CS244
Advanced Topics in Networking

Lecture 8: Programmable Forwarding
Sundararajan Renganathan

“Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN”
[Pat Bosshart et al. 2013]
“NetCache: Balancing Key-Value Stores with Fast In-Network Caching”
[X. Jin, et al. 2017]
Context

Pat Bosshart
At the time: TI (Texas Instruments)
Architect of first LISP CPU and 1GHz DSP

George Varghese
At the time: MSR
Today: Professor at UCLA

+ Others from TI

+ Others from Stanford

At the time the paper was written (2012)…

- Fastest switch ASICs were fixed function, around 1Tb/s
- Lots of interest in “disaggregated” switches for large data-centers
Switch with fixed function pipeline

Fixed Parser

L2 Table
L2 Hdr Actions

IPv4 Table
IP Hdr Actions

IPv6 Table
V6 Hdr Actions

ACL Table
ACL Actions

Fixed Header Processing Pipeline
“Programmable switches run 10x slower, consume more power and cost more.”

Conventional wisdom in 2010
Packet Forwarding Speeds

- 0.1 Gb/s
- 1 Gb/s
- 10 Gb/s
- 100 Gb/s
- 1 Tb/s
- 10 Tb/s
- 6.4 Tb/s

Year:
- 1990
- 1995
- 2000
- 2005
- 2010
- 2015
- 2020

Switch Chip (Gb/s per chip)
Domain Specific Processors

Computers
- Java Compiler
- OpenCL Compiler
- Signal Processing Compiler
- Machine Learning Compiler
- Networking Compiler

Graphics
- CPU
- GPU
- DSP
- TPU

Signal Processing

Machine Learning

Networking

Language

Compiler
Domain Specific Processors

Computers
- Java Compiler
- OpenCL Compiler
- Matlab Compiler
- TensorFlow Compiler
- Networking
  - P4 Compiler
  - PISA
    - aka “RMT”
Network systems tend to be designed “bottom-up”

“This is how I process packets”
What if they could be programmed “top-down”?

“This is precisely how you must process packets”
The RMT design [2013]
Figure 3: Switch chip architecture.
Figure 1: RMT model architecture.

(a) RMT model as a sequence of logical Match-Action stages.

(b) Flexible match table configuration.

c) VLIW action architecture.
Programmer declares which headers are recognized

Programmer declares what tables are needed and how packets are processed

All stages are identical. A “compiler target”.
PISA: Protocol Independent Switch Architecture
PISA: Protocol Independent Switch Architecture
PISA: Protocol Independent Switch Architecture

- Programmable Parser
- Ethernet MAC Address Table
- MPLS Tag Table
- IPv4 Address Table
- IPv6 Address Table
- VXLAN
- ACL Rules
P4 program example: Parsing Headers

```p4
header_type ethernet_t {
    fields {
        dstAddr: 48;
        srcAddr: 48;
        etherType: 16;
    }
}

header_type my_encap_t {
    fields {
        foo: 12;
        bar: 8;
        baz: 4;
        qux: 4;
        next_protocol: 4;
    }
}

parser parse_ethernet {
    extract(ethernet);
    \return select(latest.etherType) {
        0x8100: parse_my_encap;
        0x800: parse_ipv4;
        0x86DD: parse_ipv6;
    }
}
```
P4 program example

```
table ipv4_lpm
{
  reads {
    ipv4.dstAddr : lpm;
  }
  actions {
    set_next_hop;
    drop;
  }
}

action set_next_hop(nhop_ipv4_addr, port)
{
  modify_field(metadata.nhop_ipv4_addr, nhop_ipv4_addr);
  modify_field(standard_metadata.egress_port, port);
  add_to_field(ipv4.ttl, -1);
}

control ingress
{
  apply(l2);
  apply(my_encap);
  if (valid(ipv4) {
    apply(ipv4_lpm);
  } else {
    apply(ipv6_lpm);
  }
  apply(acl);
}
```
How programmability is used

1 Reducing complexity
IPv4 and IPv6 routing
- Unicast Routing
  - Routed Ports & SVI
  - VRF
- Unicast RPF
- Strict and Loose
  - Multicast
    - PIM-SM/DM & PIM-Bidir

