High-Performance Match-Action Table Updates from within Programmable Software Data Planes

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

Abstract

For long, P4's mantra was that table entries could only be updated by the control plane. With the ongoing Portable NIC Architecture (PNA) standardization efforts, this is changing. In fact, PNA presumably includes explicit methods for table updates from within the data planes. Now, it is onto manufacturers and developers to integrate and use this mechanism in future P4 data planes. This would enable novel and improved applications, e.g., requiring means for maintaining state.

We present our implementation of flexible match-action tables for the DPDK-based `t4p4s` target. We discuss different approaches for table updates from within the data plane and challenges that arise when operating at line rate. Further, we analyze the data consistency of our enhanced table structures in a multi-core scenario and model the memory overhead for state management purposes.

CCS Concepts

• **Networks → Network measurement**.

Keywords

P4, SDN, Software Data Planes, State Management

ACM Reference Format:
Manuel Simon, Henning Stubbe, Dominik Scholz, Sebastian Gallenmüller, Georg Carle. 2021. High-Performance Match-Action Table Updates from within Programmable Software Data Planes. In *Symposium on Architectures for Networking and Communications Systems (ANCS '21), December 13–16, 2021, Layfette, IN, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3493425.3502759

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ANCS '21, December 13–16, 2021, Layfette, IN, USA
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9168-9/21/12…$15.00
https://doi.org/10.1145/3493425.3502759

1 Introduction

When P4 was introduced in 2014 [3], the language was targeted towards a specific class of devices: switches. Since then, P4's popularity created a diverse ecosystem. Today, P4 runs on devices such as SmartNICs [1, 21]. At the same time, "spin-off" standards were introduced, e.g., P4Runtime that allows managing P4 devices or P4 In-Band Network Telemetry to monitor P4 devices. Another addition to the family of P4 standards is the *Portable NIC Architecture (PNA)* [20]. PNA proposes a common set of P4 features specifying packet processing tasks on NICs.

P4 offers two fundamental mechanisms realizing stateful packet processing, *tables* and *registers*. Tables support sophisticated matching strategies but can only be read from the data plane. Changes to table entries must be initiated from the control plane, causing additional overhead. In contrast, registers can be modified directly in the data plane, offering low-level data access; however, they are P4 externs, i.e., not part of the core language. Therefore, availability of registers and their API and implementation details are target-specific. The PNA includes mechanisms to modify tables from the data plane, allowing fast updates and matching support.

In this paper, we present our implementation of table entries that are writable from within the data plane using the software target `t4p4s` [29]. We analyze the challenges for line-rate table updates and measure the performance impact for different match-action table data structures. In particular, we discuss approaches with respect to consistency. Finally, we demonstrate and model both the overhead for state management using our cache-efficient storage design and the performance benefit for in-data plane table updates.

The remainder of the paper is structured as follows: Sec. 2 discusses related work. Sec. 3 describes mechanisms employed by `t4p4s` to change match-action table entries. We discuss our measurement methodology in Sec. 4, before Sec. 5 introduces our approach. Sec. 6 presents our proposed changes to the table architecture and analyzes possible optimization approaches. Sec. 7 discusses applicability to other targets, before we conclude this paper in Sec. 8.

2 Background & Related Work

P4 Control Plane Interfaces. With PNA still in preparation, table updates are provided via the control plane. P4Runtime [28] is a standardized, but optional, control plane interface. Platform-specific update mechanisms are still commonplace. Without a built-in approach for deploying table updates, developers must rely on such an interface. Often, this implies relying on the vendor- and platform-specific implementations, differing greatly in supported functionality.

Benefits for Applications. The P4 landscape is broad as shown by large-scale taxonomies [13, 15]. Therefore, many applications [5, 26] may benefit from in-data plane table updates. In particular, [5] and [26] propose replacing target-specific register externs through table updates. As the latter are based on P4's core feature, match-action tables, replacing them reduces indirections, resource usage and improves code readability. We argue, that middleboxes, e.g., using counters for heavy-hitter detection [12] and flow monitoring [31], will benefit from data plane table updates. Table updates can ease maintaining state in security-related applications, e.g., in IDS [17], for port knocking [30], or SYN flood mitigation [24]. [24] note state management overhead caused by the lack of direct table manipulation from the data plane.

