On-the-fly Table Insertions on Programmable Software Data Planes

Manuel Simon, Sebastian Gallenmüller, and Georg Carle
Chair of Network Architectures and Services, Technical University of Munich, Germany
{simonm|gallenmu|carle}@net.in.tum.de

***Abstract*—Novel applications require a robust and reliable connection to provide the services for next-generation networks. The complex nature of these algorithms needs fast and efficient stateful processing. Using Software-defined Networking (SDN), new algorithms can be implemented into the network in a platform-independent way. The upcoming Portable NIC Architecture (PNA) for P4, a language to program data planes in SDN, allows inserting new table entries without controller interaction. Thus, it unleashes more performant and stateful applications without the overhead of the controller. We implement and evaluate these so-called 'add-on-miss' insertions introduced by the PNA for a P4 software target. In addition, we discuss the influence of latency and throughput optimizations on software packet processing systems. We determine the impact of these optimization strategies and which performance properties and costs can be measured with each. In our analysis, we model the costs of insertions based on an extensive baseline and compare them to table entry lookups and updates. We analyze the influence of the frequency of insertions and multi-core scenarios. Finally, we demonstrate that the approach scales for realistic scenarios.**

***Index Terms*—SDN, State Management, P4, Add-on-Miss**

I. Introduction

The upcoming 6G standard for communication networks will enable novel and complex applications, ensuring an ultra-low end-to-end latency as well as an ultra-low packet loss rate. Connections with these properties are essential for critical applications in domains such as transport, industry, and medicine. Optimized reliability methods are necessary to achieve these goals. An example of such an approach is hybrid automatic repeat request (HARQ). This algorithm increases the reliability of connections using forward error correction and repetition of non-acknowledged packets. Such complex algorithms must be distributed across different components in a network, either to the network interface card (NIC) or entirely to middleboxes to deal with demanding network applications.

P4 [1] is a platform-independent language to describe the data plane targeting high-performance, vendor-independent packet processing. With the upcoming Portable NIC Architecture (PNA) [2], P4 becomes a language to program both in-network switches and end-host applications. The latter is gaining attention due to efforts to bring P4 into the Linux Kernel [3]. Moreover, Intel announced that the SmartNIC E2000 will support the P4 language [4]. The capability of efficient state management becomes especially important when P4 programs are executed on the end of the communication path. Typical stateful scenarios include TCP flow tracking and the monitoring of connections.

The PNA enables stateful packet processing directly on the data plane. This new feature can speed up existing stateful P4 applications, such as IDS (e.g., *P4ID* [5]), stateful firewalls (e.g., *P4SF* [6]), or flow monitoring (e.g., *NetSeer* [7]). However, the statefulness of the P4 processing pipeline may introduce effects that are absent from the current generation of P4 devices, such as the impact on latency or jitter caused by state updates. The fundamental change in PNA requires a fundamental change to the measurement methodology used to investigate device behavior. Therefore, we establish a novel measurement methodology and apply it to a modified version of the P4 software target T4P4S [8]. This modified version supports *add-on-miss* insertions introduced by the PNA.

Our contributions can be summarized as follows: the definition of a measurement methodology focusing on the effects of stateful packet processing; the implementation of insertions in a software P4 target; the analysis of relevant performance indicators for PNA state updates; and the measurement and analysis for a comparison of costs for table entry lookups, updates, and insertions in a software P4 pipeline.

II. Background & Implementation

a) P4: P4 [1] provides a target-independent way of programming network forwarding devices, relying on compilers for different targets. This concept allows vendor-independent mechanisms and gives sovereignty to the network operator. So-called *externs* can utilize non P4-based extensions.

Several P4 targets exist, which can be classified as hardware and software targets. *Hardware targets* provide the highest performance in terms of throughput and latency. They usually follow a pipeline model with multiple stages executing specific subtasks of the program. Several packets are processed simultaneously but at different stages in the pipeline. This processing approach becomes important considering the consistency of state updates. *Software targets*, on the other side, provide the highest degree of flexibility. While their performance is lower, software targets run on commodity hardware and allow the easy integration of new functionality. Software targets typically follow the *run-to-completion* approach for packet processing. In this approach, different subtasks are handled by the same CPU core to avoid costly transfers of packets between different cores [9]. For our evaluation, we use T4P4S [8], which translates the P4 program to C code linked with DPDK [10], a userspace library for high-performance packet processing.

b) State Updates in P4: Data plane state in P4 is traditionally handled by *registers*. These externs provide index-based read and write access. However, they lack matching support to search for and select specific entries. The size and number of registers are limited, restricting the amount of maintainable state. Moreover, state may be fragmented in memory, negatively impacting performance.