Tunneling
- IPv4 and IPv6 Routing & Switching
  - IP in IP (6in4, 4in4)
- VXLAN, NVGRE, GENEVE & GRE
- Segment Routing, ILA

MPLS
- LER and LSR
- IPv4/6 routing (L3VPN)
- L2 switching (EoMPLS, VPLS)
- MPLS over UDP/GRE

Ethernet switching
- VLAN Flooding
- MAC Learning & Aging
- STP state
- VLAN Translation

ACL
- MAC ACL, IPv4/v6 ACL, RACL
- GoS ACL, System ACL, PBR
- Port Range lookups in ACLs

QoS
- GoS Classification & marking
- Drop profiles/WRED
- RoCE v2 & FCoE
- CoPP (Control plane policing)

Load balancing
- LAG
- ECMP & WCMP
- Resilient Hashing
- Flowlet Switching

Fast Failover
- LAG & ECMP

NAT and L4 Load Balancing
Security Features
- Storm Control, IP Source Guard

Monitoring & Telemetry
- Ingress Mirroring and Egress Mirroring
  - Negative Mirroring
  - Sflow
  - INT

Counters
- Route Table Entry Counters
- VLAN/Bridge Domain Counters
- Port/Interface Counters

Protocol Offload
- BFD, OAM

Multi-chip Fabric Support
- Forwarding, QoS
Reducing complexity

Compiler

My switch.p4

Switch OS

Driver

Programmable Switch
How programmability is used

2. Adding new features
Protocol complexity 20 years ago

- Ethernet
- IPv4
- IPX

Diagram:
- Ethernet
  - ethtype
  - IPv4
  - ethtype
  - IPX
Datacenter switch today

switch.p4
Example new features

1. New encapsulations and tunnels
2. New ways to tag packets for special treatment
3. New approaches to routing: e.g. source routing in DCs
4. New approaches to congestion control
5. New ways to process packets: e.g. ticker-symbols
Example new features

1. Layer-4 Load Balancer\(^1\)
   - Replace 100 servers or 10 dedicated boxes with one programmable switch
   - Track and maintain mapping for 5-10 million http flows

2. Fast stateless firewall
   - Add/delete and track 100s of thousands of new connections per second

3. Cache for Key-value store\(^2\)
   - Memcache in-network cache for 100 servers
   - 1-2 billion operations per second

---

How programmability is used

Network telemetry
1. "Which path did my packet take?"

   "I visited Switch 1 @780ns, Switch 9 @1.3µs, Switch 12 @2.4µs"

2. "Which rules did my packet follow?"

   "In Switch 1, I followed rules 75 and 250. In Switch 9, I followed rules 3 and 80."

<table>
<thead>
<tr>
<th>#</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>192.168.0/24</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
3. "How long did my packet queue at each switch?"

   "Delay: 100ns, 200ns, 19740ns"

4. "Who did my packet share the queue with?"
3. “How long did my packet queue at each switch?”

“Delay: 100ns, 200ns, 19740ns”

4. “Who did my packet share the queue with?”

Aggressor flow!
These seem like pretty important questions

1. “Which path did my packet take?”
2. “Which rules did my packet follow?”
3. “How long did it queue at each switch?”
4. “Who did it share the queues with?”

A programmable device can potentially answer all four questions. At line rate.
INT: In-band Network Telemetry

Add: SwitchID, Arrival Time, Queue Delay, Matched Rules, …
Example using INT

Anomaly Records

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Switch Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>July 25, 2017 - 18:17:51.513 UTC</td>
<td></td>
</tr>
</tbody>
</table>

Queue Occupancy Over Time (bytes)

[Graph showing queue occupancy over time]