Implementation Challenges. The PNA aims to standardize the features of general NICs [4]. The specification [20] allows adding table entries on lookup miss, directly from the data plane using externs. The PNA authors discuss impacts on processing speed and restrictions of this mechanism on different targets, e.g., only support for exact match keys. Our implementation is not limited to lookup misses. Jain [23] presents the flexible match-action tables of Pensando SmartNICs [21]. These tables allow multiple accesses at different processing stages and table updates from the data plane via write-back table fields [26]. Using target-specific annotations, special action parameters can be defined and used to assign values. The compiler translates them to an extern call that updates the respective table field. FlowBlaze [22] (P4 implementation: [19]) is a framework for state updates in programmable data planes relying on registers, instead of writable table entries. Register state is mapped through a flow context table, thus introducing an indirection. This approach uses well-established P4 features and is compatible to many targets, but the functionality of registers is limited (i.e., total amount, width) and the data may be fragmented. [6] and [11] describe challenges to avoid state inconsistencies during updates in programmable hardware dataplanes.

3 Match-Action Tables in t4p4s

`t4p4s` [29] is a multi-target P4 compiler. It transpiles P4 to C. To support different backend targets, the C code relies on the Network Abstraction Layer, an interface that defines basic

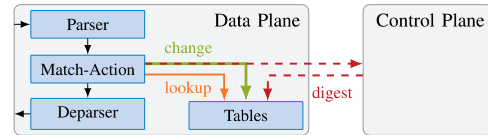

Figure 1: `t4p4s`' data and control plane interaction for table changes. Original: dashed red. Proposed: green.

functions like modifying packets or looking up table entries. These functions are then implemented for the specific target such as the Data Plane Development Kit (DPDK) [8].

`t4p4s` offers support for multi-core and multi-socket CPUs. With the help of DPDK, it performs Receive Side Scaling (RSS). There, flows are split into different queues by the NIC. The queues are then processed independently and in parallel by different threads/logical cores (lcores).

`t4p4s` follows the P4 concept offering read-only access to tables in the data plane (cf. Fig. 1). To perform table updates, the data plane sends a digest to the controller that may apply or deny the update. Digest-based table updates introduce at least one round trip time (RTT) overhead. Changing entries directly in the data plane avoids this overhead (cf. red path in Fig. 1). `t4p4s`' control and data plane are independent processes, running on the same device, communicating via sockets. Therefore, the RTT is lower compared to a remote controller. Digests are sent from the data plane process to the controller, which replies to the data plane. A message handler thread of the data plane processes the reply. However, changing entries using digests in `t4p4s` involves a 1 s wait time for synchronization, blocking the packet processing in the corresponding thread. This approach renders the current implementation impractical for frequent table updates.

The tables themselves are also synchronized for concurrent access. `t4p4s` uses lock-free double buffering consisting of two replicas of the same table. Each thread/lcore maintains a pointer to the currently active replica. Changes, e.g., inserts, are first performed on the *passive* replica. Afterward, *active* and *passive* replica are swapped. Then, a 200 µs wait time is applied to provide consistency for other threads reading from the now *passive* replica in parallel. The wait time must be larger than the time a packet remains in the pipeline in the worst-case to ensure consistency. Finally, the changes are promoted to the now *passive* replica. The wait time blocks the processing of the current thread. Reading entries remains unlocked since the active replica is never modified, thus gaining performance.

To our knowledge, both wait times (1 s and 200 µs) where chosen by the original `t4p4s` developers without further reasoning. Timing-based synchronization has the drawback that the wait time has to be chosen carefully. Unnecessarily high wait times increase latency. However, without proof for an upper bound, timing-based synchronization does not guarantee consistency. `t4p4s`' wait time for reacting to digests

is rather high; however, there is still no guarantee that neither the controller nor the message-handler thread of `t4p4s` got overloaded in the meanwhile. Therefore, the mechanism does not ensure table consistency (cf. Sec. 5).

4 Evaluation Setup

Here we discuss the testbed topology used for evaluation.

Topology. The load generator is directly connected to the device under test runnig `t4p4s`. We use `t4p4s` commit `9b91a136` based on DPDK 19.02, which is also the reference for our evaluation. MoonGen [10] is used as load generator to measure throughput and latency using hardware timestamping. Unless stated otherwise, the generated traffic consists of 84 B UDP packets. Each host features two *Intel Xeon CPU E5-2620 v2* (2.1 GHz, 15 MiB L3 cache). The used cores are isolated from the kernel scheduler.

