fig. 9: Forwarding latency given number of copied headers for 1500 Bytes packets.

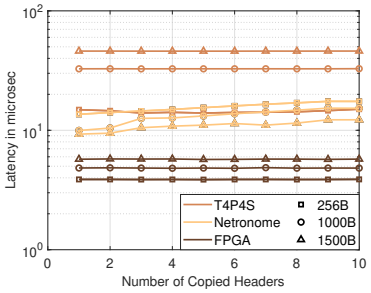


Fig. 10: Average forwarding latency given number of copied headers for different targets and packet sizes.

this variation is related to the implementation of every P4 target. While NetFPGA-SUME and T4P4S always write back all parsed headers [26], the Netronome SmartNIC writes back only the modified, *dirty*, headers. The FPGA writes all parsed headers back without differentiation because the additional logic to keep a record of the modified headers might become a critical component to satisfy the timing requirement, and its overhead will surpass its gains [27]. The distribution of the packet latency measured on different targets is similar to what was described in Subsection IV-A and it is consistent as the number of modified headers increases.

The average measured latency, in μs and in logarithmic scale, which corresponds to the three investigated targets for different packet sizes as a function of the number of modified headers is shown in Fig. 8. For all targets, we can still observe the same relation between the size of the packet and the average measured packet latency as the one analyzed based on Fig. 5. The figure also shows that the average packet forwarding latency of the NetFPGA-SUME card is minimal compared to the other two targets for all packet sizes. Additionally, we can observe that the Netronome SmartNIC outperforms T4P4S except for small packet sizes.

Note that the impact of calculating IP checksum in the P4 program after modifying an IP header field was also tested, and we do not observe any increase in measured latency in the cases of Netronome SmartNIC and T4P4S, while we observe a negligible increase, i.e. less than 0.15 μs, in the case of FPGA.

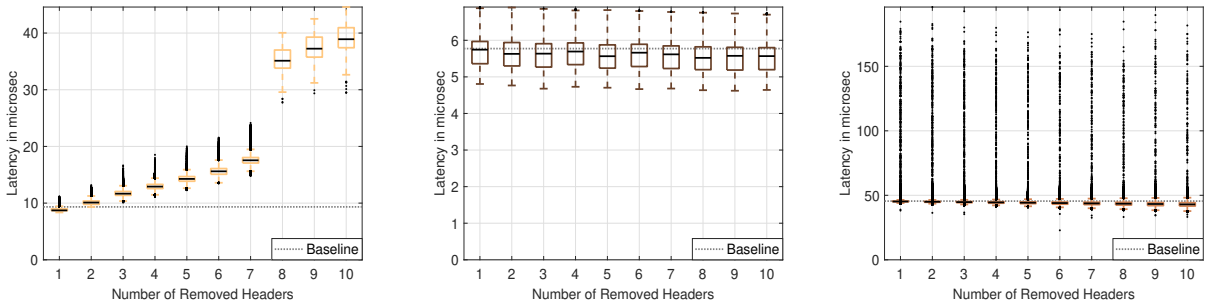
D. Copying Headers

The results of copying a different number of headers within a P4 pipeline is analyzed in this subsection. The pipelines of this experiment are built on top of **Base_14** pipeline where 14 headers are parsed and whose median latency results are presented in Fig. 9 as baselines. This Figure also shows the box plots of the packet latency measured on the three investigated targets, in μs, as the number of copied headers increases from 1 to 10 for MTU sized packets. We can generally observe that the results of this experiment are very similar to those obtained when we varied the number of modified headers from 1 to 10 in the previous experiment. This is also observed when looking at the results of Fig. 10, which shows the average measured latency, in μs and in logarithmic scale, as a function of the number of copied headers for different targets and packet sizes. The equivalence of these two experiments further verifies our previous observation in Experiment IV-B which says that modifying a single field of a header has the same impact on latency as modifying multiple fields, where copying a header to another header is an extreme case where we always modify all fields of the destination header. Note that the slight difference in the measured latency, up to 4 copied headers, in the results of these two experiments is because the headers being modified and copied in each experiment is different, as described in III-3 and III-4.

E. Removing Headers

Fig. 11 shows the box plots of the measured packet latency, in μs, as a function of the number of removed headers for packets of size equals MTU. In addition, the figure shows the median latency of the **Base_14** pipeline which is used in this experiment as a baseline.

Fig. 11a, shows that the median of the forwarding latency, measured on the Netronome SmartNIC, increases linearly from 8.7 μs to 17.5 μs as the number of removed headers increases from 1 to 7. Then, we observe a sharp increase in the measured latency which reaches 35.1 μs at 8 headers followed by a linear increase to reach 39 μs at 10 headers. The sharp increase in latency observed after 7 headers is related to Netronome SmartNIC's implementation specifics. When 7 headers need to be removed, an infrastructural process in the SmartNIC at the deparsing stage is involved. This process depends on



(a) Netronome SmartNIC. (b) NetFPGA-SUME. (c) T4P4S Switch.

Fig. 11: Forwarding latency given number of removed headers for 1500 Bytes packets.