17 Affected Flows

<table>
<thead>
<tr>
<th>Flow</th>
<th>kB In Queue</th>
<th>% of Queue Buildup</th>
<th>Packet Drops</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.32.2.2:46380 -&gt; 10.36.1.2:5101 TCP</td>
<td>3282</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>10.32.2.2:46374 -&gt; 10.36.1.2:5101 TCP</td>
<td>3073.5</td>
<td>27</td>
<td>25</td>
</tr>
<tr>
<td>10.32.2.2:46386 -&gt; 10.36.1.2:5101 TCP</td>
<td>2092.5</td>
<td>18</td>
<td>27</td>
</tr>
<tr>
<td>10.32.2.2:46388 -&gt; 10.36.1.2:5101 TCP</td>
<td>1456.5</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>10.32.2.2:46390 -&gt; 10.36.1.2:5101 TCP</td>
<td>1227</td>
<td>11</td>
<td>36</td>
</tr>
<tr>
<td>10.32.2.2:46372 -&gt; 10.36.1.2:5101 TCP</td>
<td>45</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10.32.2.2:46392 -&gt; 10.36.1.2:5101 TCP</td>
<td>37.5</td>
<td>0</td>
<td>39</td>
</tr>
<tr>
<td>10.35.1.2:34256 -&gt; 10.36.1.2:5102 TCP</td>
<td>34.5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Today’s programmable switching throughputs

Tofino™ 3 Intelligent Fabric Processor
NetCache: Balancing Key-Value Stores with Fast In-Network Caching

Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé JK Lee, Nate Foster, Chang Kim, and Ion Stoica
Goal: Fast and Cost-efficient Rack-scale Key-value Storage

- Store, retrieve, manage key-value objects
  - Critical building block for large-scale cloud services
    - Google
    - Amazon
    - Facebook
    - Twitter
    - etc.
  - Need to meet aggressive latency and throughput objectives efficiently

- Target workloads
  - Small objects
  - Read intensive
  - Highly skewed and dynamic key popularity
The problem

*KV servers under a highly-skewed & rapidly-changing workload*

gets and puts

Load  
KV Servers

ToR
The problem

KV servers under a highly-skewed & rapidly-changing workload

gets and puts

ToR

Load → KV Servers

Q: How can you achieve high throughput and bound tail latency?
The problem

It’s very hard to achieve high throughput and low tail latency at the same time.
What if we had a very fast front-end server?

Q: How big and fast the front-end cache should be?
For a front-end cache to be effective …

- **How big should it be?**
  - Keep $O(N\cdot \log N)$ hot keys where $N$ is the number of KV servers
  - Theory proves that such a front-end cache bounds the variance of KV server utilization *irrespective* of the total number of keys

- **How fast should it be?**
  - At least as large as the aggregated throughput of all KV servers ($N\cdot C$)
Why is this relevant now?

Storage layer:
- flash/disk:
  - each: $O(100)$ KQPS
  - total: $O(10)$ MQPS

Cache layer:
- in-memory:
  - each: $O(10)$ MQPS
  - total: $O(1)$ BQPS

Cache needs to provide the aggregate throughput of the storage layer.
Why is this relevant now?

Cache needs to provide the **aggregate** throughput of the storage layer.

**Storage layer**
- flash/disk: each: $O(100)$ KQPS, total: $O(10)$ MQPS

**Cache layer**
- in-memory: each: $O(10)$ MQPS, total: $O(1)$ BQPS

Small on-chip memory?
Only cache $O(N \log N)$ small items

**PISA** (real-time I/O machine)
- $O(1)$ BQPS
PISA: Protocol Independent Switch Architecture

- **Programmable Parser**
  - Parse custom key-value fields in the packet

- **Programmable Match-Action Pipeline**
  - Read and update key-value data
  - Provide query statistics for cache updates
A conventional switch built with PISA

Control plane (CPU)

Data plane (ASIC)

Match + Action

Programmable Parser

Programmable Match-Action Pipeline
A front-end KV cache built with PISA

- **Data plane**
  - Key-value store to serve queries for cached keys
  - Query statistics to enable efficient cache updates

- **Control plane**
  - Insert hot items into the cache and evict less popular items
  - Manage memory allocation for on-chip key-value store
Line-rate query handling in the data plane

Read Query (cache hit)
1. Client sends a read query to the server.
2. Server responds with a hit from the cache.
3. Client updates the cache and statistics.