Design. During evaluation, table entries are read and changed. We use 4-byte match keys and entries. An incoming packet specifies an entry to be changed, and its new value. The previous value is read and sent back. This way, one table entry is read and changed for each packet. Heuristic-based prefetching caches sequential memory accesses increasing performance [27]. To measure the worst-case performance, tests need to maximize the cache misses. Therefore, we randomize the table accesses.

5 Data Plane Table Updates

This section presents our implementation for data plane table updates, evaluates performance, and discusses multicore synchronization. Similar to Baldi et al. [4], we annotate modifiable table entries using `@__ref` in the parameter list of an action. This allows value updates that are propagated to the table. Subsequent reads must only return the updated value. Consequently, this change enables more sophisticated state-keeping operations, e.g., counters in tables.

5.1 Implementation & Evaluation

Here, we discuss different approaches for table updates. The first approach is similar to the original implementation of `t4p4s`. It uses `t4p4s`' lock-free double buffering mechanism (cf. Sec. 3, subsequently named *change method*), but bypassing the communication between data plane and controller. It requires another lookup of the entry and triggers the change using the double-buffering mechanism. This involves overhead and lowers performance. Therefore, we introduce a second approach, changing the value directly using its pointer (named *pointer method*). In this approach, the generated C hash table representing the P4 match-action table passes a pointer to the action instead of the parameter values themselves, thus enabling read and write access. We expect a higher performance when bypassing the original mechanism.

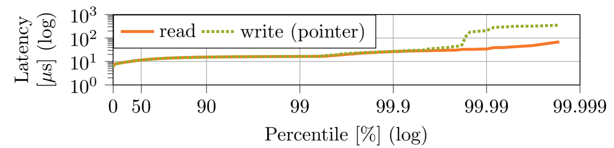

Figure 2: HDR latency histogram; 84 B packets

Method	Pointer	Change	Digest (*orig.*)	Digest (*mod.*)
Throughput	1.73 Mpps	3.39 kpps	<1 pps	4.1 kpps
25/50/75/99- $P_{k\%}$ latency (µs)	25.0/26.5/ 28.0/29.7	267/322/ 639/1108	- -	39.6/65.3/ 85.3/107.0
Synchronized	no	yes	yes	no
Sleep-Time	-	200 µs	1 s + 200 µs	-

Table 1: Data plane table update method comparison

Lastly, we investigate the digest-based way of changing entries. For the evaluation we use a table size of $2^{20} (\approx 10^6)$.

Pointer method. Fig. 2 compares the single-core latency of reading / writing table values via the *pointer method*. We measured a similar throughput (3.58 Mpps (read), 3.57 Mpps (write)), as well as latency distributions for lower percentiles. Writing increases tail latency, which we attribute to the overhead of memory writes. However, this deviation happens rarely. In general, we measured little overhead for modification compared to read-only accesses.

Change method and digests. Tab. 1 lists the single-core latency quartiles of all approaches. Maximum throughput is limited for the change and digest method. Since MoonGen cannot reliably generate rates below 15 Mbit/s, we used a packet size of 700 B, to compare all approaches. In this setting, the *pointer method* has the lowest latency (26.5 µs) while allowing a throughput of 1.73 Mpps. The *change method* has the highest median latency of 322.0 µs. Additionally, latency variance is increased, and the achievable throughput is only 3.39 kpps. To measure the impact of the timing-based synchronization, we disabled the sleep time resulting in a throughput of 1.12 Mpps and a median latency of 27.9 µs. From intuition, we believe the difference to 322 µs (latency with sleep) exceeding 200 µs is due to the unprecise `usleep` call, sleeping for *at least* the given time [14].

For comparison, we also evaluated *digests*. But, since there is the timing of 1 s, which would allow less than 1 pps, this approach hardly works at all in `t4p4s`. When disabling this mechanism, as happened in our experiment, the median latency is 65.3 µs. However, we had to limit MoonGen to 23 Mbit/s (4.1 kpps). Otherwise, the switch stopped due to an *out of memory error*. We think that that error is caused by allocating memory for every message sent to the controller.