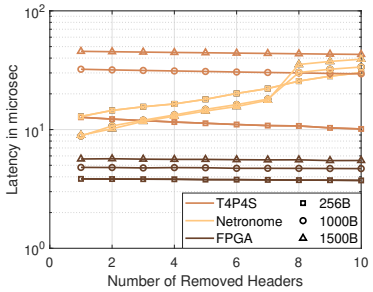


Fig. 12: Average forwarding latency given number of removed headers for different targets and packet sizes.

the space between the headers to be removed and the end of the packet. If the removed headers exceed a specific size, this process becomes more expensive and involves moving the whole payload which makes it also dependent on the size of the packet. Our measurements reveal that this behavior takes place when more than $7 \cdot 16B = 112B$ are removed from the header stack of a packet.

Figures 11b and 11c show that the median of the measured latency on the NetFPGA-SUME card and T4P4S slightly decreases from 5.7 μs to 5.6 μs and from 45 μs to 43 μs respectively when the number of removed headers increases from 1 to 10. This slight decrease in latency is related to the fact that the NetFPGA-SUME card and T4P4S write the whole parsed stack back in the deparsing stage, as discussed in Subsection IV-C, and when headers are removed from this stack, the cost of this operation will decrease. The distribution of the packet latency measured on different targets is similar to what was described in Subsection IV-A and it is consistent as the number of removed headers increases.

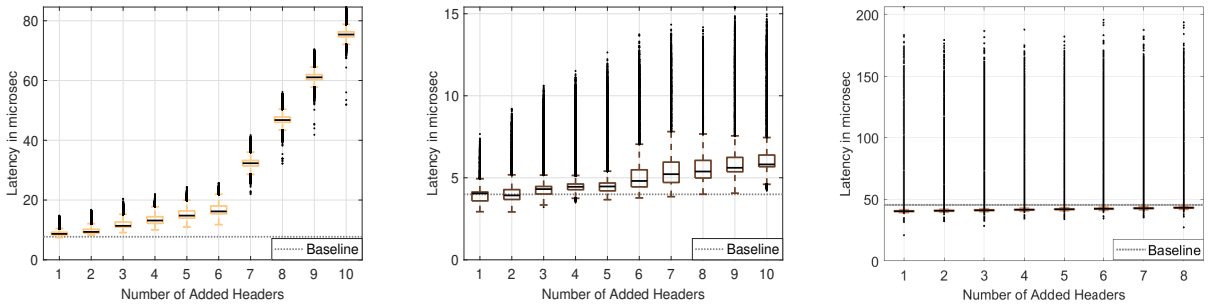
The average measured latency, in μs and in logarithmic scale, which corresponds to the three investigated targets for different packet sizes as a function of the number of removed headers, is shown in Fig. 12. The same relation as before between the size of the packet and the average measured packet latency still holds for all the targets except for the Netronome SmartNIC after 7 headers where removing headers operation becomes more expensive for larger packets due to the reason discussed before. The figure also shows that the average packet forwarding latency of the NetFPGA-SUME card is the smallest compared to the other two targets for all packet sizes. We can also observe that the Netronome SmartNIC outperforms T4P4S except when the packet size is equal to 256 Bytes, and the case when the number of removed headers exceeds 7 headers at packet size equals 1000 Bytes.

F. Adding Headers

The box plots of the measured packet latency, in μs, as a function of the number of added headers for 1300 Bytes packets, almost MTU, is shown in Fig. 13. The pipelines of this experiment are built on top of the **Base_4** pipeline where 4 headers are parsed. The result of the baseline pipeline, although corresponds to 1500 Bytes packets, is presented in Fig. 13 for different targets.

Fig. 13a shows that the median of the forwarding latency, measured on the Netronome SmartNIC, increases linearly from 8.7 μs to 16.2 μs as the number of added headers increases from 1 to 6. Then, the latency increases with a steeper slope to reach 75.4 μs when the number of added headers increases to 10 headers. The reason for the sharp increase in latency after 6 headers is again related to the involvement of the infrastructural process in the Netronome SmartNIC, which depends on the space between the headers to be added and the beginning of the packet. Note that if the added headers exceed a specific size, this process becomes more expensive and involves moving the whole payload which makes it also dependent on the size of the packet. From Fig. 13a, we can observe that this behavior starts when more than $6 \cdot 16B = 96B$ are added to the header stack of a packet.

From Fig. 13b and Fig. 13c, we can observe that the median of the measured latency on the NetFPGA-SUME card increases from 4 μs to 5.8 μs as the number of added headers increases from 1 to 10, while the median of the measured latency on T4P4S increases from 40.3 μs to 43.1 μs when the number of added headers increases from 1 to 8. This increase is again related to the fact that these two targets will emit a larger header stack as more headers are added. Note that the maximum number of headers that can be added on T4P4S is 8 headers due to the DPDK implementation. In front of the packet data, there is an extra space for adding headers which is defined as the constant *RTE_PKTMBUF_HEADROOM* and set equal to 128 Bytes [20]. In our case, each added header has the size of 16 Bytes, which makes the maximum number



(a) Netronome SmartNIC. (b) NetFPGA-SUME. (c) T4P4S Switch.

Fig. 13: Forwarding latency given number of added headers for 1300 Bytes packets.

