Dataplane Specialization for High-performance OpenFlow Software Switching

László Molnár, Gergely Pongrácz, Gábor Enyedi, Zoltán Lajos Kis Levente Csikor, Ferenc Juhász, Attila Kőrösi, Gábor Rétvári

TrafficLab, Ericsson Research, Hungary
Department of Telecommunications and Media Informatics, BME
MTA-BME Information Systems Research Group

SIGCOMM’16, August 22-26, 2016, Florianopolis, Brazil
“OpenFlow is expressive but troublesome to make fast on x86.”


Dataplane specialization may help to alleviate the “expressibility vs. performance” tension.
Expressibility vs. Performance

How to support diverse workloads in a single device efficiently?
Datapath Programmability Is Hard

• **Packet forwarding**: map received packets to the action(s) to be executed on them (and execute these)

  \[
  \text{fast-path packet classifier} \\
  \text{packet} \rightarrow \text{header tuple} \rightarrow \text{flow entry} \rightarrow \text{action(s)}
  \]

• Supporting OpenFlow’s expressibility makes the fast-path packet classifier rather complex

• But software-based packet classification is slow

OpenFlow softswitch architectures are all about working around the complexity of fast-path packet classification
### Simple Load Balancer + ACL

#### Table: ACL Rules

<table>
<thead>
<tr>
<th>priority</th>
<th>in_port</th>
<th>ip_src</th>
<th>ip_dst</th>
<th>tcp_src</th>
<th>tcp_dst</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>external</td>
<td>0.0.0.0/1</td>
<td>192.0.2.1</td>
<td>*</td>
<td>80, 443</td>
<td>push_vlan=vlan1, output:server1</td>
</tr>
<tr>
<td>10</td>
<td>external</td>
<td>128.0.0.0/1</td>
<td>192.0.2.1</td>
<td>*</td>
<td>80, 443</td>
<td>push_vlan=vlan1, output:server2</td>
</tr>
<tr>
<td>10</td>
<td>external</td>
<td>0.0.0.0/1</td>
<td>192.0.2.1</td>
<td>*</td>
<td>80, 443</td>
<td>push_vlan=vlan2, output:server3</td>
</tr>
<tr>
<td>10</td>
<td>external</td>
<td>128.0.0.0/1</td>
<td>192.0.2.2</td>
<td>*</td>
<td>80, 443</td>
<td>push_vlan=vlan2, output:server4</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>external</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td>goto_table:26</td>
</tr>
<tr>
<td>10</td>
<td>internal</td>
<td>*</td>
<td>*</td>
<td></td>
<td>80, 443</td>
<td>output:external</td>
</tr>
<tr>
<td>1</td>
<td>internal</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>drop</td>
</tr>
</tbody>
</table>

- **Strict flow priorities**
- **Heavy cross-layer matching**
- **Longest Prefix Match**
- **Port sets/intervals**
- **Wildcard matches**
- **Multi-stage pipelines**
- **Complex actions**
Generic Switch Architectures

- Universal dataplane that supports all use cases “well” (CPqD, xDPd, LINC, OVS, 6WINDGate)
- Tackle difficulty of packet classification by avoiding it
  - do the classification for flows’ first packets
  - use result for subsequent packets: flow caching
- But flow caching introduces its own share of problems
  - breaks on widely changing traffic/header fields: hidden assumptions and performance artifacts
    - [PAM 2009], [HotSDN 2013], [CCR 2014], [EANTC 2015]
  - cache management hard: complex architecture
    - [NSDI 2015]
  - breaks tenant isolation: DOS attacks on caches
    - [NSDI 2014], [CCR 2014]
Our Idea: Dataplane Specialization

- Generic switch architectures over-generalize: optimize for the lowest common denominator
- Instead, let the switch **automagically optimize its dataplane for the given workload**
  - into an Ethernet softswitch for L2 use cases
  - an LPM engine for IP
  - an optimal combination for mixed workloads
- This allows to **choose the best fast-path classifier for each flow table** in the pipeline separately
- Very efficient for simple pipelines, achieve what’s possible for complex ones
- A new dataplane compiler to transform OpenFlow programs into custom fast-paths

