Chair of Network Architectures and Services
Department of Informatics
Technical University of Munich

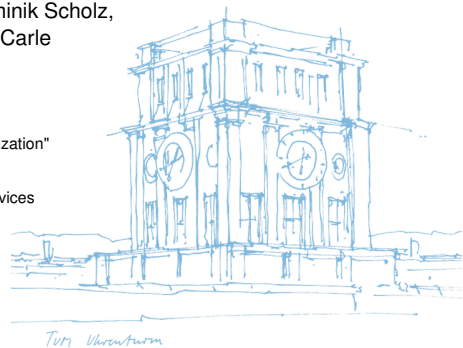
High-Performance Match-Action Table Updates within Programmable Software Data Planes

Manuel Simon, Henning Stubbe, Dominik Scholz,
Sebastian Gallenmüller, Georg Carle

Thursday 7th April, 2022

3. KuVS Fachgespräch "Network Softwarization"

Chair of Network Architectures and Services
Department of Informatics
Technical University of Munich



State Keeping in Data Planes

- State keeping is essential for many applications
- *Registers* (*arrays*) are unstructured memory areas accessible by indices
  - may be fragmented in memory
  - no matching support
  - limited functionality
- In *tables*, structured state can be accessed by sophisticated key matching
- State is often kept by the control plane which decreases performance for state-heavy applications
- We implemented state keeping via *tables* directly in the data plane

P4

- P4 [1] is a domain-specific language for SDN data planes
- In P4, *registers* are changeable within the data plane, *tables* only by the control plane

→ Updatable table entries would increase performance

→ We implemented this for the P4 software target `t4p4s` using a `@__ref` annotation

Introduction

Background

P4

- P4 [1] is a domain-specific language for SDN data planes
- In P4, *registers* are changeable within the data plane, *tables* only by the control plane

→ Updatable table entries would increase performance

→ We implemented this for the P4 software target `t4p4s` using a `@__ref` annotation

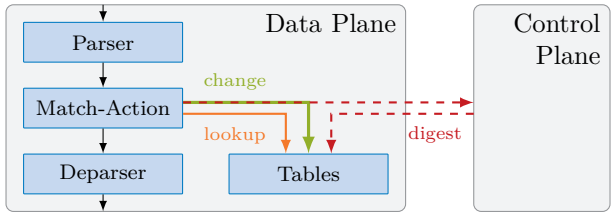
T4P4S

- `t4p4s` [10] is a hardware-indepent transpiler from P4 to C code linked with DPDK developed by ELTE
- The Data Plane Development Kit (DPDK) [2] is an open-source framework enabling fast packet processing in user space
- DPDK performs Receive Side Scaling (RSS) to split traffic among several *lcores*/threads

- The Portable NIC Architecture (PNA) [5] allows adding table entries on lookup misses
- Flexible match-action tables in Pensando SmartNICs [6, 8] allow table update via write-back table fields
  - using target-specific annotations translated to externs
  - no adaption of P4 language/compilers required
- FlowBlaze [7] allows state updates in programmable data planes relying on *registers*

Table Updates

Digest - Current P4 Way



Current State

- For changes in match-action tables, the data plane has to send a digest to the control plane
  - in `t4p4s`: the controller is a separate process, communication via a socket (low round-trip time (RTT))
- Controller requests data plane to update the table

→ Digest-based approach introduces overhead

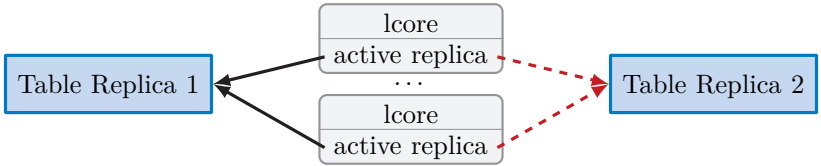
→ Avoid the detour over the controller could improve performance

Investigated Approaches

- **Digest:** introduces a sleep of 1 second
- **Change method:** close to original implementation, but avoids detour
  - uses original timing-based synchronization mechanism
  - sleep time of 200 µs
- **Pointer method:** directly changes entries using their pointers
  - requires alternative synchronization mechanism