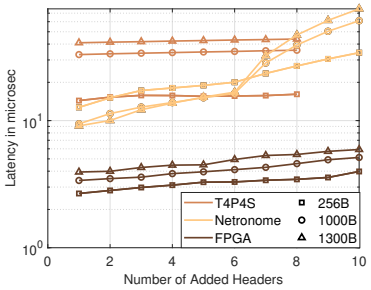


Fig. 14: Average forwarding latency given number of added headers for different targets and packet sizes.

of headers that can be added into a packet before filling its *HEADROOM* is 8 headers, $128B/16B = 8$. The distribution of the packet latency measured on different targets is similar to what was described in Subsection IV-A and it is consistent as the number of added headers increases except for the NetFPGA-SUME card, where we observe more outliers with high latency values.

Fig. 14 shows the average measured latency, in μs and in logarithmic scale that corresponds to the different investigated targets for different packet sizes as a function of the number of added headers. The previously analyzed relation between the size of the packet and the average measured packet latency still holds for all the targets except for the Netronome SmartNIC after 6 headers where adding headers operation becomes more expensive for larger packets due to the reason discussed before. The NetFPGA-SUME card always has the lowest average packet forwarding latency compared to the other two targets for all packet sizes. The Netronome SmartNIC outperforms T4P4S except for small packet size after adding one header and the case when the number of added headers is equal to 8 with packet sizes equal 1000 and 1500 Bytes.

G. Adding Tables

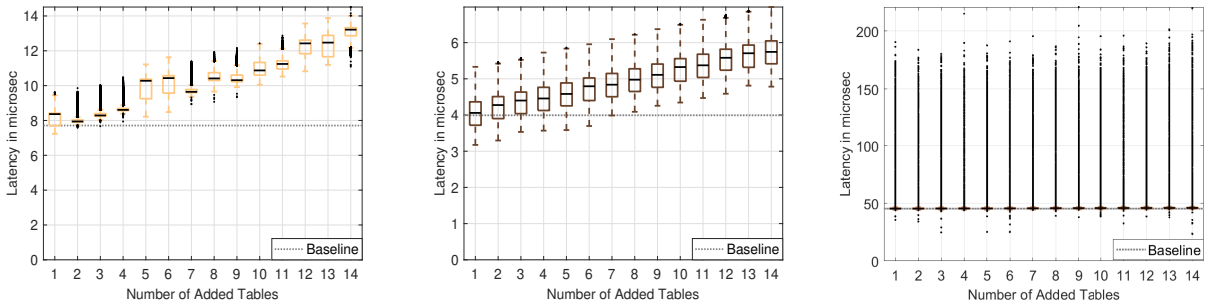
The box plots of the measured packet latency, in μs, as a function of the number of added tables for packets of size equals 1500 Bytes is shown in Fig. 15 along with the median latency of the **Base_4** pipeline which serves as a baseline.

As the number of added tables increases from 1 to 14, we can observe from Figures 15a, 15b, and 15c that the median of the measured latency increases from 8.4 μs to 13.2 μs in the case of the Netronome SmartNIC, from 4 μs to 5.7 μs in the case of the NetFPGA-SUME card, while it slightly increases from 45.5 μs to 46.1 μs in the case of T4P4S. This increase is expected for all targets as more matching and lookup operations are applied.

Fig. 16 shows the average measured latency, in μs and in logarithmic scale, that corresponds to the three investigated targets for different packet sizes as a function of the number of added tables. For all packet sizes, we can observe that the average forwarding latency of the NetFPGA-SUME card is less than that of the Netronome SmartNIC which is less than that of the T4P4S. The relation between packet sizes and measured latency described before still holds for all the targets except for T4P4S when the packet size is equal to 256 Bytes and the number of added tables exceeds 3 tables. In this case, the latency increases significantly and we observe packet drop. In T4P4S, tables for exact lookup are implemented with DPDK hash tables [7], [20]. For small packet size case, i.e., higher packet rate, as the number of tables increase, more lookup operations per second are executed. This increasing number of required lookup operations hit a performance limit related to the number of table lookups and actions that can be fitted on the same CPU core due to hardware constraints such as available CPU cycles, cache memory size, cache transfer Bandwidth, memory transfer Bandwidth, etc [20]. For 256 Bytes packets, the maximum rate that can be processed in T4P4S with 14 matching tables without packet drop was examined and it is equal to 4.5 Gbps. Finally, it is worth noting that the impact of increasing the number of rules installed in a single table from 1 to 20 rules was examined and we did not observe a notable change in the measured latency on all three targets. Moreover, the type of matching, i.e, exact versus wildcard, was also investigated where we did not observe noticeable difference in the measured packet latency when matching type changed.

V. Estimation Method

In this section, with the assumption of deterministic processing, we build on top of the previously derived results and observations to propose a method for estimating the average latency of any network function (NF) written as a P4 program.



(a) Netronome SmartNIC. (b) NetFPGA-SUME. (c) T4P4S Switch.

Fig. 15: Forwarding latency given number of added tables for 1500 Bytes packets.

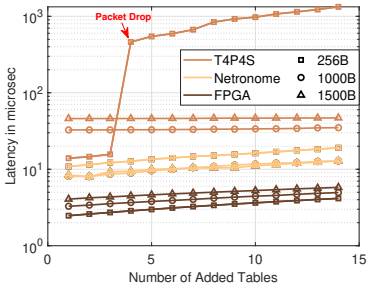


Fig. 16: Average forwarding latency given number of added tables for different targets and packet sizes.