In addition, *tables* can be used to maintain state. A table entry consists of the key(s) to be matched, associated with an action and the parameters to call the action. However, using traditional P4, table entries could only be modified by the control plane, including additional round trips. Table updates can be distinguished into two operations: new state can be inserted, meaning a new table entry with its keys, action, and associated parameters is created and inserted into the table. Additionally, existing table entries, i.e., the action parameters, can be modified after the lookup. The underlying table data structure has to provide consistency for both types: inserts and updates. For inserts, the modification of the data structure itself has to be synchronized to allow insertions of possibly multiple producers. For updates, the modification of each individual entry has to be synchronized to avoid race conditions and stale data shared by multiple consumers.

The data plane can request an update or insertion of an entry by sending a digest to the controller. The controller afterward decides to allow or deny the request and sends a notification to the data plane to trigger the update or insertion. This digest-based approach causes at least one RTT overhead (in T4P4S: hardcoded one-second-sleep). Allowing the data plane to change the table entries helps increase performance by avoiding the detour over the controller. This immediate reaction to table modifications facilitates the use of P4 in latency-critical applications.

c) Portable NIC Architecture: Different P4 models or architectures specify the capabilities of the used target. While the prominent P4 architectures (*v1model* and *Portable Switch Architecture (PSA)* are designed for in-network switches, the *PNA* standard focuses on bringing P4 to end devices, such as NICs. The PNA aims to offload specific tasks to the NIC to speed up functionality and to address requirements for state-keeping. For instance, packet processing tasks in end hosts, such as handling protocols like TCP or QUIC, tend to be more complex and require frequent state modifications.

The PNA targets may support table changes: *insertions* and *updates*. While insertions create a new table entry for a non-existing key, an update changes an existing entry. PNA allows `add-on-miss` insertions, which are performed on lookup misses and can be activated for given tables. These insertions are triggered with the same key, that caused the table lookup miss. Inside the default action code, a new extern `add_entry<T>()` allows adding a new entry to the table with a specified associated action.

Updates allow the action code to use the parameters on the left-hand side of an assignment. Changes to the write-back parameters are synchronized to the underlying table. Furthermore, PNA allows the specification of an expiry timer, after which the control plane may delete an unused entry. This can be useful if, e.g., protocol session state is no longer required, e.g., after a TCP session timeout.

d) Implementation: We use the modifications for updatable table entries implemented in previous work [11] as the basis of our implementation, which is available on GitHub [12]. It uses a lock-free hash table provided by DPDK, which is compatible with table updates. The lock-free mechanism applies an optimistic approach, checking at the end of the transaction whether there were concurrent changes and repeating the process if required. The consistency towards multiple insertions is ensured. However, the optimistic approach may be unsuited for heavy insertion scenarios with multiple threads. In that case, the transactions have to be restarted, potentially multiple times, to eventually reach a consistent state.

For the P4 code translation, two adaptions had to be made: The functionality of the new extern method `add_entry` has to be generated for add-on-miss enabled tables. The method requires the table name and the keys to be added, which are only implicitly given in the P4 source code. Therefore, we pass the table name to the action and recalculate the key inside the `add_entry` function. To minimize the overhead, the table name is passed only to default actions of add-on-miss tables.

e) Performance Indicators: The two main features of the PNA are the add-on-miss insertions of table entries and the possibility to update existing ones. Both interweave with each other to provide efficient state management. Therefore, an evaluation has to answer the following questions: 1) What is the cost of an insertion? How does it compare to the cost of lookups? 2) What is the maximum throughput when adding new entries? How does the insertion rate influence it? 3) How do the cost of insertions differ from the cost of updates?

Question 1 infers a worst-case analysis that only consists of insertions. The comparison to the cost of lookups gives the relative overhead for a target whose maximum performance can be different. We will use the cost model presented in Section IV to answer it. We investigate a more realistic use case in Question 2, where only a subset of packets causes state insertions. For instance, TCP connection tracking requires an insertion for a new flow, but most traffic will update already-known flows. Therefore, it is important to investigate state insertions at different rates to see their impact on the maximum throughput. This way, the programmer or network operator is able to infer requirements for a given use case. Question 3 helps to weigh the costs and effects of creating a new state or changing an existing one. I.e., it answers whether the usage of placeholder entries might help improve performance.

III. Related Work

a) Stateful packet processing architectures: Verdu et al. [13] propose a multi-layered architecture named *MLP* for packet processing that exploits parallelism as much as possible. Their results demonstrate that the well-established paradigms run-to-completion and software pipelining both come with drawbacks for stateful processing. Bianchi et al. [14] discussed *OpenState* as a way to maintain state inside OpenFlow applications. For that, an extended Finite State Machine XFSM is implemented in the data plane, avoiding controller interaction and splitting the tables into flow-tables and an XFSM table. With *Open Packet Processor* [15], they generalize the XFSM-based approach to run it on hardware. It still has similar concepts as OpenFlow and relies on a flow context table giving access to flow-related state. It allows more sophisticated stateful tasks than the basic OpenFlow match/action model. The approach is further extended into *FlowBlaze* [16], designed by Pontarelli et al., who implemented it for the NetFPGA platform. Sun et al. [17] follow a similar approach called *SDPA*, proposing a stateful "match-state-action" paradigm for Software-defined Networking (SDN). In contrast to Bianchi et al., they also claim to support indefinite state machines.