  OpenFlow pipeline \(\xrightarrow{\text{ESWITCH}}\) custom fast-path

- Rebuild the datapath for each add-flow/del-flow: compilation speed is crucial

- ESWITCH invokes template-based code generation
  - deconstruct the pipeline into simple packet processing primitives
  - represent primitives with precompiled codelets
  - link templates into executable machine code
ESWITCH: Templates

- Unit of pkt processing behavior that admits a simple and composable machine code implementation
- **Parser template**: raw packets $\rightarrow$ matchable tuples
- Separate parser for each protocol in OpenFlow spec

```assembly
PROTOCOL_PARSER: <set protocol bitmask in r15>
L2_PARSER: mov r12, <pointer to L2 header>
L3_PARSER: mov r13, <pointer to L3 header>
L4_PARSER: mov r14, <pointer to L4 header>
```

- **Matcher template**: match on some header field
- E.g., a matcher for entry ip_dst = ADDR/MASK:

```assembly
macro IP_DST_ADDR_MATCHER(ADDR, MASK):
  mov eax, [r13+0x10] ; IP dst address in eax
  xor eax, ADDR       ; match ADDR
  and eax, MASK       ; apply MASK
  jne ADDR_NEXT_FLOW  ; no match: next entry
```
ESWITCH: Templates

- **Flow table template**: basic classification types

<table>
<thead>
<tr>
<th>Name</th>
<th>Prerequisite</th>
<th>Match type</th>
<th>Implementation</th>
<th>Application</th>
<th>Fallback</th>
</tr>
</thead>
<tbody>
<tr>
<td>direct code</td>
<td>#flows ≤ 4</td>
<td>arbitrary</td>
<td>machine code</td>
<td>universal</td>
<td>compound hash</td>
</tr>
<tr>
<td>compound hash</td>
<td>global mask</td>
<td>exact match</td>
<td>perfect hash</td>
<td>MAC switching &amp; port filtering</td>
<td>LPM</td>
</tr>
<tr>
<td>LPM</td>
<td>prefix masks</td>
<td>longest prefix match</td>
<td>DPDK LPM lib</td>
<td>IP forwarding</td>
<td>linked list</td>
</tr>
<tr>
<td>linked list</td>
<td>none</td>
<td>tuple space search</td>
<td>machine code</td>
<td>complex pipelines</td>
<td>none</td>
</tr>
</tbody>
</table>

- Start with best template, fallback if prerequisite fails

- **Action template**: packet processing functionality

- Separate for each action type, shared across flows
Directly Compiled Datapath

- An OpenFlow pipeline with the below flow entry

```
... priority=i, ip_dst=ADDR/MASK, action=ACTION ...
```

- ES\textsc{witch} compiles it into a sequence of templates

```
PROTOCOL\_PARSER: <set protocol bitmask in r15>
L2\_PARSER: mov r12, <pointer to L2 header>
L3\_PARSER: mov r13, <pointer to L3 header>
...
FLOW_i:
  bt r15d, IP  ; flow entry starts
  jae ADDR\_NEXT\_FLOW  ; packet contains IP header?
  IP\_DST\_MATCHER(ADDR,MASK)  ; ip\_dst=ADDR/MASK?
  jmp ACTION  ; jump to ACTION
FLOW_{(i+1)}:
...
ACTION: ...  ; execute ACTION
```
Compilation Process

- **ESWITCH** divides code generation into 3 stages
  1. **Flow table analysis:** divide pipeline into templates
  - **ESWITCH** uses flow table decomposition to promote tables to efficient table templates
  - **Theorem:** optimal table decomposition is NP-hard
  - We use fast greedy heuristics
2. **Template specialization**: patch templates with flow keys, masks, etc.
   - Code contains constants to avoid memory references