Then, we validate the prediction accuracy of the proposed method.

The estimation method regards the surveyed P4 constructs as a basis and the P4 data paths as a span of this basis. Therefore, by quantifying the impact of the basic set of P4 constructs on packet latency, we can estimate the latency of an arbitrary P4 program as the linear combination of the analyzed constructs. First, Subsection V-A describes how the profiles of the three investigated P4 targets are derived using the results of Section IV related to the impact of different P4 constructs on packet latency. Second, Subsection V-B describes the relevant information that should be extracted from a P4 program to perform the estimation. Then, Subsection V-C explains how the estimated average latency, parameterized with the derived targets' profiles, can be calculated as a function of P4 programs' extracted information. Fig. 17 illustrates the workflow of the proposed estimation method. Finally, in Subsection V-D, we validate the proposed method by testing it on three different realistic network functions.

A. Profiling Targets

Using the results observed in Section IV, and for every target, we require the following two information:

(1.) The mapping between the number of parsed headers and the measured packet latency, denoted by P_{target}, which returns the measured average forwarding latency, in μs, as a function of the number of parsed headers for a given target. This mapping can be read using the results shown in Fig. 5.

(2.) Using linear interpolation, we extract the slopes of the curves capturing the increase in packet latency as a function of the number of modified headers, number of copied headers, number of removed headers, number of added headers, and number of added tables from Figures 8, 10, 12, 14, and 16 respectively. The slopes of the fitted curves indicate the packet latency cost, in μs, per added P4 construct. For example, if the interpolated slope of adding headers curve is equal to 1, then the latency cost of every added header on top of the baseline pipeline is equal to 1 μs. Note that piece-wise linear interpolation is performed for the cases of adding and removing headers on the Netronome SmartNIC to capture the changing behavior of the card when big number of headers are added or removed. The root mean square error when performing the linear interpolation for the latency plots of all the targets and for all P4 constructs is always less than 0.4 μs.

In the following, we refer to the results with MTU sized packets. The method can be easily extended to other packet sizes by applying necessary substitutions, especially that the packet size has the effect of only shifting the parallel curves corresponding to a specific target, without major changes on curves' slopes as analyzed in Section IV.

$$a^{T}_{target} = \begin{bmatrix} \Delta^{target}_{ModifyHeader} \\ \Delta^{target}_{CopyHeader} \\ \Delta^{target}_{RemoveHeader} \\ \Delta^{target}_{AddHeader} \\ \Delta^{target}_{AddTable} \end{bmatrix} \quad (1)$$

We define the **target-profile-vector**, a_{target}, to be a row vector containing the interpolated slopes corresponding to different P4 constructs for a specific target, denoted by $\Delta^{target}_{P4Construct}$. The order of the slopes of P4 constructs in a_{target} is shown in Eq. 1.

Accordingly, the targets-profile-vectors corresponding to T4P4S, NetFPGA-SUME, and Netronome SmartNIC can be derived as shown in Equations 2, 3, and 4 respectively.

$$a_{T4P4S} = \begin{bmatrix} 0 & 0 & -0.29 & 0.4 & 0.08 \end{bmatrix} \quad (2)$$

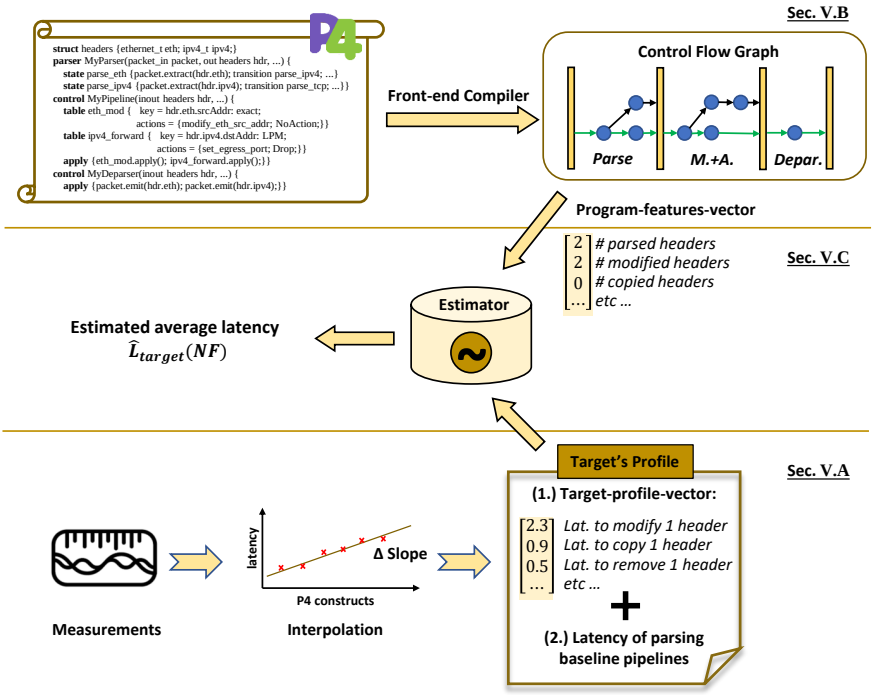


Fig. 17: Overview of the workflow to estimate the average latency. Latency estimation depends on target-related information as described in Subsection V-A and program-related information as described in Subsection V-B.