5.2 Discussion

The original *digest method* performs worse by a whole magnitude than every other candidate owing to the prototype nature of its implementation. We modified the *digest method*

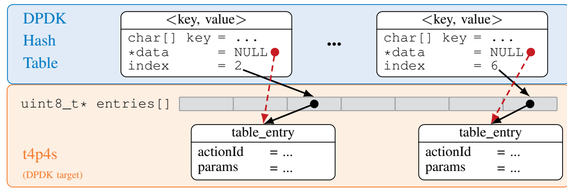

Figure 3: Orig. (black) and dyn. (red) storage design

to get an approximated performance by removing the timer-based synchronization. Despite the improved performance of the modified approach, the digest-based approaches share a fundamental flaw: the one RTT update delay between data and control plane. To mitigate this, we created two methods directly modifying the table entries removing the control plane from the equation. The *change method* is near to the original implementation but still subjected to a timing-based synchronization overhead—leading to unnecessary high latency and low throughput figures. The *pointer method* offers the best performance, entirely bypassing synchronization mechanisms. However, it is not compatible with double-buffering. Therefore, it requires an alternative low-overhead synchronization method having only one replica.

6 Table Architecture

We identified the *pointer method* as the most performant solution. Here, we modify the table architecture to ensure data consistency. Thus, we replace `t4p4s`' synchronization mechanism with another one provided by DPDK. Additionally, we implement a lock for entries to avoid inter-packet data races. Last, we improve the storage design to be more cache-efficient. The synchronization mechanism is independent of the storage design. Therefore, we investigate both separately and name the following plots according to the tuple: *(storage design/sync. mechanism)*.

Tables in `t4p4s` use DPDK's hash table implementation [9] that matches the keys to their corresponding entries. Entries are maintained by `t4p4s` and contain, for instance, the associated action and their parameters. `t4p4s`, therefore, also maintains the memory allocations for its table entries, as depicted in Fig. 3 (black arrows).

6.1 Consistency

Maintaining consistency can be split into two subproblems concerning the two different parts of the tables: after inserting and removing, and after updates.

The first subproblem affects the DPDK hash table part. When a new entry is inserted, the hash table adds the key to its table as well as the link to the corresponding entry, which is created by the `t4p4s` part. So far, the DPDK hash table was run without support for multi-core execution since the synchronization was done by `t4p4s`. We refer to this synchronization mechanism as *t4p4s*. Since this design cannot

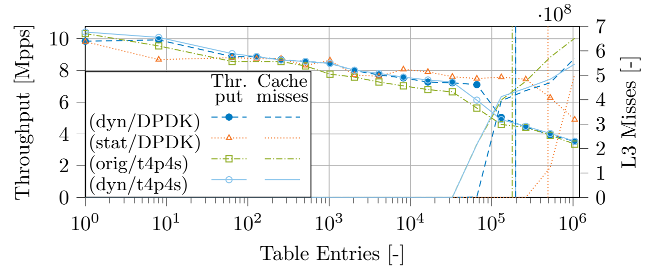

Figure 4: Throughput and L3 cache misses for different storage designs; cache models drawn vertically

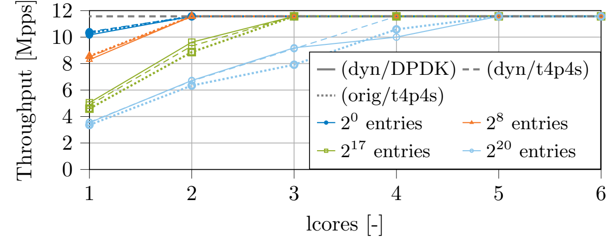

Figure 5: Achievable throughput with different designs

provide consistency when using the *pointer method*, we replace this mechanism and use only one replica. In this replica, a lock-free synchronization mechanism of DPDK hash tables is enabled. We name this mechanism *DPDK*. The optimistic method of DPDK checks whether the table was changed by others during the insertion using an atomic counter incremented with each change. If the counter was altered during the lookup the procedure is retried until it succeeds. Hence, this approach is beneficial for performance for a low number of insertions.

Fig. 4 depicts the maximum achievable throughput in single-core execution of the synchronization mechanism of `t4p4s` (*dyn/t4p4s*) and DPDK (*dyn/DPDK*) using the later introduced improved dynamic storage design (*dyn*). On single-core, the *DPDK* mechanism performs only slightly worse than the *t4p4s* one. Having *one* entry, the throughput is 10.40 Mpps with *t4p4s* synchronization, but only 10.16 Mpps with *DPDK*. For 2^{17} entries, the *DPDK* mechanism performs better (5.04 Mpps) than *t4p4s* (4.86 Mpps). The *DPDK* mechanism scales better in multi-core scenarios (cf. Fig. 5).