3. **Linking**: resolve dangling pointers to direct address
   - `goto_table` pointers go through per-table trampolines
   - Thus updates are transactional and per-flow-table
     - new code built side-by-side with running datapath
     - trampoline updated when ready
     - all `goto_table` pointers thus updated atomically
Implementation/Evaluation

- PoC ESWITCH prototype on top of the Intel DPDK
- Measured against Open vSwitch (OVS): generic dataplane with multi-level flow cache hierarchy
- Mobile access gateway use case (among others)

10 CEs, 20 users per CE, IP routing table: 10K IP prefixes, couple of dozen flow tables

Access Gateway: Custom Dataplane

1. **Compound hash template**
   - Table:0
     | ip_port | action |
     |---------|--------|
     | vlan1   | goto_table:11 |
     | vlan2   | goto_table:12 |
     | ...     | ...     |
     | net1    | goto_table:110 |
     | net2    | goto_table:110 |

2. **Direct code template**
   - Table:0
     | ip_port | action |
     |---------|--------|
     | vlan1   | goto_table:11 |
     | vlan2   | goto_table:12 |
     | ...     | ...     |
     | net1    | goto_table:110 |
     | net2    | goto_table:110 |

3. **LPM template**
   - Table:110
     | ip_dst | action |
     |--------|--------|
     | IP_11  | push_vlan: vlan1, mod_nw_dst: IP_1, output: vlan1 |
     | IP_12  | push_vlan: vlan1, mod_nw_dst: IP_2, output: vlan1 |
     | IP_21  | push_vlan: vlan2, mod_nw_dst: IP_1, output: vlan2 |

4. **Compound hash template**
   - Table:11
     | ip_src | action |
     |--------|--------|
     | IP_1   | write_actions(stripped_vlan, mod_nw_src: IP_11), goto_table:200 |
     | IP_2   | write_actions(stripped_vlan, mod_nw_src: IP_12), goto_table:200 |
     | *      | controller |
Throughput

Throughput performance comparison between ESwitch and OVS with single core, 64-byte packets, Intel Xeon, XL710 @ 40 Gb.
Latency

![Latency Graph](image)

- **ESwitch**
- **OVS**

**Graph Details:**
- **CPU cycles/packet** vs. **number of active flows**
- **Parameters:**
  - Single core
  - 64-byte packets
  - Intel Xeon, XL710 @ 40 Gb
Throughput Under Updates

The graph shows the normed packet rate [pps] as a function of the number of updates per second for ESwitch and OVS. The x-axis represents the number of updates per second, ranging from 0 to 100K, and the y-axis represents the normed packet rate [pps], ranging from 0 to 1.0. The graph includes data points for single core, 64-byte packets, with random updates to IP routing table.
Conclusions

• For a switch to be truly programmable, the dataplane itself must also be adaptable
• **ESWITCH** is a datapath compiler to turn OpenFlow programs into runnable fast-paths
  ○ (at least) twice the packet rate of OVS
  ○ orders of magnitude smaller latency
  ○ even under heavy update load
• Admits analytic performance models (see paper)
• **ESWITCH** is now in production at Ericsson!
Hope you’ve seen the demo! If not, please talk to us, we may find a way to show you ESWITCH in operation.

ESWITCH is about to become open-source (as soon as we resolve IPR issues)!

Besides, we are looking for visiting researcher positions...
ESWITCH vs P4

• Both P4 and ESWITCH are datapath compilers, but ESWITCH is restricted to OpenFlow while P4 is generic
• OTOH, P4 is static (knows pipeline semantics only), while ESWITCH sees the actual pipeline contents
• The allows ESWITCH to use several runtime optimization techniques, similar to JIT compilers:
  ◦ template specialization with full constant inlining
  ◦ direct jump pointers
  ◦ small tables JITted to the direct code template
• Potentially more efficient code with ESWITCH than with equivalent P4 program
• There is no reason why dataplane specialization could not be extended to P4