$$a_{FPGA} = \begin{bmatrix} 0 & 0 & -0.02 & 0.23 & 0.13 \end{bmatrix} \quad (3)$$

$$a_{Netronome} = \begin{bmatrix} 0.5 & 0.28 & 1.43 & 1.59 & 0.37 \end{bmatrix} \quad (4)$$

For accurate profiling of Netronome SmartNIC's latency variation, the following two measures should be taken. In case the number of added headers is greater than 6, the interpolated slope corresponding to adding headers in Eq. 4 should be substituted with the value 14.4 instead of 1.59 to capture the steeper slope after adding 6 headers, as described in Subsection IV-F. Additionally, a correction factor of 17.68 μs should be added to the estimated average latency when the number of removed headers is more than 7 as observed in Subsection IV-E.

B. Analyzing P4 Programs

Given a P4 program, we first compile it with the standard front-end compiler and get the IR of the program's logic which can be drawn as a Control Flow Graph (CFG). The CFG is a directed acyclic graph that starts when a packet is taken from an ingress queue and terminates when that packet is pushed to an egress queue or dropped. The arrows indicate the relative order. Note that because of the generality of a P4 program, different packets can traverse different paths in the CFG. Therefore, the vector should reveal the actual path of a packet (e.g., the green lower path with two parsed headers and two packet header operations shown in Fig. 17). In the parse phase, each node represents a state where a header is parsed (extracted), and the next header to be parsed is identified based on a certain field of the extracted header, e.g., Ethernet Type or IP Protocol fields. In the match-action phase, each node represents a matching table and all associated actions, e.g., a table matching on IP destination address and two actions to modify the address or mark to drop the packet.

Afterwards, we extract the **program-features-vector**, denoted by $x(NF)$, from the CFG of the analyzed P4 program. The vector $x(NF)$ contains the number of occurrence of the basic P4 constructs in the given P4 program, written for network function NF, with the following order:
$x(NF)$ = [#parsed headers, #modified headers, #copied headers, #removed headers, #added headers, #tables-1].

Note that the last element in this vector counts the number of tables added to the baseline pipeline used in the experiment described in III-7. In this experiment, the used baseline pipeline, **Base_4**, already contains a table for forwarding the packets, so the number of added tables on top of this pipeline will be equal to the total number of tables counted in the P4 program minus one.

C. Average Latency Calculation:

We recall that all designed experiments in Section III, corresponding to all analyzed P4 constructs, build on top of baseline pipelines that parse some number of headers, and modify the egress port according to the ingress port in a single table to forward the packet. Additionally, we made sure that the analyzed P4 construct is the only parameter varying in every experiment. Therefore, in estimating the average latency of an NF, we add the following two terms: *(1.)* The measured

packet latency as a function of the number of parsed headers captured by P_{target}, which includes: tx delay, propagation delay, packet processing delays before and after entering the P4 pipeline, and the processing delay of parsing operation; *(2.)* The sum of the extra latency cost of applying a P4 construct on top of the baseline parsing pipelines, captured by a_{target}, weighted with the number of occurrences of each P4 construct after parsing extracted for a given P4 program, $x(NF)[2 : end]$. Eq. 5 shows the formula for evaluating the estimated average packet latency in μs, $\hat{L}_{target}(NF)$, of running network function, NF, on any target.

$$\hat{L}_{target}(NF) = P_{target}(x[1]) + a_{target} \cdot x^T[2 : end] \quad (5)$$

D. Validation

For validation and illustration purposes, we apply our proposed estimation method on three network functions, with an increasing complexity, written in P4. Then, we compare the estimated latency to the latency measured when the tested network functions are loaded to each of the investigated P4 targets. The examined network functions are the following:

L3_Forwarding: The P4 pipeline of this network function includes parsing Ethernet and IPv4 headers, matching on IPv4 destination address in a single table, and modifying the source and destination MAC addresses of Ethernet header and the time to live (TTL) field of IPv4 header upon matching. The program-features-vector corresponding to this network function, $x(L3\_Fwd)$, is equal to $\begin{bmatrix}2 & 2 & 0 & 0 & 0 & 0\end{bmatrix}$.

L3_Forwarding + UDP-based Firewall: The P4 pipeline corresponding to this network function is similar to the one described in L3_Forwarding but it also involves parsing UDP header. Additionally, the data path of this network function includes one more table matching on UDP destination port to drop undesired packets. The program-features-vector corresponding to this network function, $x(L3\_Fwd + Firewall)$, is equal to $\begin{bmatrix}3 & 2 & 0 & 0 & 0 & 1\end{bmatrix}$.

VxLAN Decapsulation: In this network function, VxLAN decapsulation is performed. Note that we perform this functionality while maintaining the header stack up to PTP header to guarantee that the packet generator, MoonGen, can still perform latency measurements. The evaluated P4 pipeline parses the following header stack: Eth, IPv4, UDP, PTP, ETH, IPv4, UDP, VxLAN, Eth, IPv4, UDP. Besides the table that forwards packets based on ingress port matching as in the baseline pipeline, we have another table that matches on VXLAN Network Identifier (VNI) and performs the VxLAN Decapsulation action. The VxLAN Decapsulation action includes copying the inner Ethernet, inner IPv4, and inner UDP headers into the outer headers, then removing the VxLAN, inner Ethernet, inner IPv4, and inner UDP headers. The program-features-vector, $x(VxLAN\_Decap)$, corresponding to this network function is equal to $\begin{bmatrix}11 & 0 & 3 & 4 & 0 & 1\end{bmatrix}$.