Following the adaptation of the table architecture, the second challenge is to prevent inter-packet data races. Parallel packet processing with simultaneous changes to the same entry can lead to data races causing, e.g., lost updates [16]. To solve that, we introduce optional locks for each entry. The entry is locked when the execution of its action starts and is released afterward. We use spin-locks that force threads into busy waiting until being released. This busy waiting might waste CPU cycles, however, waking up sleeping threads introduces context switches and, thus, additional latency.

Fig. 6 depicts the achievable throughputs with (*synced*) and without (*unsynced*) locking the entries. As expected, the

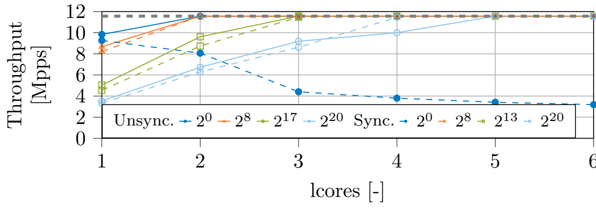

Figure 6: Throughputs using *(un-)synced* table entries

performance decreased if entries were locked. Having only one entry, the throughput is decreased by 5.9% on single-threaded execution. Additionally, for one entry, this approach does not scale for multi-core execution. The reason is that all packets match the same entry, but only one core can access the action at a time. Furthermore, the additional overhead of locking negatively influences performance. For larger table sizes, the mechanism scales, eventually hitting line rate.

6.2 Cache-efficient Storage Design

After ensuring data consistency, we further improve performance by implementing a cache-efficient storage design.

Recall Fig. 3 which shows the *original* table design (*orig*). There, the DPDK hash table maintains indices. These are used to access an array that stores pointers to the value structs. This design introduces two levels of indirection decreasing performance. As a first step, we removed the second level and directly stored the pointer to the entry in the hash table (red). This improves performance, as shown in Fig. 5. Performance is improved for two reasons: saved dereferencing cost and lower memory footprint improving cache utilization. Since the entries themselves are allocated dynamically upon insertion, we call this design *dynamic*.

For the analysis, we define $A = \max_{a \in \text{actions}} (\texttt{sizeof}(params(a)))$ being the action parameter size, n being the number of table entries. s is the maximum table size. The alignment function is defined as $\lceil x \rceil_y = \text{align}(x, y) = y \times \lceil x \div y \rceil$. We extend the memory consumption model proposed by Scholz et al. [25, Eq. 4] for the original design as follows:

$$m_{orig}(n, s, A) = \underbrace{s \times \lceil 8 + k \rceil_{16}}_{\text{DPDK hash table, keys and data pointer}} + \underbrace{n \times \lceil 4\,\text{B} + A \rceil_{64}}_{\text{dyn. allocated entries, each aligned to 64 B}} + \underbrace{8\,\text{B} \times s}_{\text{array with pointers}}$$

The reason for the alignments lies in the memory allocation of DPDK which itself tries to optimize the memory layout and allocates whole cache-lines. The calculated memory consumption then leads to the cache fitting model by setting n equal s, i.e., filling the table completely:

$$n_{orig}(m, k, A) = m / (\lceil 8 + k \rceil_{16} + \lceil 4\,\text{B} + A \rceil_{64} + 8\,\text{B}) \quad (1)$$

The improved design reduces the memory consumption (m_{dyn}) by the size of the skipped array shown in Eq. 2; the corresponding cache model in Eq. 3.

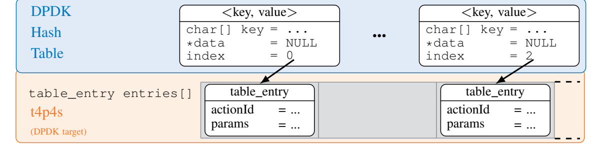

Figure 7: *Static* storage design

$$m_{dyn}(n, s, A) = s \times \lceil 8 + k \rceil_{16} + n \times \lceil 4\,\text{B} + A \rceil_{64} \quad (2)$$

$$n_{dyn}(m, k, A) = m / (\lceil 8 + k \rceil_{16} + \lceil 4\,\text{B} + A \rceil_{64}) \quad (3)$$

For a 15 MiB L3 cache size and an action size $A = 9\,\text{B}$, the number of entries fitting in the cache can be calculated using Eq. 1 $n_{orig} = 178734$ for the original design, and $n_{dyn} = 196608$ for the improved *dynamic* design using Eq. 3. The results depicted in Fig. 4 show that, except for 2^{17} entries, the measured cache misses are lower for the *dynamic* design.