b) State Updates in P4: State update considerations using the P4 language also exist. Caiazzi et al. [18] present *Switcharoo*, implementing a key-value data structure into the ASIC-based P4 hardware target Intel Tofino. Their implementation runs entirely in the data plane, thus avoiding any overhead from the controller, enabling high performance for stateful applications. In previous work [11], we implement a way to update existing table entries in a P4 software target in the data plane. We also discuss, which consistencies must be maintained in state updates and how flow-related state differs from global state. *FlowBlaze* was implemented in P4 [19] and provides updatable state in registers that are mapped through a flow context table. It thereby introduces some indirection. The software target P4-DPDK [20] supports the PNA and its table state modifications.

c) Distributed Data Plane State: Data plane state may require network-wide synchronization. Luo et al. [21] implemented a framework named *Swing State* for state management and consistent state migration to other nodes. They implement a P4 prototype of the framework that piggybacks the state on live traffic and automatically identifies state to migrate with static analysis of the P4 program. *SwiSh* [22] is a state management layer for P4 programs. There, Zeno et al. implement different consistency protocols to distribute state and evaluate it using an Intel Tofino ASIC. Zhou et al. present *P4Update* [23], implementing distributed consistent network updates using P4. Consistency is ensured using local verification of the update messages, relieving the control plane.

While there is significant interest in (P4-based) stateful packet processing and its evaluation, there is also a lack of a concise measurement methodology for that. In this paper, we aim to provide such a measurement methodology and apply this to our implementation of a PNA software target.

IV. Methodology

Our performance evaluation aims to calculate the operations' costs, i.e., CPU cycles. Software packet-processing systems process batches of packets to reduce the I/O overhead from/to the NIC. The size of the batches influences the system's behavior, ranging from latency-optimized (l.-opt.), i.e., smaller batch size, e.g., one, to throughput-optimized (t.-opt.), e.g., batch size 32+. There also exist approaches to self-adjust the batch size according to the currently processed traffic [24]. The optimization towards one of these performance goals influences what and how the costs of the performed operations can be measured. In the following, we describe performance models for both optimizations, assuming constant batch I/O costs. Table I lists all used variables for the built models.

Table I: Variables and their units for the model

Variable	Description	Unit
n	Batch size	packets
B_n	I/O cost of batch with size n	CPU cycles
c_i	Processing cost of packet i	CPU cycles
c_{avg}	Average processing cost per packet	CPU cycles
f_{CPU}	CPU frequency / cycles per second	CPU cycles / s
C_n	Processing cost of batch with size n	CPU cycles

a) Batch Model: For the performance model, we assume that the I/O cost B_n for a batch is constant, depending on the size of the batch n. These costs include the transfer of the packets from/to the NIC and all preprocessing required to access the packets. Each packet i of a batch further requires processing costs c_i, which may be different for each packet. The cost of the whole batch C_n can be modeled as in Eq. 1:

$$C_n = B_n + \sum_{i=1}^{n} c_i \tag{1}$$

When achieving a packet rate of r, r/n batches are processed in the given time interval. Thus, f_{CPU} can be set equal to the costs per second, as in Equation 2:

$$f_{CPU} = C_n \cdot \frac{r}{n} = \left(B_n + \sum_{i=1}^{n} c_i \right) \frac{r}{n} \tag{2}$$

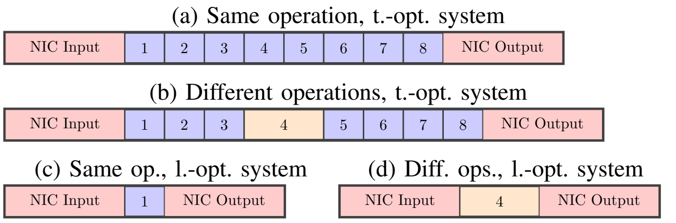
Increasing per-packet cost c_i, therefore, raises the number of CPU cycles spent on processing. In a run-to-completion model, this results in both a higher latency of the whole batch C_n and a decrease in the throughput r.

b) Throughput-optimized: A t.-opt. software packet-processing system aims for a larger batch size B_n, as the influence of n is minimal. A larger batch size is helpful to amortize this (constant) overhead B_n. On the other hand, the latency is increased since the first packet is not sent out until the last packet of the batch has been processed. Miao et al. [24] give additional insights into batched queueing costs.