Then, the estimated average latency of each network function can be calculated using Eq. 5, for a given target where its profile information should be substituted. As an example, the average latency of L3_Forwarding network function running on Netronome SmartNIC is estimated as follows $P_{Netronome}(2)$+[0.5, 0.28, 1.43, 1.59, 0.37] $\cdot$ [2, 0, 0, 0, 0]^T which is equal to $7.73 + 2 \cdot 0.5 = 8.73$ μs.

The three tested network functions are loaded into the three investigated P4 targets. The average latency for each network function was measured and compared to the estimated latency derived based on the method described in Subsections V-A, V-B, and V-C. Table II shows the measured and estimated average latency of the three tested network functions on the three investigated P4 targets. We can always observe that the difference between the estimated latency and the measured latency is less than one microsecond.

Moreover, the accuracy of the method in estimating the average packet forwarding latency is always greater than 94%. This recorded high accuracy reflects the validity of the proposed estimation method. By looking into different P4 targets, we can observe that predicting the latency of the NetFPGA-SUME card has the highest accuracy, followed by that of T4P4S, followed by that of the Netronome SmartNIC. This is due to the high dependence of the Netronome SmartNIC on the loaded P4 pipeline compared to the other targets, as can be inferred based on the target-profile-vectors derived in Subsection V-A. On the other hand, through looking into different network functions, we can observe that as the complexity of the network function increases, the accuracy of the estimation method slightly decreases, as we can see in the case of the VxLAN Decapsulation network function.

The proposed methodology can be generalized to other P4 targets by profiling these targets and extracting the **target-profile-vector** as described in Subsection V-A. The profiling step needs to be performed once per target, where the relevant performance information, summarized in the **target-profile-vector**, can be published by researchers or third part institutions such as *OPNFV*. The compiler can easily analyze/ decompose the provided P4 program to extract the **programs-feature-vector**, and make use of the publicly available **target-profile-vector** to calculate the estimated average packet latency for a given P4 program on a specific target, as described in Subsections V-B and V-C.

TABLE II: Validation results of the proposed method for different network functions and P4 targets.

NF	P4 target	Meas. Avg. L	Est. Avg. L
L3_Fwd	T4P4S	45.9 μs	45.9 μs
	Netro. SmartNIC	8.2 μs	8.7 μs
	NetFPGA-SUME	3.7 μs	3.7 μs
L3_Fwd + Firewall	T4P4S	45.9 μs	45.9 μs
	Netro. SmartNIC	8.9 μs	9.1 μs
	NetFPGA-SUME	4.0 μs	4.0 μs
VxLAN_ Decap	T4P4S	45.9 μs	45μs
	Netro. SmartNIC	15.2 μs	15.7μs
	NetFPGA-SUME	5.2 μs	5.2μs

VI. Related Work

The most closely related work is Whippersnapper [28], which describes a P4 performance benchmarking tool. We proceed a step further by evaluating the impact of a com-

plete set of P4 constructs on packet latency for state-of-the-art software and hardware P4 targets with comprehensive measurements. Moreover, we propose a method to estimate the average latency given a P4 pipeline, which does not exist in the literature. Below, we explain in detail the related work on benchmarking of networking devices, general latency modeling and analysis, and performance evaluation of P4.

Benchmarking. Various tools have been proposed for different kinds of networking devices, e.g., legacy switches [29], [30], SDN switches [31], [32], SDN control plane [33], VNFs [34], [35], and P4 switches [13], [28], to evaluate various aspects such as throughput, latency, and reliability. Our work focuses on the latency aspect and covers various P4 architectures.

General latency modeling and analysis. Numerous studies have been performed to analyze the network latency from different aspects such as the underlying software architecture [36] and the impact of control plane [37]. From the modeling perspective, single devices, such as general software switches [38], software DPDK switch [39], OpenFlow switch [25], [40] and VNF [41], and networks ranging from switches [42], [43] and VNF chains [44] are covered in depth. The models applied include stochastic models based on queuing theory [25], [43], [45] and network calculus [46]. Our work, however, follows an experimental approach: we derive the latency of each P4 construct with comprehensive measurements.

Performance evaluation of P4. Static performance of software switches [28] and FPGA [27] running P4 programs, as well as performance variation during runtime reconfiguration [47], are explored in different works. On a smaller scale, researches have been performed on evaluating the performance of P4 targets executing specific functionalities, such as encryption [48] (on NetFPGA and SmartNIC), in-network event processor [49]–[51] (on SmartNIC and Tofino), and stateless load-balancing [52], [53] (on NetFPGA and Tofino). Instead of taking a network function written in P4 as a whole, we break it down into atomic constructs, which can be applied to all targets and assembled back to form different network functions.

