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

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

Monday 13th December, 2021
4th European P4 Workshop (EuroP4 ’21)

Chair of Network Architectures and Services
Department of Informatics
Technical University of Munich
Introduction
State Keeping in Data Planes

- State keeping is essential for many applications
- *Registers* are the native way for data plane state in P4
  - may be fragmented in memory
  - no matching support
  - limited functionality
- *Tables* are only changeable by the control plane

→ Updatable table entries would increase performance
→ We implemented this for the P4 software target *t4p4s*
  using a `@__ref` annotation
Introduction
State Keeping in Data Planes

- State keeping is essential for many applications
- *Registers* are the native way for data plane state in P4
  - may be fragmented in memory
  - no matching support
  - limited functionality
- *Tables* are only changeable 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 [8] is a hardware-independent transpiler from P4 to C code linked with DPDK developed by ELTE
- The Data Plane Development Kit (DPDK) [1] 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

Related Work

- The Portable NIC Architecture (PNA) [4] allows adding table entries on lookup misses
- Flexible match-action tables in Pensando SmartNICs [5, 7] allow table update via write-back table fields
  - using target-specific annotations translated to externs
  - no adaption of P4 language/compilers required
- FlowBlaze [6] allows state updates in programmable data planes relying on registers
Introduction

State Keeping in Data Planes

- State keeping is essential for many applications
- *Registers* are the native way for data plane state in P4
  - may be fragmented in memory
  - no matching support
  - limited functionality
- *Tables* are only changeable by the control plane
  → Updatable table entries would increase performance
  → We implemented this for the P4 software target `t4p4s`
    using a `@__ref` annotation

Related Work

- The Portable NIC Architecture (PNA) [4] allows adding table entries on lookup misses
- Flexible match-action tables in Pensando SmartNICs [5, 7] allow table update via write-back table fields
  - using target-specific annotations translated to externs
  - no adaption of P4 language/compilers required
- FlowBlaze [6] allows state updates in programmable data planes relying on *registers*

T4P4S

- `t4p4s` [8] is a hardware-independent transpiler from P4 to C code linked with DPDK developed by ELTE
- The Data Plane Development Kit (DPDK) [1] 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
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 [2] 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

<table>
<thead>
<tr>
<th>Change Method</th>
<th>Pointer Method</th>
<th>Digest (mod.)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Latency [µs]</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1,200</td>
<td>0</td>
</tr>
<tr>
<td>400</td>
<td>800</td>
<td>400</td>
</tr>
<tr>
<td>1,200</td>
<td></td>
<td>800</td>
</tr>
</tbody>
</table>

- **Change method**
  - Bad performance
    - 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)

- **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
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 [3]

Further contributions not presented

• Cache-efficient storage design
• Cache fitting models → read our paper
Bibliography


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

DPDK Hash Table

```
uint8_t* entries[]
<key, value>
char[] key = ...
*data = NULL
index = 2
<key, value>
char[] key = ...
*data = NULL
index = 6
```

t4p4s (DPDK target)
Storage Design

Static Storage Design

dpdk hash table

<table>
<thead>
<tr>
<th>table_entry entries[]</th>
<th>table_entry</th>
<th>table_entry</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>actionId = ...</td>
<td>actionId = ...</td>
</tr>
<tr>
<td></td>
<td>params = ...</td>
<td>params = ...</td>
</tr>
</tbody>
</table>

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

M. Simon et al. — High-Performance Match-Action Table Updates within Programmable Software Data Planes
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%