Table Updates

Double-Buffering

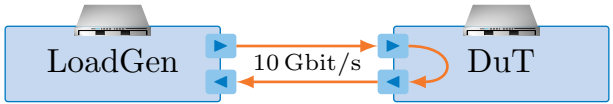


Current State

- Lock-free double-buffering
  - Changes are done to the currently *passive* replica
  - Replicas are swapped
  - Sleep between replica change of 200 µs
  - Changes then promoted to now *passive*
- *Pointer method* not compatible

Evaluation

Topology



Setup

- MoonGen [3] is used to generate traffic
- DuT (`t4p4s`): Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10 GHz, L3-cache size: 15 MiB
- Packets specifies key and new value of updated table entry
- Old value sent back → read and write
- 4 Byte key and value size

Typical cache-based optimizations

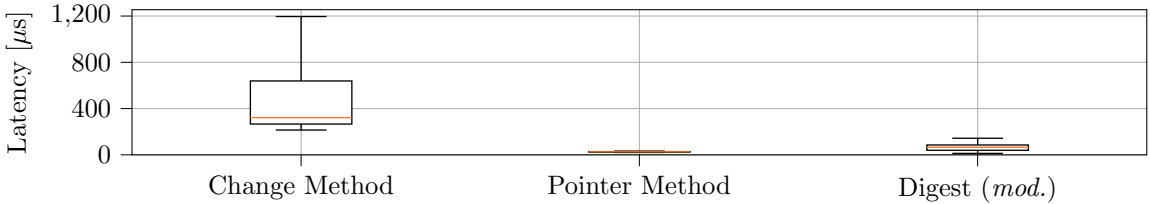
- Load whole cache lines (e.g. 64 B) (spatial locality)
- Heuristic-based prefetching (time locality)

Measure worst-case scenario → maximize cache misses

- Key is pseudo-randomly selected in [0; TABLE_SIZE)
- Large table size exceeding cache size

Evaluation

Table Update Methods, 700 B Packets



Change method

- Bad performace
  - 3.39 kpps
  - 322 µs median latency (high variance)
- uses original synchronization mechanism → wait time of 200 µs

Pointer method

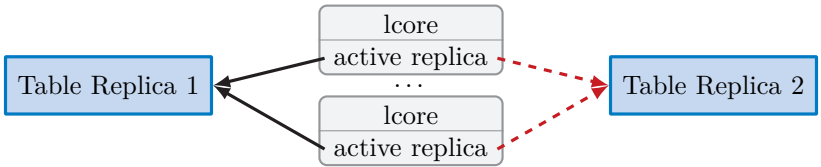
- Good performance
  - 1.73 Mpps (hits linerate)
  - 26.5 µs median latency (almost constant)
- Not compatible with synchronization mechanism

Digest method

- Ignoring sleep
  - 4.1 kpps (else *out of memory*)
  - 65.3 µs median latency (low variance)
- Hardcoded sleep of 1 second would allow < 1 pps

Table Architecture

Overview

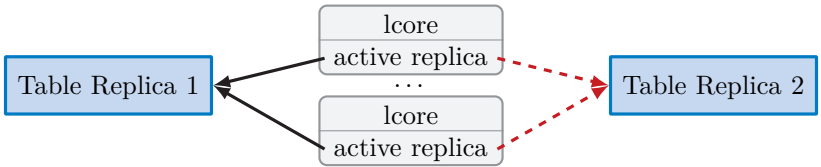


Current State

- Lock-free double-buffering
  - Changes are done to the currently *passive* replica
  - Replicas are swapped
  - Sleep between replica change of 200 µs
  - Changes then promoted to now *passive*
- *Pointer method* not compatible

Table Architecture

Overview



Current State

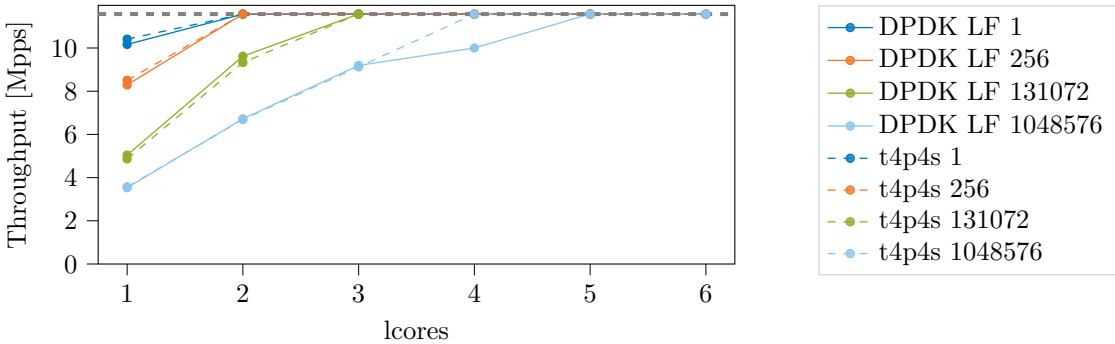
- Lock-free double-buffering
  - Changes are done to the currently *passive* replica
  - Replicas are swapped
  - Sleep between replica change of 200 µs
  - Changes then promoted to now *passive*
- *Pointer method* not compatible

