Never Miss Twice – Add-on-Miss Table Updates in Software Data Planes

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

***Abstract*—State Management at line rate is crucial for critical applications in next-generation networks. P4 is a language used in Software-Defined Networking to program the data plane. The data plane can profit in many circumstances when it is allowed to manage some of its state without any detour over a controller. This work is based on a previous study investigating the potential and performance implications of add-on-miss state insertions. The state-keeping capabilities of P4 are limited regarding the amount of data and the update frequency. We follow the tentative specification of an upcoming Portable-NIC-Architecture and implement these changes into the software P4 target T4P4S. We show first results, which suggest that insertions are possible with only a slight overhead compared to lookups. Furthermore, the initial evaluation shows an influence of the rate of insertions on their latency.**

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

I. Introduction

Next-generation networks are required to process an increasing amount of data with an ever-decreasing allowed maximum latency. Critical applications in 5G or 6G networks rely on ultra-reliable, low-latency communication. To accomplish these goals, not only high-performant network devices are required, but also the capability of extending these devices by intelligent mechanisms. The P4 language [1] allows the target-independent programming of network data planes as part of Software-Defined Networking (SDN). SDN splits the network into two views: the *control plane*, which manages the network, e.g., by adapting specified rules like routing tables, and the *data plane*, which forwards traffic and efficiently applies these rules. The data plane and control plane must work hand-in-hand when it comes to more complex applications that require state. In case state (i.e., rules/table entries) has to be updated, the data plane has to reach out to the controller to digest its update proposal. This involves additional latency of at least one RTT, which hinders efficient processing. To get rid of this detour, the data plane should be able to maintain additional local state without the interference of the control plane. These updates include either the insertion or deletion of table entries or the modification of them. We investigated the latter in previous work [2]. This paper extends that work by implementing and studying the on-the-fly addition of new table entries using the software-target T4P4S [3]. A new entry is created with given parameters if there is no matching entry for a given lookup. Add-on-miss updates will also be part of the upcoming P4 Portable NIC Architecture (PNA) [4]. Both techniques, especially if combined, increase the capability of state management in P4 data planes. They enable additional data plane applications, e.g., flow data tracking.

The remainder is structured as follows: Section II provides background on P4 and table updates and gives insights into the implementation of add-on-miss insertions. Section III describes related work. Section IV evaluates its performance, and Section V finally summarizes our findings.

II. Background & Implementation

In this section, we provide general background information on P4 and table updates. Furthermore, we introduce add-on-miss insertions.

A. P4

P4 [1] is a domain-specific language to describe the behavior of data plane devices in SDN. P4 is designed as a target-independent language and abstracts from the underlying hardware. Compilers translate the programs exploiting features of the used target. There exist hardware targets, e.g., ASICs, FPGAs [5], SmartNICs, and software targets like *p4-dpdk* [6], bmv2 [7], and T4P4S [3]. For our implementation, we use T4P4S, which transpiles P4 programs into C-code that is linked with the Data Plane Development Kit (DPDK), a high-performant userspace packet-processing library. T4P4S is extensible and offers high performance in contrast to BMv2, which is meant as a reference implementation for prototyping. Due to the better performance, we are able to gain knowledge in the evaluation of realistic scenarios.

The P4 pipeline contains a *parser* that parses incoming packets according to a given finite state machine and header specifications. Afterward, the packets traverse match/action tables, determining performed actions and their parameters for specified header field values. In the end, the packet is *deparse*d again and sent out or dropped.

State can be maintained mainly using unstructured memory, i.e., *registers*. These are limited in size and number, inflexible, and have no matching support. Another way to work with data is by using the mentioned match/action tables. They support sophisticated matching (e.g., exact, lpm, ternary) and lay structured in the memory. P4 programs can be extended by non-P4, target-dependent *externs*.

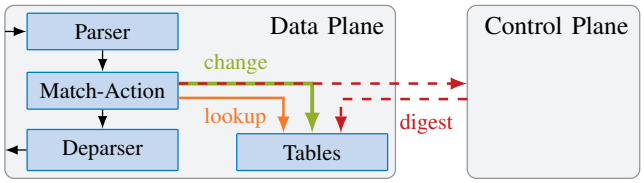


Figure 1: Control and data plane interaction for table updates (cf. [2]).