Read Query (cache miss)
1. Client sends a read query to the server.
2. Server responds with a miss.
3. Client invalidates the cache and updates the server.
4. Client updates the cache and statistics.

Write Query
1. Client sends a write query to the server.
2. Server responds with an invalidate.
3. Client invalidates the cache.
4. Client updates the cache and statistics.

Server
- Hit
- Cache
- Update
- Stats
- Miss
- Invalidate
- Cache
- Stats
Key-value caching in network ASIC at line rate

- How to identify application-level packet fields?
- How to store and serve variable-length data?
- How to efficiently keep the cache up-to-date?
Packet format

- Application-layer protocol; compatible with existing L2-L4 layers
- Only the front-end cache needs to parse NetCache fields
Key-value caching in network ASIC at line rate

- How to identify application-level packet fields?
- How to store and serve variable-length data?
- How to efficiently keep the cache up-to-date?
Key-value store using register array in network ASIC

```python
action process_array(idx):
    if pkt.op == read:
        pkt.value ← array[idx]
    elif pkt.op == cache_update:
        array[idx] ← pkt.value
```

Register Array

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
</table>

- Key-value store using register array in network ASIC
Key-value store using register array in network ASIC

<table>
<thead>
<tr>
<th>Match</th>
<th>pkt.key == A</th>
<th>pkt.key == B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>process_array(0)</td>
<td>process_array(1)</td>
</tr>
</tbody>
</table>

 action process_array(idx):
  if pkt.op == read:
    pkt.value ← array[idx]
  elif pkt.op == cache_update:
    array[idx] ← pkt.value
Variable-length key-value store in network ASIC?

<table>
<thead>
<tr>
<th>Match</th>
<th>pkt.key == A</th>
<th>pkt.key == B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>process_array(0)</td>
<td>process_array(1)</td>
</tr>
</tbody>
</table>

pkt.value:  

```
A   B
```

0 1 2 3

Register Array

**Key Challenges:**

- No loop or string due to strict timing requirements
- Need to minimize hardware resources consumption
  - Number of table entries
  - Size of action data for each entry in table
  - Size of intermediate metadata across tables
Combine outputs from multiple arrays

**Lookup Table**

<table>
<thead>
<tr>
<th>Match</th>
<th>pkt.key == A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>bitmap = 111</td>
</tr>
<tr>
<td></td>
<td>index = 0</td>
</tr>
</tbody>
</table>

**pkt.value:** A0 A1 A2

**Bitmap** indicates arrays that store the key’s value

**Index** indicates slots in the arrays to get the value

**Minimal hardware resource overhead**

**Value Table 0**

<table>
<thead>
<tr>
<th>Match</th>
<th>bitmap[0] == 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>process_array_0 (index )</td>
</tr>
</tbody>
</table>

→ 0 1 2 3

Register Array 0

A0

**Value Table 1**

<table>
<thead>
<tr>
<th>Match</th>
<th>bitmap[1] == 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>process_array_1 (index )</td>
</tr>
</tbody>
</table>

→ 0 1 2 3

Register Array 1

A1

**Value Table 2**

<table>
<thead>
<tr>
<th>Match</th>
<th>bitmap[2] == 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>process_array_2 (index )</td>
</tr>
</tbody>
</table>

→ 0 1 2 3

Register Array 2

A2
**Combine outputs from multiple arrays**

**Lookup Table**

- **Match**: pkt.key == A, pkt.key == B
- **Action**: bitmap = 111, bitmap = 110
  - index = 0
  - index = 1

- **pkt.value**: A0, A1, A2, B0, B1

**Value Table 0**

- **Match**: bitmap[0] == 1
- **Action**: process_array_0 (index)

- **Value Table 1**

- **Match**: bitmap[1] == 1
- **Action**: process_array_1 (index)

- **Value Table 2**

- **Match**: bitmap[2] == 1
- **Action**: process_array_2 (index)

- **Register Arrays**
  - Register Array 0
  - Register Array 1
  - Register Array 2
Combine outputs from multiple arrays