VII. Conclusion

The introduction of data plane programmability into packet processors added the complexity of the configured data plane as one more variable that affects packet processing latency. The relation between the complexity of P4 data planes and packet forwarding latency was investigated in this paper. Towards this objective, we measured the impact of the basic P4 constructs on packet latency for three P4 targets: NPU-based Netronome SmartNIC, NetFPGA-SUME card, and DPDK-based T4P4S software switch. The measurement results reveal that different P4 targets had different P4 constructs as influential parameters that affect the target's forwarding latency. Additionally, we observed a linear variation in the forwarding latency, for all the targets, when an influential P4 construct varied. Based on these observations, we proposed a method for estimating the average packet forwarding latency of arbitrary P4 programs written using the surveyed P4 constructs. The proposed method was applied to three realistic network functions on the three investigated targets and recorded a promising sub-microsecond precision.

This work can be extended by applying the proposed methodology to other P4 targets. Although our preliminary evaluation revealed that the number of installed flow rules/entries has a negligible impact on measured packet latency, a thorough evaluation can be further performed on this topic. Moreover, extending the proposed approach for estimating the deadline latency and jitter of a device by considering the maximum and the variance of the measured packet latency can be further investigated. This work serves as the first building block towards deriving analytical performance models for P4 programmable devices. Based on the estimated processing latency analyzed in this work, which provides information related to the service process of P4 devices, we can consider different arrival processes where the impact of different input traffic patterns can be evaluated using queuing theory based models.

References

[1] M. He, A. M. Alba, A. Basta, A. Blenk, and W. Kellerer, "Flexibility in softwarized networks: Classifications and research challenges," *IEEE Communications Surveys & Tutorials*, 2019.

[2] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn," *Queue*, vol. 11, no. 12, pp. 20–40, 2013.

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[4] R. Bifulco and G. Rétvári, "A survey on the programmable data plane: Abstractions, architectures, and open problems," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2018, pp. 1–7.

[5] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "Netfpga sume: Toward 100 gbps as research commodity," *IEEE micro*, vol. 34, no. 5, pp. 32–41, 2014.

[6] Netronome. (2017) Netronome smartnic. https://www.netronome.com/products/smartnic/overview/. Accessed: 2019-07-04.

[7] P. Vörös, D. Horpácsi, R. Kitlei, D. Leskó, M. Tejfel, and S. Laki, "T4p4s: A target-independent compiler for protocol-independent packet processors," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2018, pp. 1–8.

[8] H. Song, "Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013, pp. 127–132.

[9] P. L. Consortium. (2016) P4 16 Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html. Accessed: 2020-01-25.

[10] A. Van Bemten, N. Deric, A. Varasteh, A. Blenk, S. Schmid, and W. Kellerer, "Empirical predictability study of sdn switches," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2019, pp. 1–13.

[11] B. Martini, F. Paganelli, P. Cappanera, S. Turchi, and P. Castoldi, "Latency-aware composition of virtual functions in 5g," in *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2015, pp. 1–6.

[12] L. Qu, C. Assi, and K. Shaban, "Delay-aware scheduling and resource optimization with network function virtualization," *IEEE Transactions on Communications*, vol. 64, no. 9, pp. 3746–3758, 2016.

[13] H. Harkous, M. Jarschel, M. He, R. Priest, and W. Kellerer, "Towards understanding the performance of p4 programmable hardware," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 2019, pp. 1–6.

[14] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu, "Dc.p4: Programming the forwarding plane of a data-center switch," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. ACM, 2015.

[15] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[16] P. L. Consortium. (2016) P4C. https://github.com/p4lang/p4c. Accessed: 2020-01-25.

[17] T. K. Dangeti, R. Upadrasta *et al.*, "P4llvm: An llvm based p4 compiler," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, 2018, pp. 424–429.

[18] P. L. Consortium. (2016) Behavioral Model (bmv2). https://github.com/p4lang/behavioral-model. Accessed: 2020-01-25.

[19] Netronome. (2017) Mapping P4 to SmartNICs. https://p4.org/assets/p4_d2_2017_nfp_architecture.pdf. Accessed: 2020-01-25.

[20] D. Project. (2010) DPDK (Data Plane Development Kit). https://www.dpdk.org/. Accessed: 2020-01-25.

[21] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The p4 $\rightarrow$ netfpga workflow for line-rate packet processing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 1–9.

[22] Xilinix. (2018) Virtex-7 FPGAs. https://www.xilinx.com/products/silicon-devices/fpga/virtex-7.html. Accessed: 2020-01-25.

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

[24] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

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

[26] P. Vörös, D. Horpácsi, R. Kitlei, D. Leskó, M. Tejfel, and S. Laki. (2017) t4p4s repository. https://github.com/P4ELTE/t4p4s. Accessed: 2020-01-25.

[27] P. Benáček, V. Puš, H. Kubátová, and T. Čejka, "P4-to-vhdl: Automatic generation of high-speed input and output network blocks," *Microprocessors and Microsystems*, vol. 56, pp. 22–33, 2018.

[28] H. T. Dang, H. Wang, T. Jepsen, G. Brebner, C. Kim, J. Rexford, R. Soulé, and H. Weatherspoon, "Whippersnapper: A p4 language benchmark suite," in *Proceedings of the Symposium on SDN Research*, 2017, pp. 95–101.

[29] S. Zhang, Y. Liu, W. Meng, Z. Luo, J. Bu, S. Yang, P. Liang, D. Pei, J. Xu, Y. Zhang *et al.*, "Prefix: Switch failure prediction in datacenter networks," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 1, pp. 1–29, 2018.