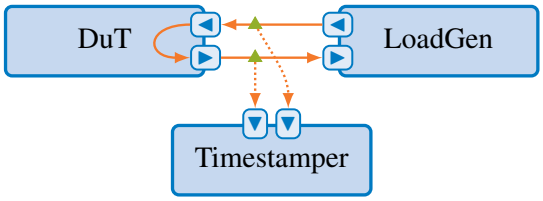


Figure 2: Three-host setup

B. Table Updates

Two update types must be distinguished: inserting and modifying table entries. In traditional P4, both can only be performed after the controller has initiated the update, leading to an additional overhead if the data plane wants to update its state (cf. Figure 1). Control and data plane communicate either in target-specific protocols or using the standardized P4Runtime [8] interface. In previous work [2], we extended T4P4S to allow modifiable table entries (green). This allows state updates at line rate. Modifiable parameters are marked using `@__ref`-annotation to be treated as references. It has to be ensured that consistency is maintained between multiple updates (inter-packet races) and between updates and possibly parallel insertions (insertion/update consistency). Both are ensured: using a lock-free, DPDK built-in mechanism for hash tables, ensuring insert-update consistency and per-entry locks avoiding inter-packet races. Annotations to the table definition allow the programmer to enable the mentioned consistency mechanisms, as required.

The same detour stands for table entry insertions. Traditionally, the controller handles insertions (red). Subsequently, allowing the data plane to add table entries would avoid any detours and decrease the overhead in latency. It has to be noted that table insertions are expected to happen less frequently than entry updates since this has to be done only once, e.g., per flow. The insertions are done if the corresponding lookup (orange) results in a miss. Then, a new entry is directly added to the table.

The PNA defines a new table property '`add_on_miss`,' specifying whether the mechanism is active for the table. Moreover, it defines an extern `add_entry<T>(tablename, params)` that allows adding new table entries. This extern can be called in the default action and is executed, in case of a miss. A code snippet on initializing new entries is given in Listing 1. The example code shows a simplified L2-forwarder. The `forward` table matches the source MAC address. In case of a miss, a new entry is added to the table using the broadcast MAC address as value.

The advantage of this approach is that the language is not adapted, but the functionality is provided using an additional hardware-dependent extern. This way, not every target has to provide the functionality. Additionally, the creation of a new table entry may be done conditionally. However, the conditional addition may cause problems, especially for hardware targets, when this is mapped to pipeline stages.

```
table forward {
      actions= {forward, add}
      key = {hdr.eth.srcAddr: exact;}
      add_on_miss = true;
      default_action=add;
}

action forward(bit<48> dstMac) {
      ...
}

action forward_add() {
      bit<48> dstMac = 0xffffffffffff;
      add_entry<forward_params_t>
            ("forward", {dstMac});
}
```

Listing 1: PNA add-on-miss entry initialization

III. Related Work

The upcoming Portable NIC Architecture (PNA) [4] tries to bring P4 to the end of the network and therefore requires more sophisticated state management to allow efficient hardware offloading, helping applications running on the end host. Many applications may profit from cheap state updates, e.g., flow monitoring [9] or IDS [10].

SwiSh [11] implements a distributed state layer to programmable switches. They evaluated different types of consistency and implemented the layer for the Intel Tofino. The approach shares state between several network nodes but still involves their controllers. FlowBlaze [12] (P4 implementation: [13]) implements data plane updates relying on registers instead of table entries. This approach uses native P4 features mapping registers using a flow context table. Therefore, it is widely compatible, but the functionality is limited since registers are constrained in terms of, e.g., total number and width.