Consistency

- **Insert/Update consistency**
  → one replica of lock-free DPDK hash map
- **Inter-packet race conditions**
  → per-entry locks

Table Architecture

Insert/Update-Consistency, 84 Byte Packets

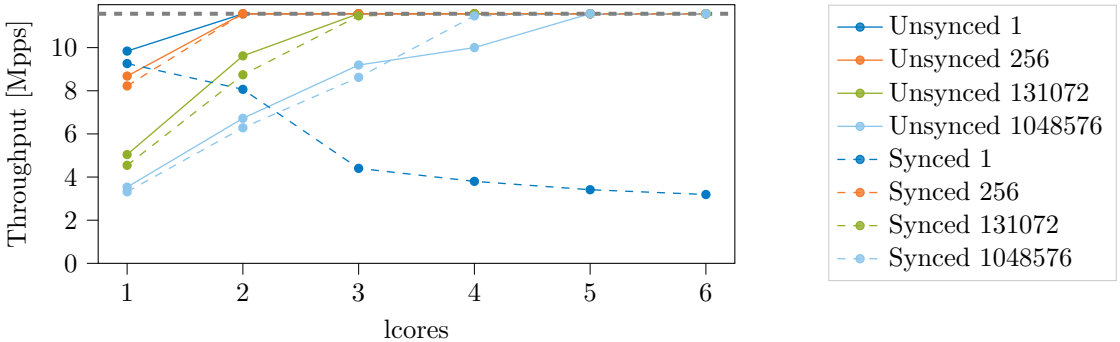


Replace double-buffering mechanism (t4p4s) through lock-free DPDK hash table implementation (DPDK LF)

- DPDK design is also lock-free → nearly same performance
- Only one replica required → allowing *pointer method* to work

Table Architecture

Avoiding Inter-Packet Data Races, 84 Byte Packets



Each entry includes an (optional) lock (Synced)

- Lock is acquired before executing action, and released afterwards
- Locking decreases performance up to 10 %
- Only necessary for global (i.e., flow-independent) entries/state

Conclusion

Contributions

- Implementation of writable table entries in `t4p4s` using `@__ref` annotation
  - → comparable performance to only reading entries
- Synchronization and storage design configurable using `@tableconfig` annotation
- Avoiding inter-packet races using per-entry locks
- Source code available on GitHub [4]

Further contributions not presented

- Cache-efficient storage design
- Cache fitting models

→ read our paper [9]

Bibliography

[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.
Computer Communication Review, 44(3):87–95, 2014.

[2] DPDK.
Data Plane Development Kit, 2021.
Last accessed: 2021-10-03.

[3] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle.
Moongen: A scriptable high-speed packet generator.
In K. Cho, K. Fukuda, V. S. Pai, and N. Spring, editors, *Proceedings of the 2015 ACM Internet Measurement Conference, IMC 2015, Tokyo, Japan, October 28-30, 2015*, pages 275–287. ACM, 2015.

[4] Manuel Simon.
manuel-simon/t4p4s - GitHub Repository, 2021.
Last accessed: 2021-12-02.

[5] P4 Language Consortium.
P4 Portable NIC Architecture (PNA), 2021.
Last accessed: 2021-10-03.

[6] Pensando.
Pensando DSC-25 Distributed Services Cardi—Product Brief, 2021.
Last accessed: 2021-10-11.

[7] 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)*, pages 531–548, 2019.

Bibliography

[8] Prem Jain.
The Value of P4 Programmability at the Network Edge, 2021.
Last accessed: 2021-10-11.

[9] 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*, pages 102–108. ACM, 2021.

[10] P. Vörös, D. Horpácsi, R. Kitlei, D. Leskó, M. Tejfel, and S. Laki.
T4P4S: A Target-independent Compiler for Protocol-independent Packet Processors.
In *IEEE 19th International Conference on High Performance Switching and Routing, HPSR 2018, Bucharest, Romania, June 18-20, 2018*, pages 1–8. IEEE, 2018.