In previous work [25], we derived the I/O overhead using a baseline scenario. To create the baseline, we measure a simple Layer 2 forwarder with minimal packet processing to approximate an I/O-only scenario. Using the CPU frequency f_{CPU} and the packet rate of the baseline r_{baseline}, we calculate the per-packet I/O overhead. The processing costs c_i are 0 in this case, inserting into Eq. 2, gives $C_n = B_n = f/r_{\text{baseline}}$. Using this baseline, and $c_{\text{avg}} = C_n/n$, Eq. 3 models the costs:

$$\frac{f_{\text{CPU}}}{r} = \frac{f_{\text{CPU}}}{r_{\text{baseline}}} + c_{\text{avg}} \Rightarrow c_{\text{avg}} = \frac{f_{\text{CPU}}}{r} - \frac{f_{\text{CPU}}}{r_{\text{baseline}}} \tag{3}$$

However, this model can only be used to calculate *average* costs, which infer that the operation performed on each packet


(a) Same operation, t.-opt. system

(b) Different operations, t.-opt. system

(c) Same op., l.-opt. system

(d) Diff. ops., l.-opt. system

Figure 1: Model of average (t.-opt.) and packet (l.-opt.) costs

Table II: Testbed specifications

Measurement	Intel Xeon CPU	RAM	Intel NIC
Throughput	E5-2620 v2 @ 6×2.1 GHz	128 GB	82599WS
Latency	D-1518 @ 4×2.2 GHz	32 GB	X552

should be the same, as depicted in Figure 1a. If the operations differ or the cost of the operation is not constant, as it is depicted in Figure 1b, the individual costs cannot be measured. In the example, we cannot calculate the different costs of the blue and the yellow (4) packets, but only the average cost of all measured packets. Since we are interested in comparing different operations, i.e., lookup and insertion of state, we have to switch to a l.-opt. version of the software target.

c) Latency-optimized: In a l.-opt. system, the batch size is minimized to improve latency at the cost of amortizing I/O expenses. Reducing output batch size alone might be enough for latency improvements. This investigation only discusses a shared batch size for input and output since the performance models are built on differences towards baseline scenarios. Reducing the throughput sufficiently also leads to a smaller batch size since the queues are only partially filled then.

Figures 1c and 1d show a reduced batch size of one. If packets cause operations with non-constant costs or different operations, these only affect the single packet of the batch. Therefore, a l.-opt. system is suited to measure the costs of each packet and not only the average cost.

We previously modeled the cost per packet, measuring the latency [26]. Again, we can compare different latencies l_i (in seconds) to a baseline scenario l_{baseline}, i.e., a forwarder, to calculate the cost c_i of packet i:

$$c_i = f_{\text{CPU}} \cdot (l_i - l_{\text{baseline}}) \tag{4}$$

Using our test setup, we can determine the latency of every processed packet. Following the performance model, we determine individual packet costs, even if they carry out different operations. We investigate both versions, optimized for throughput or latency. The first evaluates the maximum performance and, therefore, the practicability of the approaches. The latter analyzes the involved costs in detail.

V. Setup

a) Topology: For the evaluation, we use two different setups, cf. Table II. A two-host topology is used to measure the maximum throughput. The Device under Test (DuT) is interconnected using a 10 Gbit/s fiber link with the load generator (LoadGen). The LoadGen generates traffic using MoonGen [27], which is processed and forwarded by the DuT. For latency measurements, we use a three-host topology. Both links are mirrored using an optical splitter towards the Timestamper that timestamps all packets with a precision of 12.5 ns [28] for latency calculation.

b) DuT: The DuT runs T4P4S (based on DPDK 21.08) with the modifications required for state updates and insertions [12] on Debian Bullseye. It uses a P4 program that has one table performing a table lookup on a specified key in a packet header. Based on the existence of an entry in the one P4 table, the looked-up value is sent back using another header field. If there is no matching entry, an add-on-miss insertion is triggered. Upcoming lookups of the same key will eventually succeed afterward. Every packet is forwarded back to the originator. The batch size in the t.-opt. measurements is set to 32. For latency optimization, we turn off any draining and send out processed packets without waiting for the output batch to be filled, i.e., the effective *output* batch size is one.

c) Scenario: The LoadGen generates traffic in 300 flows, alternating the source IP address with a constant bitrate (CBR). All packets have a size of 84 B without CRC. The key k that is used for lookup by the P4 program cycles pseudorandomly through $k \in [0, m]$ in a way that the cycle hits every element exactly once before the cycle repeats, i.e., the period length of the generated sequence is m. Therefore, the experiment can be divided into two phases: 1) The first m packets will trigger an insertion, as the key is unknown. 2) The following packets will contain a key already in the table, so a lookup is performed. To measure the influence of different insertion rates r, every r-th packet contains a new key in the range $[m, 2m]$. That way, the first phase covers the insertion-only traffic. The second phase covers the usual case of rare insertions into a non-empty table. The explained scenario is typical for a newly started device, such as a stateful firewall. Shortly after starting, state of tracked connections is mainly inserted, after that, during regular operation state is mainly looked up.