IV. Evaluation

In this section, we present first results of our evaluation. We conduct experiments to measure the impact, i.e., latency, of the insertions. Thereby, we consider two scenarios: having an insertion-only pattern, and therefore only lookup misses, and a mixed one, in which only a tiny subset of packets cause lookup misses and trigger insertions.

a) Setup: For the evaluation, we use a three-host setup depicted in Figure 2 that allows us to measure the latency of every packet. One host is our Device under Test (*DuT*) running T4P4S on a single CPU core; the other host (*LoadGen*) generates packets having a size of 84 B using MoonGen [14].

Both hosts are connected via two 10 Gbit/s fiber links. They are each monitored using optical splitters, and the monitored packet stream is forwarded to the third host (*Timestamper*) that timestamps the packets before and after the DuT to calculate the latency [15]. For timestamping, we exploit the hardware timestamping capabilities of the Intel X552 NIC [16] with a resolution of 12.5 ns. Every generated packet contains a key that is used for matching, producing either a lookup or an add-on-miss. The DuT runs on Debian Bullseye on an Intel Xeon D-1518 @ 2.2 GHz CPU with 32 GB RAM.

We built a latency-optimized version of T4P4S originally based on commit [17] for our evaluation. Latency is optimized by isolating CPU cores and removing any draining. Moreover, the batch size is set to one, which means that one packet is read from the NIC, processed, and sent to the NIC again, one after another. Therefore, differences in latency are caused by the overhead in terms of the additionally required CPU cycles of the investigated approach. This way, we can see the performance indications of the investigated operations as clearly as possible.

b) P4 Program: The P4 program contains only one table, for which the add-on-miss feature is activated, similar to the previous example. The table is initially empty and filled later by the add-on-miss insertions. Generated packets contain a key in a header field that cycles through pseudorandom values. This key is used for the table matching, and every packet is forwarded back to the load generator.

c) Insertion Latency: The first conducted experiment measures the required time for insertions, i.e., the latency of packets leading to an insertion. The key values cycle through $[0, 2^{20})$ several times. The table is initially empty. The experiment can, therefore, be divided into two phases: The *first* 2^{20} packets do not match any entry and will, therefore, result in an insertion. Every subsequent request will be successfully looked up without any more insertions required in the *second* phase. In that phase, we may additionally add some keys above the key range of phase one, resulting in an insertion. This way, we can evaluate the performance for two scenarios: when only insertions occur and the average scenario of rarely happening insertions. The latter is likely the case for real-world applications. We set the sending rate of the load generator to 300 Mbit/s to avoid any influence from the load of the DuT. We filtered the amount of data visualized for the following latency plots due to the high number of data points. In both phases, only every 997th packet is shown; however, all latencies are displayed for the insertions in the second phase.

Figure 3 shows the observed latencies while processing five million packets in total. Insertions caused by add-on-misses are only performed during the first phase. There are no additional insertions in the second phase. The median latency during the insertion phase is higher, 3900 ns compared to 3600 ns. It can be seen that the performance is of the same magnitude when only lookups are performed, as it was also the case for table entry updates [2]. Our previous adoption of the table architecture towards a lock-free single replica table enables the implementation.

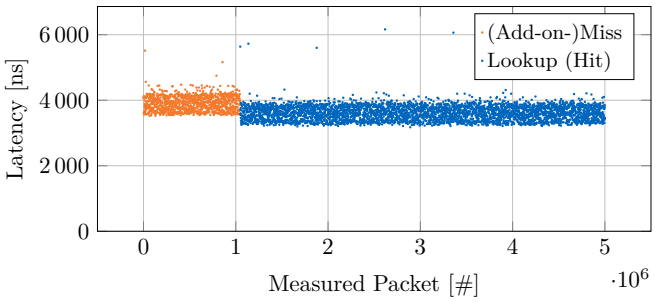


Figure 3: Latencies while inserting 2^{20} new entries through add-on-miss, followed by $\approx 4M$ lookup hits (300 Mbit/s)