Removing the second indirection leads to a performance gain. Yet, the actual table entries are still fragmented in memory since the memory for each entry is allocated dynamically when inserted. This reduces the amount of unused allocated memory for partially filled tables; however, having all entries in one contiguous memory block may improve performance. The continuous memory usage improves the spatial locality of data, allows more efficient prefetching of data into caches, and subsequently increases performance.

To ensure this spatial locality, we reintroduced the array strategy (cf.Fig. 7). But, instead of a pointer, the table entry structs are directly stored in the array. This strategy offers a single level of indirection. Hence, we can align the entries to 16 B instead of 64 B as before, saving memory for small entries, which allows more entries to lay in the cache, as a second advantage. We call this *static storage*. Eq. 4 shows the modeled memory consumption; Eq. 5 the cache model:

$$m_{static}(s, A) = s \times \lceil 8 + k \rceil_{16} + \underbrace{s \times \lceil 4\,\text{B} + A \rceil_{16}}_{\text{static sized array containing the entries, each aligned to 16 B}} \quad (4)$$

$$n_{static}(m, k, A) = m / (\lceil 8 + k \rceil_{16} + \lceil 4\,\text{B} + A \rceil_{16}) \quad (5)$$

A drawback of the *static storage* is the potentially higher memory consumption for scarcely filled tables. There, the consumption depends only on the maximum table size, while it linearly increases with the filling rate for the *dynamic* one. The relevant factor for selecting the most efficient table design is the filling rate of the table. The minimum filling rate, so that the *static* storage outperforms the *dynamic* depends on A due to the alignment to 16 B. The minimum rate for $A \leq 12$ is 25 %, 50 % for $12\,\text{B} < A \leq 28\,\text{B}$, 75 % for $28\,\text{B} < A \leq 44\,\text{B}$, and 100 % for $44\,\text{B} < A \leq 60\,\text{B}$. Similar calculation can be done for $A + 4 > 64\,\text{B}$.

For $A = 9\,\text{B}$, the number of table entries fitting according to Eq. 5 equals $n_{static} = 491520$, which is much more than for the *dynamic store*. This can also be seen in the evaluation (*stat/DPDK* vs. *dyn/DPDK*). The cache misses start to rise

for a higher number of table entries than for the *dynamic* one. Therefore, leading to better performance, especially in the range between the cache fitting sizes. For 2^{19} entries, the *static storage* achieves 6.28 Mpps, while the *dynamic* only manages 3.95 Mpps, a 60.0 % gain. For smaller table sizes, both storage designs perform similarly. The cache misses start to rise left of the calculated sizes since the cache is used by other parts of `t4p4s`, other processes, and the OS.

6.3 Discussion

Using the *pointer method* to modify table entries directly raised the need for adapting the synchronization mechanism. This could be done without a major performance penalty since we were able to improve the storage by removing one indirection. Though, if it is known that nothing will be added to or deleted from the table, then there is no need to change the mechanism introducing at least a slight performance loss. The same stands if no entries should be changed at all.

Locking each entry separately to prevent inter-packet races introduces a performance loss of about 10 %, due to the locking itself and the extended entry size. Locking is only required if the write access is global and, therefore, flow-independent. RSS results in a static flow-to-thread mapping, i.e., a flow is always processed by the same thread. Flow-dependent, as well as read-only access, renders locking unnecessary, thus saving overhead.

The *static storage* design leads to fewer cache misses and lower memory consumption for high filling rates. In case table sizes are highly predictable, the *static storage* is a good choice offering high performance. However, low filling rates waste memory, decreasing performance. The *dynamic* design offers higher flexibility maintaining high performance for low filling rates. But, for small action sizes, memory is lost due to the alignment to the cache line size. The cache utilization is worse because of the fragmentation in the memory.

7 Application to Other Targets

Table information has a decisive role determining the processing of a packet. Making table entries changeable, leads to fundamental challenges. As entries may change at any time, also during packet processing, we must ensure consistent state information for a specific packet. There are two possibilities to handle this kind of consistency—we can ensure this consistency in software (SW) or in hardware (HW) that supports specific instructions to ensure consistency.