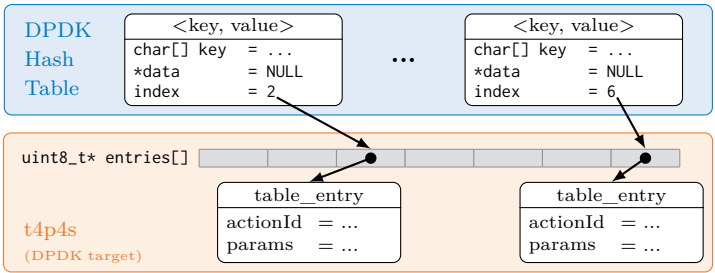
Additional slides

Storage Design

Original Storage Design

- So far, we only considered fast table updates and consistency
- Performance can be further improved by a cache-efficient storage design

→ Ensure spatial locality

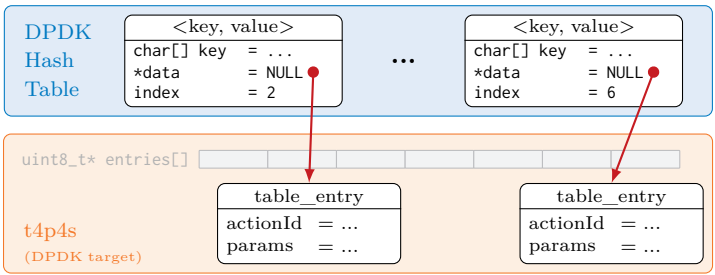


Problems

- Double indirection
- Memory lost due to alignment to 64 Byte
- Entries lay fragmented in memory

Storage Design

Dynamic Storage Design



Advantages

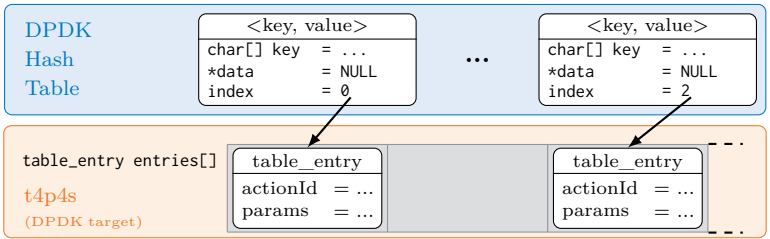
- Only one indirection
- Dynamic allocation of required memory for entries
- Flexible table size

Problems

- Memory lost due to alignment to 64 Byte
- Entries lay fragmented in memory

Storage Design

Static Storage Design



Advantages

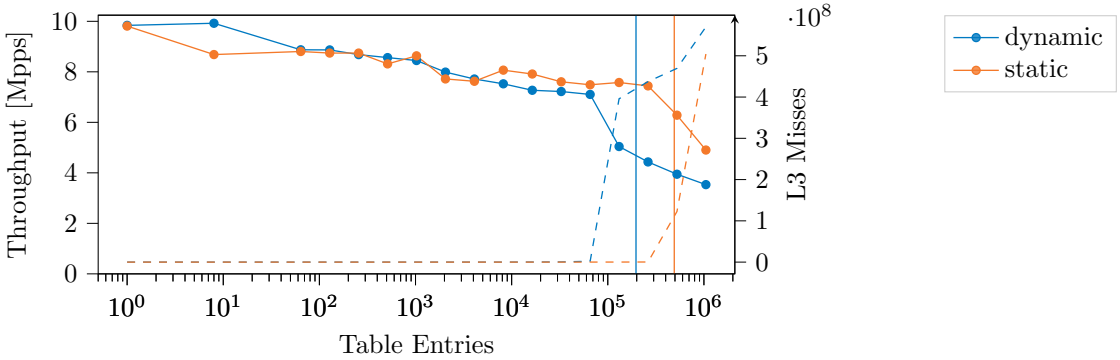
- Only one indirection
- Enforcing spatial locality
- Aligned to 16 Byte
- Better cache utilization

Problems

- Fixed table size
- Lost memory for low table fill rates

Storage Design

Throughput and Cache Misses for Static and Dynamic Storage, 84 Byte Packets



- Static design achieves more throughput, especially for large table sizes
- Performance gain up to 40 %