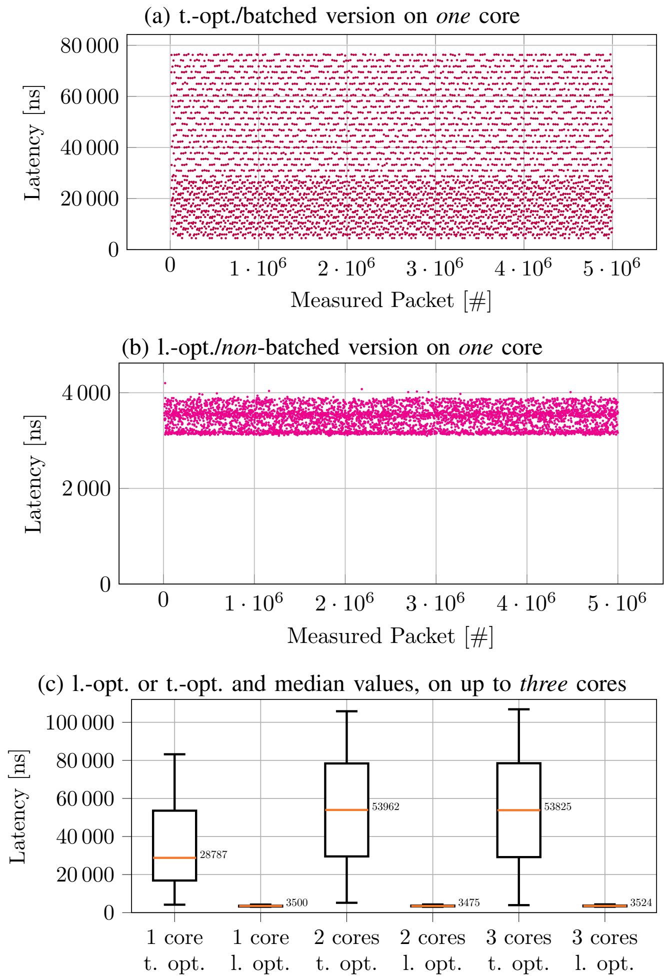
For the throughput measurements, the maximum bitrate is determined, which still achieves a packet loss of $< 0.01\,\%$. The rate is calculated with an accuracy of < 1 Mbit/s.

For the latency measurements, we generate traffic with a CBR of 300 Mbit/s. That way, we ensure to not overload the device. Using low-rate CBR traffic and a minimized batch size, we ensure measuring the cost, i.e., the latency, of each individual packet, while mitigating the influence of batching and queueing. The latency plots show each 997th packet to handle the figure sizes, but every insertion during the second phase is shown and all packets are considered for analysis.

VI. Evaluation

We first examine the performance of a forwarder to determine the baseline. Afterward, we dive into the performance evaluation of the insertions at different frequencies and compare them to lookups and table entry changes.

a) Baseline: We use a P4 forwarder program, which only sets the egress port, to measure the I/O overhead B_n. The



(a) t.-opt./batched version on *one* core

(b) l.-opt./*non*-batched version on *one* core

(c) l.-opt. or t.-opt. and median values, on up to *three* cores

Figure 2: Latencies of P4 forwarder

processing costs c_i equal 0 in this case. We investigate both the t.- and l.-opt. versions of T4P4S.

Figure 2a shows the occurring latencies of the batched version. The 32 batching stages can be clearly observed. Therefore, the latency has a high variance and a comparable high median of ≈28.8 µs (cf. Figure 2c). The measurements demonstrate that numerically smaller latencies occur more frequently. With a packet rate of 300 Mbit/s, the first few batching stages are filled more often and corresponding latencies happen more often. This observation can be confirmed when investigating higher packet rates.

The l.-opt. version, depicted in Figure 2b, has, as expected, a lower median latency of 3.5 µs (cf. Figure 2c) and the variance is small. However, the achievable throughput is reduced. The l.-opt. version achieves a maximum packet rate of ≈4.36 Mpps compared to a rate of ≈6.76 Mpps, a decrease of ≈54.9 %, in a single-core scenario (cf. Figure 4). Looking into multi-core scenarios, both versions scale. The median latency of the l.-opt. versions remains approx. constant (cf. Figure 2c). The latency of the t.-opt. version increases when using more than one core but remains similar when using more than two cores. The throughput of both versions (cf. Figure 4), however, scales linearly until hitting the line rate.

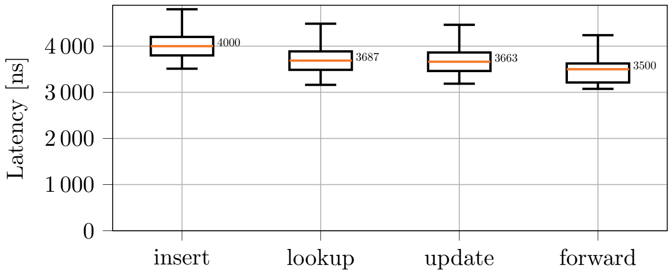


Figure 3: Comparison of base operations using *one* core

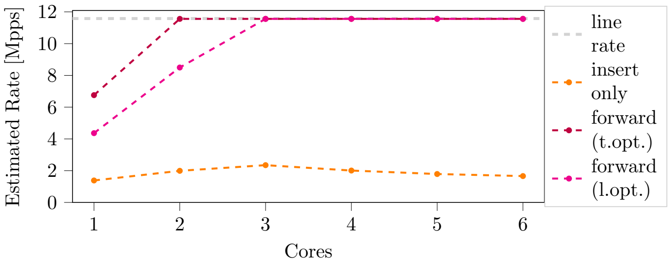


Figure 4: Comparison of throughput for insertion-only (first phase) and the optimized forwarders using up to *six* cores