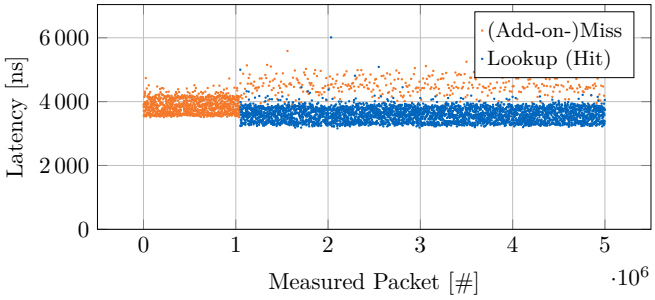
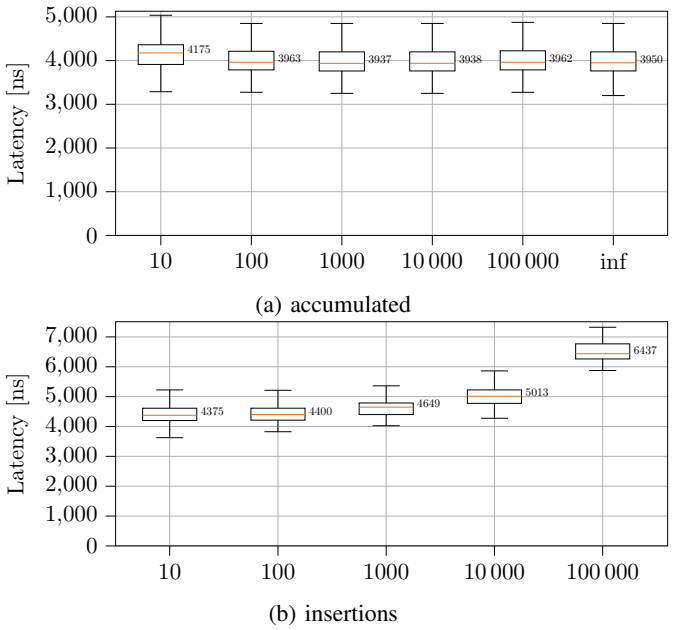


Figure 4: Latencies while inserting 2^{20} new entries through add-on-miss, followed by $\approx 4\,M$ additional packets, with an insertion rate of 10 000 (300 Mbit/s)

d) Insertion rates: While the first experiment shows the latencies when only insertions are performed, we now investigate isolated insertions at different rates. This is a more realistic use case since, e.g., new flows touch the device less often than known ones. Figure 4 shows again observed latencies, but now there is an add-on-miss with every 10 000-th packet during the second phase. The lookups still perform better than insertions. The latency of the insertions is higher when performed between lookups than when executed during an insertion-only phase. This is likely due to worse cache efficiency. Different branches of the compiled program are executed when mixing insertions and lookup, leading to a worse branch prediction. This is primarily an issue for software targets running on CPUs. Hardware targets mostly follow a different architecture using pipeline stages, leading to a more constant latency, independent of the taken branch in the control flow [18].

This trend continues with a decreasing rate of insertions. Figure 5 shows the latencies induced by different rates of insertions during the second phase. To show the trend more clearly, we increased the load to the device by raising the sending rate of the load generator to 800 Mbit/s. The accumulated (cf. Figure 5a) latency of all packets (both hits/lookups and misses/insertions) slightly decreases when fewer costly insertions are done. The reason is quite apparent. At the same time, the cost of each insertion (cf. Figure 5b) rises when

fewer insertions are performed. Having an insertion only every 100 000-th packet introduces a median latency of 6437 ns, which is an increase of 47.1 % compared to an insertion rate of 10. The reason for that was already discussed before.



(a) accumulated

(b) insertions

Figure 5: Latencies having insertions with different rates (800 Mbit/s)

V. Conclusion

This paper investigated an approach toward adding state in P4 data planes on the fly. If a table lookup results in a miss, a new entry is added by calling a special *extern*. This is possible with only a little overhead with regard to latency. Allowing table entries to be added by the data plane itself increases the possible low-latency applications. The evolution of P4 towards that is a highly welcome but also required step towards next-generation networks.

To what extent this trend is a step back away from the split view of control and data plane towards more intelligence in the data plane itself can be discussed. Conversely, the methods presented in the paper enable quick updates in the local state. Local state is especially suited for tasks like flow tracking inside the application. Advanced applications may still require a global state managed by the control plane, ensuring consistency between several nodes. Both state types help hand-in-hand to enable both fast and powerful applications.