**Lookup Table**

<table>
<thead>
<tr>
<th>Match</th>
<th>pkt.key == A</th>
<th>pkt.key == B</th>
<th>pkt.key == C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>bitmap = 111</td>
<td>bitmap = 110</td>
<td>bitmap = 010</td>
</tr>
<tr>
<td></td>
<td>index = 0</td>
<td>index = 1</td>
<td>index = 2</td>
</tr>
</tbody>
</table>

**pkt.value:**

A0 | A1 | A2 | B0 | B1 | C0

**Value Table 0**

<table>
<thead>
<tr>
<th>Match</th>
<th>bitmap[0] == 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>process_array_0(index)</td>
</tr>
</tbody>
</table>

Register Array 0

A0 | B0

**Value Table 1**

<table>
<thead>
<tr>
<th>Match</th>
<th>bitmap[1] == 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>process_array_1(index)</td>
</tr>
</tbody>
</table>

Register Array 1

A1 | B1 | C0

**Value Table 2**

<table>
<thead>
<tr>
<th>Match</th>
<th>bitmap[2] == 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>process_array_2(index)</td>
</tr>
</tbody>
</table>

Register Array 2

A2

Combine outputs from multiple arrays:

- pkt.value: A0 A1 A2 B0 B1 C0
**Combine outputs from multiple arrays**

<table>
<thead>
<tr>
<th>Lookup Table</th>
<th>pkt.key == A</th>
<th>pkt.key == B</th>
<th>pkt.key == C</th>
<th>pkt.key == D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>bitmap = 111</td>
<td>bitmap = 110</td>
<td>bitmap = 010</td>
<td>bitmap = 101</td>
</tr>
<tr>
<td>index</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

**pkt.value:**

- A0
- A1
- A2
- B0
- B1
- C0
- C0
- D0
- D1

<table>
<thead>
<tr>
<th>Value Table 0</th>
<th>Match</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitmap[0] == 1</td>
<td></td>
<td>process_array_0 (index )</td>
</tr>
</tbody>
</table>

**Value Table 0**

<table>
<thead>
<tr>
<th>Value Table 1</th>
<th>Match</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitmap[1] == 1</td>
<td></td>
<td>process_array_1 (index )</td>
</tr>
</tbody>
</table>

**Value Table 1**

<table>
<thead>
<tr>
<th>Value Table 2</th>
<th>Match</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitmap[2] == 1</td>
<td></td>
<td>process_array_2 (index )</td>
</tr>
</tbody>
</table>

**Value Table 2**

**Register Array 0:**

<table>
<thead>
<tr>
<th></th>
<th>A0</th>
<th>B0</th>
<th>D0</th>
</tr>
</thead>
</table>

**Register Array 1:**

<table>
<thead>
<tr>
<th></th>
<th>A1</th>
<th>B1</th>
<th>C0</th>
</tr>
</thead>
</table>

**Register Array 2:**

<table>
<thead>
<tr>
<th></th>
<th>A2</th>
<th>D1</th>
</tr>
</thead>
</table>
Key-value caching in network ASIC at line rate

- How to identify application-level packet fields?
- How to store and serve variable-length data?
- How to efficiently keep the cache up-to-date?
Cache insertion and eviction

- Challenge: Keeping the hottest $O(N \log N)$ items in the cache
- Goal: React quickly and effectively to workload changes with **minimal updates**

Diagram:

1. Data plane reports hot keys
2. Control plane compares loads of new hot and sampled cached keys
3. Control plane fetches values for keys to be inserted to the cache
4. Control plane inserts and evicts keys
Query statistics in the data plane

- **Cached key**: per-key counter array
- **Uncached key**
  - Count-Min sketch: report new hot keys
  - Bloom filter: remove duplicated hot key reports
The “boring life” of a NetCache system

Single switch benchmark

![Graphs showing throughput vs. value size and cache size][1]

---

1. Throughput vs. value size.
2. Throughput vs. cache size.

---

Not a typo! 😊 It is indeed BQPS.
And it’s “not so boring” benefits

1 switch + 128 storage servers

3-10x throughput improvements
End.