To build our models in the following steps, our baseline single-core performances are r_{baseline}=6.76 Mpps for the throughput, and l_{baseline}=3500 ns for the latency measurements.

b) Insertions Only: First, we conducted the scenario explained in Section V-0c, with insertions *only* in the first phase, i.e. r=0. Figure 3 shows the latency of the two phases. The median latency of the first phase (insert) is ≈4000 ns, and the latency of the second phase (lookup) ≈3687 ns. Additionally, another experiment was performed, which updated the table entries by setting their value according to the header field of the incoming packet. Therefore, an update/change is performed instead of a lookup. Its median latency is comparable to the median of the lookup; its difference is less than two times the timestamp resolution.

The maximum achievable packet rates for the first insertion-only phase are depicted in Figure 4 (in orange). The packet rate starts with ≈1.38 Mpps using a single core and increases up to ≈2.35 Mpps using *three* cores. Afterward, the overhead of the optimistic locking mechanism becomes more dominant, and therefore, the performance decreases.

Table IIIa shows the calculated costs following the model of Eq. 4. The costs are modeled using the median measured latencies for each operation. In the model, we consider constant I/O and processing costs. The latency, however, is affected by additional, non-deterministic factors introducing variance to the measurements. An insertion is approx. two times more expensive than a lookup or insertion in a single-core scenario. This assumption only holds for batched insertions.

c) Insertion Rates: Batched table insertions may be used at the start-up but are unrealistic during regular operation.

Table III: Modelled Costs/CPU-Cycles

	Δ l [ns]	Cycles
Insertion	500	1100
Lookup	187	411
Update	163	358
Resolution	12.5	28

(a) Operations

Insertion Rate	Δ l [ns]	Cycles
1	500	1100
10	587	1291
100	649	1428
1000	912	2006
10000	1337	3941
100000	2749	6048

(b) Insertions with different rates

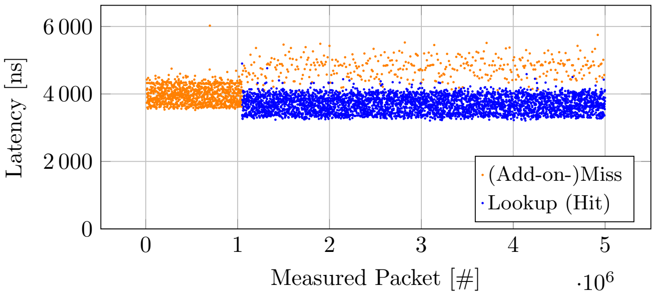
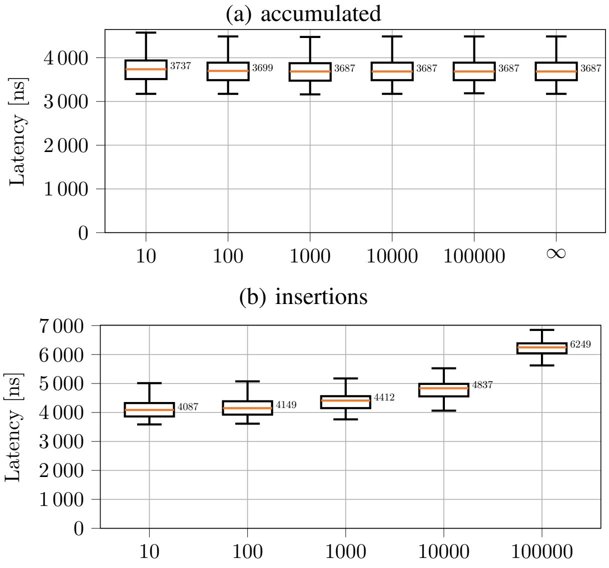


Figure 5: Latencies while inserting 2^{20} new entries through add-on-miss, followed by $\approx 4\,M$ additional packets, with an insertion rate of 10 000 using *one* core

Therefore, we now investigate how performance changes when the insertions happen at lower frequencies. For that, we include additional insertions into the second phase of the experiment.

Figure 5 shows the measured latencies when every 10 000-th packet triggers an additional insertion during the second phase. Still, the latency of the insertions (orange) in the first phase is lower than that of the lookups (blue). However, the additional insertions in the second phase have an increased latency compared to the lookups and the first phase's insertions.


(a) accumulated

(b) insertions

Figure 6: Comparison of different insertion-rates on *one* core

Figure 6b shows the occurring latencies for different insertion rates. The latency of the packets triggering an insertion increases, the less often these insertions happen. While the median latency is about 4087 ns with an insertion rate of *ten*, it rises to ≈6249 ns for a rate of *100 000*, an increase of ≈52.9 %.