[30] M. Sharafeddin, H. Harkous, S. Mansour, M. A. R. Saghir, H. Akkary, H. Artail, and H. Hajj, "On the efficiency of automatically generated accelerators for reconfigurable active ssds," in *2014 26th International Conference on Microelectronics (ICM)*, 2014, pp. 124–127.

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

[32] A. Van Bemten, N. Deric, J. Zerwas, A. Blenk, S. Schmid, and W. Kellerer, "Loko: Predictable latency in small networks," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, 2019, pp. 355–369.

[33] M. Jarschel, F. Lehrieder, Z. Magyari, and R. Pries, "A flexible openflow-controller benchmark," in *2012 European Workshop on Software Defined Networking*. IEEE, 2012, pp. 48–53.

[34] D. Cotroneo, L. De Simone, and R. Natella, "Nfv-bench: A dependability benchmark for network function virtualization systems," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 934–948, 2017.

[35] T. Zhang, L. Linguaglossa, J. Roberts, L. Iannone, M. Gallo, and P. Giaccone, "A benchmarking methodology for evaluating software switch performance for nfv," in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019, pp. 251–253.

[36] D. Y. Huang, K. Yocum, and A. C. Snoeren, "High-fidelity switch models for software-defined network emulation," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013, pp. 43–48.

[37] K. He, J. Khalid, S. Das, A. Gember-Jacobson, C. Prakash, A. Akella, L. E. Li, and M. Thottan, "Latency in software defined networks: Measurements and mitigation techniques," in *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2015, pp. 435–436.

[38] K. Suksomboon, N. Matsumoto, S. Okamoto, M. Hayashi, and Y. Ji, "Configuring a software router by the erlang-k-based packet latency prediction," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, pp. 422–437, 2018.

[39] T. Begin, B. Baynat, G. A. Gallardo, and V. Jardin, "An accurate and efficient modeling framework for the performance evaluation of dpdk-based virtual switches," *IEEE Transactions on Network and Service Management*, vol. 15, no. 4, pp. 1407–1421, 2018.

[40] Y. Goto, B. Ng, W. K. Seah, and Y. Takahashi, "Queueing analysis of software defined network with realistic openflow–based switch model," *Computer Networks*, vol. 164, p. 106892, 2019.

[41] J. Prados-Garzon, P. Ameigeiras, J. J. Ramos-Munoz, P. Andres-Maldonado, and J. M. Lopez-Soler, "Analytical modeling for virtualized network functions," in *2017 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 2017, pp. 979–985.

[42] K. Mahmood, A. Chilwan, O. Østerbø, and M. Jarschel, "Modelling of openflow-based software-defined networks: the multiple node case," *IET Networks*, vol. 4, no. 5, pp. 278–284, 2015.

[43] B. Xiong, K. Yang, J. Zhao, W. Li, and K. Li, "Performance evaluation of openflow-based software-defined networks based on queueing model," *Computer Networks*, vol. 102, pp. 172–185, 2016.

[44] Q. Ye, W. Zhuang, X. Li, and J. Rao, "End-to-end delay modeling for embedded vnf chains in 5g core networks," *IEEE Internet of Things Journal*, vol. 6, no. 1, pp. 692–704, 2018.

[45] G. Shen, Q. Li, S. Ai, Y. Jiang, M. Xu, and X. Jia, "How powerful switches should be deployed: A precise estimation based on queuing theory," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 811–819.

[46] A. K. Koohanestani, A. G. Osgouei, H. Saidi, and A. Fanian, "An analytical model for delay bound of openflow based sdn using network calculus," *Journal of Network and Computer Applications*, vol. 96, pp. 31–38, 2017.

[47] M. He, A. Basta, A. Blenk, N. Deric, and W. Kellerer, "P4NFV: An NFV architecture with flexible data plane Reconfiguration," in *Proceedings of the International Conference on Network and Service Management (CNSM)*. IEEE, 2018, pp. 90–98.

[48] D. Scholz, A. Oeldemann, F. Geyer, S. Gallenmüller, H. Stubbe, T. Wild, A. Herkersdorf, and G. Carle, "Cryptographic hashing in p4 data planes," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 2019, pp. 1–6.

[49] T. Kohler, R. Mayer, F. Dürr, M. Maaß, S. Bhowmik, and K. Rothermel, "P4cep: Towards in-network complex event processing," in *Proceedings of the 2018 Morning Workshop on In-Network Computing*, 2018, pp. 33–38.

[50] P. Bressana, N. Zilberman, D. Vucinic, and R. Soule, "Trading latency for compute in the network," in *ACM SIGCOMM 2020 Workshop on Network Application Integration/CoDesign (NAI)*. ACM, 2020, pp. 1–6.

[51] Y. Qiao, X. Kong, M. Zhang, Y. Zhou, M. Xu, and J. Bi, "Towards in-network acceleration of erasure coding," in *Proceedings of the Symposium on SDN Research (SOSR)*, 2020, pp. 41–47.

[52] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with beamer," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 125–139.

[53] G. Grigoryan, Y. Liu, and M. Kwon, "iload: In-network load balancing with programmable data plane," in *Proceedings of the 15th International Conference on emerging Networking EXperiments and Technologies*, 2019, pp. 17–19.