In this work, we focus on the software target (ST) `t4p4s`. STs usually work on DRAM that lacks HW, but allows SW synchronization mechanisms. Our implementation optimized synchronization mechanisms to lower the impact on throughput or latency. This optimization allows us to free CPU resources to be used for packet processing.

STs follow the "run-to-completion paradigm" minimizing in-memory moves [7], hardware targets (HTs) usually pipeline the packets. Thus, the concurrency models and the required locking for the different state types (global and flow state, as also discussed by [6, 11]) are different. HTs outperform ST, as they can dedicate specialized functional units to synchronization. This hardware offload minimizes the impact on other system resources resulting in higher throughput paired with low latency. [2] has shown rates up to 200 Gbit/s while managing state updates with each packet.

Despite this disadvantage we still consider STs relevant due to different use cases. STs support 1 TB or more of cheap DRAM, whereas HTs offer memory in the lower GB range.

8 Conclusion

Currently, P4 offers only rudimentary state keeping using registers. The PNA introduces mechanisms for table manipulation directly from the data plane, promising sophisticated data structures and high update rates. In this work, we present table updates for the DPDK-based P4 software target `t4p4s`. We replaced the table structure through DPDK's data structures for high performance and multi-core scalability. Benchmarks show 10 Gbit/s throughput on a multi-core server with each packet causing a table update.

Our implementation provides several optimizations featuring configurable table entry synchronization and table growth characteristics. Microbenchmarks demonstrate that enabling dynamically changing table sizes or synchronized table entry updates lowers the performance of the P4 program. Therefore, our implementation allows activating both features independently and conveniently through annotations in the P4 program. The required features are inherently linked to the implemented algorithm; thus, it is left to the P4 developer to enable the required table features. We provide models to predict the tables' memory consumption to choose the best table design. Our discussions may help P4 target developers choosing their concurrency and table design.

We argue that table updates from within the data plane are a crucial extension for the future of the P4 landscape. While the proposal of the PNA is a first step, more general approaches, allowing table updates not only on lookup misses, should be discussed. Further, target-specific side-effects, e.g., synchronizing table entries, should be mentioned. To encourage other developers to adapt or enhance our proposed mechanisms, we published the source code at GitHub [18].

Acknowledgments

This project was funded by the Deutsche Forschungsgemeinschaft (DFG) (Modanet, grant no. 397973531) and the Bavarian Ministry of Economic Affairs, Regional Development and Energy as part of the project 6G Future Lab Bavaria.

References

[1] Agilio CX SmartNICs. 2021. Netronome. https://www.netronome.com/products/agilio-cx/ Last accessed: 2021-10-03.

[2] Mario Baldi, Diego Crupnicoff, and Silvano Gai. 2020. Programmable Dataplane Architecture for Distributed Services at the Network Edge. In *2020 Seventh International Conference on Software Defined Systems (SDS)*. 107–114.

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *Computer Communication Review* 44, 3 (2014), 87–95.

[4] Gordon Brebner, Mario Baldi, and John Cruz. April 2020. Plenary: P4 Use Cases for Programmable NICs. http://www.opennetworking.org/wp-content/uploads/2020/04/Plenary-4-Slide-Deck.pdf in P4 Expert Roundtable Series.

[5] Antonio Capone and Carmelo Cascone. November 2015. openstate.p4 - Supporting Stateful Forwarding in P4. https://static.sched.com/hosted_files/2ndp4workshop2015/80/Politecnico%20di%20Milano%2C%20P4%20Workshop%20Nov%202015.pdf in 2nd P4 Workshop by Stanford/ONRC.

[6] Carmelo Cascone, Roberto Bifulco, Salvatore Pontarelli, and Antonio Capone. 2018. Relaxing State-Access Constraints in Stateful Programmable Data Planes. *SIGCOMM Comput. Commun. Rev.* 48, 1 (April 2018), 3–9.

[7] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. 2009. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 15–28.

[8] DPDK. 2021. Data Plane Development Kit. https://www.dpdk.org/ Last accessed: 2021-10-03.

[9] DPDK. 2021. DPDK documentation—Hash Library. https://doc.dpdk.org/guides/prog_guide/hash_lib.html Last accessed: 2021-10-14.