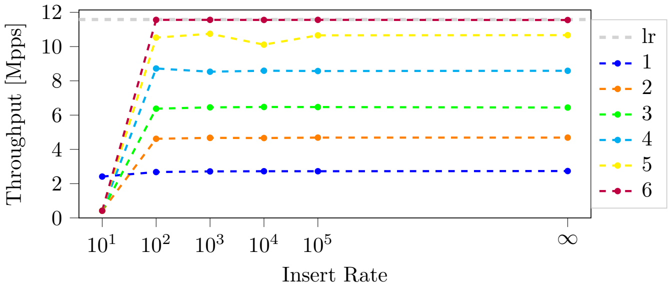


Figure 7: Maximum throughputs having insertions with different rates using up to *six* cores; line rate (lr) depicted in gray

Table IIIb shows the modeled costs for these different rates. The increased costs are likely due to worse cache optimization and branch prediction. Different branches of the compiled C program are taken when mixing different operations. This problem mainly concerns software targets since these run on a CPU with such optimizations. Hardware targets follow the pipeline approach and, therefore, come with constant latency independent of the executed branch.

On the other hand, the overall median latency for the mix of lookups and insertions slightly decreases with a decreasing rate as shown in Figure 6a. The insertions themselves are more expensive, but costs are amortized due to their rare occurrence.

Figure 7 depicts the maximum achievable throughputs for the different insertions rates. As explained, the throughput is an indicator of the average costs. The more costly insertions are amortized with rare insertions, and the achievable packet rates are approx. constant, starting with an insertion rate of 10^2. Additionally, for these realistic scenarios, the throughput scales linearly with the number of CPU cores used.

However, the packet rates guaranteeing a zero-packet loss behavior are reduced for an insertion rate of 10 in multi-core scenarios: In single-core scenarios, the throughput is increased towards the insertion-only performance: from ≈1.38 Mpps to ≈2.41 Mpps. The picture changes for multi-core scenarios. There, the performance drops to ≈0.42 Mpps independently of the number of cores. In this case, the mixture of operations and concurrent access decreases the performance of the underlying data structure. Although lock-free, the optimistic approach of the DPDK hashtable seems to be overloaded. The approach checks whether the table remained unchanged during the operations. In case it was altered in between, the operation is executed again. Fortunately, the limitation only exists for rather unrealistic frequencies of insertions.

VII. Discussion & Conclusion

In this paper, we implemented and evaluated add-on-miss insertions in a P4 software target. These on-the-fly insertions allow new applications to run in the data plane and improve performance by avoiding any overhead with the control plane.

The question, whether this is a step backward in SDN, may arise. The split into a fast data plane and a more complex control plane was made by intent. This separation leads to clear responsibilities and better performance. State updates and insertions in the data plane without controller interaction blur the concept to a certain extent. However, we argue that global and local state can work hand-in-hand. A globally maintained and potentially synchronized state between several nodes will still be needed. The controller is still required to ensure a consistent global view. On the other hand, the local state helps implement applications requiring flow and state tracking, but the kept state is optional for the general network behavior. Hence, it is not required that the control plane is kept informed about the local state. Moreover, the PNA proposal with the state updates originates from the P4 and SDN community. As the PNA brings P4 to the end-host, statekeeping is required anyway to offload applications to the NIC.

Our results show that the cost of insertions is approx. two times higher than table entry lookups or updates. Due to worse branch prediction and cache optimization, the insertion cost depends on the insertion rate, at least on software targets. Therefore, exceptionally high insertion rates (e.g., every 10th packet) on different cores lower the performance. However, these effects did not occur for the other, more realistic, rates, we measured in our investigation. At this point, the lock-free solution scales well in multi-core scenarios, considering the throughput and the baseline performance of T4P4S.

Limitations on the influence of insertion rates do not apply to hardware targets. Their pipelined architectures typically offer constant latency. All pipeline stages are traversed independently of the taken control flow. On the other hand, ensuring consistency becomes harder when many packets are processed at different stages, as a packet may change the shared state in a previous stage.

Acknowledgments

This work was supported by the EU's Horizon 2020 programme as part of the projects SLICES-PP (10107977) and GreenDIGIT (4101131207), by the German Federal Ministry of Education and Research (BMBF) under the projects 6G-life (16KISK002) and 6G-ANNA (16KISK107), and by the German Research Foundation (HyperNIC, CA595/13-1).

References

[1] 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," *Comput. Commun. Rev.*, vol. 44, no. 3, 2014.

[2] "P4 Portable NIC Architecture (PNA), version 0.5," Last accessed: 2024-09-13. [Online]. Available: https://p4.org/p4-spec/docs/PNA.html

[3] J. H. Salim, D. Chatterjee, V. Nogueira, P. Tammela, T. Osinski, E. Haleplidis, B. Sambasivam, U. Gupta, K. Jain, and S. Sethuramapandian, "Introducing P4TC - A P4 implementation on linux kernel using traffic control," in *EuroP4 2023, Paris, France*. ACM, 2023.