Future work can extend this analysis by implementing a performance model and splitting the costs into different subparts. Additionally, this model can be built up on CPU cycles rather than latency. Moreover, it can investigate the maximum possible throughput in a throughput-optimized version. When it comes to multi-core scenarios, the mentioned different locking and synchronization strategies of the match/action tables come into play and their differences in performance can be analyzed for combined insertions and update patterns.

Acknowledgements

This work received funding by the Bavarian Ministry of Economic Affairs, Regional Development and Energy as part of the project 6G Future Lab Bavaria. Moreover, this work is partially funded by the German Federal Ministry of Education and Research (BMBF) under the projects 6G-life (16KISK002) and 6G-ANNA (16KISK107) as well as the German Research Foundation (HyperNIC, grant no. CA595/13-1).

References

[1] 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, pp. 87–95, 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[2] 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 '21: Symposium on Architectures for Networking and Communications Systems, Layfette, IN, USA, December 13 - 16, 2021*. ACM, 2021, pp. 102–108. [Online]. Available: https://doi.org/10.1145/3493425.3502759

[3] 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.

[4] "P4 portable nic architecture (pna), version 0.5," accessed: 2023-03-10. [Online]. Available: https://p4.org/p4-spec/docs/PNA.html

[5] S. Ibanez, G. J. Brebner, N. McKeown, and N. Zilberman, "The p4->netfpga workflow for line-rate packet processing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2019, Seaside, CA, USA, February 24-26, 2019*, K. Bazargan and S. Neuendorffer, Eds. ACM, 2019, pp. 1–9. [Online]. Available: https://doi.org/10.1145/3289602.3293924

[6] "P4 dpdk target (github repository)," accessed: 2023-03-10. [Online]. Available: https://github.com/p4lang/p4-dpdk-target

[7] "behavioral-model: The reference p4 software switch (github repository," accessed: 2023-03-10. [Online]. Available: https://github.com/p4lang/behavioral-model

[8] The P4.org API Working Group, "P4Runtime Specification," 2021, last accessed: 2021-10-21. [Online]. Available: https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html/

[9] 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 '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*. ACM, 2020, pp. 76–89.

[10] B. Lewis, M. Broadbent, and N. Race, "P4id: P4 enhanced intrusion detection," in *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2019, pp. 1–4.

[11] 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 *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 171–191.

[12] 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 *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 531–548.

[13] D. Moro, D. Sanvito, and A. Capone, "Flowblaze.p4: a library for quick prototyping of stateful sdn applications in p4," in *2020 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2020, pp. 95–99.

[14] 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*, ser. IMC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 275–287.

[15] S. Gallenmüller, F. Wiedner, J. Naab, and G. Carle, "How low can you go? A limbo dance for low-latency network functions," *J. Netw. Syst. Manag.*, vol. 31, no. 1, p. 20, 2023. [Online]. Available: https://doi.org/10.1007/s10922-022-09710-3

[16] Intel, "Intel ethernet controller x550 datasheet rev 2.6," 01 2021, accessed: 2023-05-24. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/333369/intel-ethernet-controller-x550-datasheet.html

[17] "P4elte/t4p4s - github: Commit a3a54e3," accessed: 2023-05-10. [Online]. Available: https://github.com/P4ELTE/t4p4s/commit/a3a54e37521dcc61365d09dd705c3709a533e07a

[18] E. Hauser, M. Simon, H. Stubbe, S. Gallenmüller, and G. Carle, "Slicing networks with P4 hardware and software targets," in *5G-MeMU '22: Proceedings of the ACM SIGCOMM Workshop on 5G and Beyond Network Measurements, Modeling, and Use Cases, Amsterdam, The Netherlands, August 22, 2022*, Ö. Alay and Y. Wang, Eds. ACM, 2022, pp. 36–42. [Online]. Available: https://doi.org/10.1145/3538394.3546043