[10] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: A Scriptable High-Speed Packet Generator. In *Proceedings of the 2015 Internet Measurement Conference* (Tokyo, Japan) *(IMC '15)*. Association for Computing Machinery, New York, NY, USA, 275–287.

[11] J. H. Han, P. Mundkur, C. Rotsos, G. Antichi, N. Dave, A. W. Moore, and P. G. Neumann. 2015. Blueswitch: enabling provably consistent configuration of network switches. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 17–27.

[12] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. 2018. Network-Wide Heavy Hitter Detection with Commodity Switches. In *Proceedings of the Symposium on SDN Research, SOSR 2018, Los Angeles, CA, USA, March 28-29, 2018.* ACM, 8:1–8:7.

[13] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. 2021. A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research. *CoRR* abs/2101.10632 (2021). arXiv:2101.10632 https://arxiv.org/abs/2101.10632

[14] Michael Kerrisk. [n.d.]. *usleep(3) — Linux manual page.* https://man7.org/linux/man-pages/man3/usleep.3.html Last accessed: 2021-11-25.

[15] Elie F. Kfoury, Jorge Crichigno, and Elias Bou-Harb. 2021. An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends. *CoRR* (2021). arXiv:2102.00643 https://arxiv.org/abs/2102.00643

[16] David Kroenke and David Auer. 2016. *Database Processing: Fundamentals, Design, and Implementation.* Pearson Education, Limited.

[17] Benjamin Lewis, Matthew Broadbent, and Nicholas Race. 2019. P4ID: P4 Enhanced Intrusion Detection. In *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 1–4.

[18] Manuel Simon. 2021. manuel-simon/t4p4s - GitHub Repository. https://github.com/manuel-simon/t4p4s/tree/paper Last accessed: 2021-12-02.

[19] Daniele Moro, Davide Sanvito, and Antonio Capone. 2020. 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)*. 95–99.

[20] P4 Language Consortium. 2021. P4 Portable NIC Architecture (PNA). https://p4.org/p4-spec/docs/PNA.html Last accessed: 2021-10-03.

[21] Pensando. 2021. Pensando DSC-25 Distributed Services Cardi—Product Brief. https://pensando.io/wp-content/uploads/2020/03/Pensando-DSC-25-Product-Brief.pdf Last accessed: 2021-10-11.

[22] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, et al. 2019. Flowblaze: Stateful packet processing in hardware. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 531–548.

[23] Prem Jain. 2021. The Value of P4 Programmability at the Network Edge. https://opennetworking.org/wp-content/uploads/2021/05/2021-P4-WS-Prem-Jain-Slides.pdf Last accessed: 2021-10-11.

[24] Dominik Scholz, Sebastian Gallenmüller, Henning Stubbe, and Georg Carle. 2020. SYN Flood Defense in Programmable Data Planes. In *EuroP4@CoNEXT 2020: Proceedings of the 3rd P4 Workshop in Europe, Barcelona, Spain, December 1, 2020.* ACM, 13–20.

[25] Dominik Scholz, Henning Stubbe, Sebastian Gallenmüller, and Georg Carle. 2020. Key Properties of Programmable Data Plane Targets. In *32nd International Teletraffic Congress, ITC 2020, Osaka, Japan, September 22-24, 2020*, Yuming Jiang, Hideyuki Shimonishi, and Kenji Leibnitz (Eds.). IEEE, 114–122.

[26] Alexandru Seibulescu and Mario Baldi. 2020. Leveraging P4 Flexibility to Expose Target-specific Features. In *EuroP4@CoNEXT 2020: Proceedings of the 3rd P4 Workshop in Europe, Barcelona, Spain, December 1, 2020.* ACM, 36–42.

[27] Yan Solihin. 2015. *Fundamentals of parallel multicore architecture.* CRC Press.

[28] The P4.org API Working Group. 2021. P4Runtime Specification. https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html/ Last accessed: 2021-10-21.

[29] Péter Vörös, Dániel Horpácsi, Róbert Kitlei, Dániel Leskó, Máté Tejfel, and Sandor Laki. 2018. T4P4S: A Target-independent Compiler for Protocol-independent Packet Processors.

[30] Eder Ollora Zaballa, David Franco, Zifan Zhou, and Michael S. Berger. 2020. P4Knocking: Offloading host-based firewall functionalities to the network. In *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. 7–12.

[31] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. 2020. 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, 76–89.