[4] B. Burres, D. Daly, M. Debbage, E. Louzoun, C. Severns-Williams, N. Sundar, N. Turbovich, B. Wolford, and Y. Li, "Intel's Hyperscale-Ready Infrastructure Processing Unit (IPU)," in *HCS 33, 2021*. IEEE, 2021.

[5] B. Lewis, M. Broadbent, and N. Race, "P4ID: P4 Enhanced Intrusion Detection," in *NFV-SDN 2019*, 2019.

[6] L. Teng, C.-H. Hung, and C. H.-P. Wen, "P4SF: A High-Performance Stateful Firewall on Commodity P4-Programmable Switch," in *NOMS 2022*, 2022.

[7] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, P. Zhang, D. Cai, M. Zhang, and M. Xu, "Flow Event Telemetry on Programmable Data Plane," in *SIGCOMM 2020*. ACM, 2020.

[8] P. Vörös, D. Horpácsi, R. Kitlei, D. Leskó, M. Tejfel, and S. Laki, "T4p4s: A target-independent compiler for protocol-independent packet processors," in *HPSR 2018*. IEEE, 2018.

[9] M. Dobrescu, N. Egi, K. J. Argyraki, B. Chun, K. R. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: exploiting parallelism to scale software routers," in *SOSP 2009, Big Sky, USA, 2009*. ACM, 2009.

[10] "DPDK," Last accessed: 2024-09-13. [Online]. Available: https://www.dpdk.org/

[11] M. Simon, H. Stubbe, D. Scholz, S. Gallenmüller, and G. Carle, "High-performance match-action table updates from within programmable software data planes," in *ANCS 2021, Layfette, USA*. ACM, 2021.

[12] "t4p4s at addonmiss· manuel-simon/t4p4s · GitHub," Last accessed: 2024-09-13. [Online]. Available: https://github.com/manuel-simon/t4p4s/tree/addonmiss

[13] J. Verdú, M. Nemirovsky, and M. Valero, "MultiLayer processing - an execution model for parallel stateful packet processing," in *ANCS 2008*. ACM, 2008.

[14] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, 2014.

[15] G. Bianchi, M. Bonola, S. Pontarelli, D. Sanvito, A. Capone, and C. Cascone, "Open Packet Processor: a programmable architecture for wire speed platform-independent stateful in-network processing," 2016. [Online]. Available: http://arxiv.org/abs/1605.01977

[16] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda *et al.*, "Flowblaze: Stateful packet processing in hardware," in *NSDI 2019*, 2019.

[17] C. Sun, J. Bi, H. Chen, H. Hu, Z. Zheng, S. Zhu, and C. Wu, "SDPA: Toward a Stateful Data Plane in Software-Defined Networking," *IEEE/ACM Transactions on Networking*, vol. 25, no. 6, 2017.

[18] T. Caiazzi, M. Scazzariello, and M. Chiesa, "Millions of low-latency state insertions on ASIC switches," *PACMNET*, vol. 1, no. CoNEXT3, 2023.

[19] D. Moro, D. Sanvito, and A. Capone, "Flowblaze.p4: a library for quick prototyping of stateful sdn applications in p4," in *NFV-SDN 2020*, 2020.

[20] "p4c/backends/dpdk at main · p4lang/p4c · GitHub," Last accessed: 2024-09-13. [Online]. Available: https://github.com/p4lang/p4c/tree/main/backends/dpdk

[21] S. Luo, H. Yu, and L. Vanbever, "Swing State: Consistent Updates for Stateful and Programmable Data Planes," in *SOSR 2017*. ACM, 2017.

[22] L. Zeno, D. R. Ports, J. Nelson, D. Kim, S. Landau-Feibish, I. Keidar, A. Rinberg, A. Rashelbach, I. De-Paula, and M. Silberstein, "SwiSh: Distributed Shared State Abstractions for Programmable Switches," in *NSDI 2022*, 2022.

[23] Z. Zhou, M. He, W. Kellerer, A. Blenk, and K.-T. Foerster, "P4Update: fast and locally verifiable consistent network updates in the P4 data plane," in *CoNEXT 2021*. ACM, 2021.

[24] M. Miao, W. Cheng, F. Ren, and J. Xie, "Smart Batching: A Load-Sensitive Self-Tuning Packet I/O Using Dynamic Batch Sizing," in *HPCC/SmartCity/DSS 2016*. IEEE, 2016.

[25] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of frameworks for high-performance packet IO," in *ANCS 2015, Oakland, USA, 2015*. IEEE Computer Society, 2015.

[26] S. Gallenmüller, J. Naab, I. Adam, and G. Carle, "5g qos: Impact of security functions on latency," in *NOMS 2020, Budapest, Hungary*. IEEE, 2020.

[27] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *IMC 2015*. ACM, 2015.

[28] Intel, "Intel ethernet controller x550 datasheet rev 2.6," 2021, Last accessed: 2024-09-13. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/333369/intel-ethernet-controller-x550-datasheet.html