Compiling Packet Programs to Reconfigurable Switches

Lisa Yan and Lavanya Jose
Stanford University
Switch OS

Run-time API

Driver

Fixed-function ASIC

“Bottom-Up”
“This is how I process packets”

Switch OS

Run-time API

Driver

Programmable Switch

“Top-down”
“This is how the switch must process packets”
Programmable switches

Some switches are more programmable than fixed-function ASICs

- CPU/NPU (OVS, Ezchip, Netronome, etc.)
- FPGA (Xilinx, Altera, Corsa)
- **Flexible Match+Action ASICs**
  (Intel Flexpipe, Cisco Doppler, Xpliant, …)
P4: A high-level language

Header Fields: VLAN

```c
header_type vlan_tag_t {
   fields {
      ...
      vid : 12;
      etherType : 16;
   }
} header vlan_tag_t vlan_tag[NUM];
```

Parser

```c
parser parse_ethernet {
   extract(ethernet);
   return switch(latest.etherType) {
      ETHERTYPE_VLAN : parse_vlan;
      ETHERTYPE_MPLS : parse_mpls;
      ETHERTYPE_IPV4 : parse_ipv4;
      ETHERTYPE_IPV6 : parse_ipv6;
      ...
   }
} parser parse_vlan {
   extract(vlan_tag[next]);
   ...
}
```

Match+Action Table: VLAN

```c
table port_vlan {
   reads {
      std_metadata.ingress_port : exact;
      vlan_tag[OUTER_VLAN].vid : exact;
   }
   actions {
      drop, ing_lif_extract;
   }
   size 16384;
}
```

Control Flow: Ingress

```c
control ingress {
   apply_table(mac_learning);
   if (valid(vlan_tag[0])) {
      apply_table(port_vlan);
   }
   apply_table(routable) {
      ucast_action {
         apply_table(ucast);
      }
      mcast_action {
         ...
      }
      ...
   }
}
### Table

<table>
<thead>
<tr>
<th>Table name</th>
<th>Match width</th>
<th># entries</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC Learning</td>
<td>60 b</td>
<td>4000</td>
<td>exact</td>
</tr>
<tr>
<td>Unicast</td>
<td>32 b</td>
<td>2000</td>
<td>ternary</td>
</tr>
<tr>
<td>Switching</td>
<td>60 b</td>
<td>4000</td>
<td>exact</td>
</tr>
<tr>
<td>ACL</td>
<td>110 b</td>
<td>1000</td>
<td>Ternary</td>
</tr>
</tbody>
</table>

### Diagram

- **MAC Learning**
  - M: SMAC, ...
  - No VLAN?
- **Port VLAN**
  - M: VLAN, ...
- **Routable**
  - M: SMAC, DMAC, VLAN
- **Unicast**
  - M: IPv4 DIP
  - A: SMAC, VLAN, ...
- **Switching**
  - M: DMAC, VLAN, ...
- **Multicast**
  - M: ...
- **IGMP**
  - ...

### Code Snippet

```
# Table
<table>
<thead>
<tr>
<th>Table name</th>
<th>Match width</th>
<th># entries</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC Learning</td>
<td>60 b</td>
<td>4000</td>
<td>exact</td>
</tr>
<tr>
<td>Unicast</td>
<td>32 b</td>
<td>2000</td>
<td>ternary</td>
</tr>
<tr>
<td>Switching</td>
<td>60 b</td>
<td>4000</td>
<td>exact</td>
</tr>
<tr>
<td>ACL</td>
<td>110 b</td>
<td>1000</td>
<td>Ternary</td>
</tr>
</tbody>
</table>
```
Why a compiler?
The Program: A representation

- **Assignment constraint**: all tables should be allocated somewhere in the pipeline

- How do we respect control flow while maximizing concurrency in a switch pipeline?

<table>
<thead>
<tr>
<th>Table name</th>
<th>Match width</th>
<th># entries</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC Learning</td>
<td>60 b</td>
<td>4000</td>
<td>exact</td>
</tr>
<tr>
<td>Unicast</td>
<td>32 b</td>
<td>2000</td>
<td>ternary</td>
</tr>
<tr>
<td>Switching</td>
<td>60b</td>
<td>4000</td>
<td>exact</td>
</tr>
<tr>
<td>ACL</td>
<td>110b</td>
<td>1000</td>
<td>Ternary</td>
</tr>
</tbody>
</table>

- **MAC learning**: reads SMAC
- **Switching**: reads outgoing DMAC, modifies egress port
- **Port VLAN**: drops based on VLAN
- **unicast**: modifies SMAC, DMAC, and VLAN
Representing Control Flow

- **Table Dependency Graph (TDG)**
- Directly generated from P4 program
- **Dependency constraint**: all tables assigned should respect the program control flow
The Switch

- Memory types
- Overhead memory: actions, statistics
- Capacity constraint: assignment of tables should not overflow memory per stage
- Switch-specific: input crossbars, match layout
Summary of the compiler problem

Goal: map a program to a switch’s pipeline by exploiting concurrency

Constraints:
- Assignment – all tables are somewhere
- Dependency – respect program control flow
- Capacity – obey memory limits of switch
- Switch-specific
Approach 1: ILP

- Integer Linear Programming (ILP)
  - Make constraints into inequalities
  - Find the best valid solution to an objective function

- Constraints:
  assignment, capacity, dependency, switch-specific

- Objective functions:
  number of stages used, pipeline latency, power consumed

minimize: $4y$
subject to:
$3x + 2y \geq 6$
$1 \leq y \leq 2$
$x \leq 1$
ILP Example: Assignment

- All entries of a table must be assigned
- Table A: 5000 entries, exact match
- Memory types $m = \text{exact, ternary.} \ldots$
- $W_{s,A,m} = \# \text{ entries from Table A assigned to stage } s, \text{ memory type } m$

$$\sum_{s,m} W_{s,A,m} \geq 5000$$
Approach 1: ILP

(+) global view of the problem
(+) returns best solution when possible

(–) NP-complete
(–) solver runtime is long
Approach 2: Greedy

1. Sort tables according to some heuristic

2. For each table:
   - Put in earliest stage possible
   - If constraints are violated, move to next stage
   - Repeat until all tables assigned
Approach 2: Greedy

(+) fast runtime
(+) control over sorting heuristic

(–) local view of resources
(–) may not find best (or any) solution
Experiments

- 2 different switches: RMT, FlexPipe
- 4 different benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Switch</th>
<th>Tables</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2L3 Complex</td>
<td>RMT</td>
<td>24</td>
<td>33</td>
</tr>
<tr>
<td>L2L3 Simple</td>
<td>RMT</td>
<td>16</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>FlexPipe</td>
<td>13</td>
<td>16</td>
</tr>
<tr>
<td>L2L3 mTag</td>
<td>RMT</td>
<td>19</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>FlexPipe</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>L3 DC</td>
<td>RMT</td>
<td>13</td>
<td>11</td>
</tr>
</tbody>
</table>
Results

- 3 greedy heuristics
- 3 ILP objective functions

Worst median greedy runtime: 0.33 seconds
Worst median ILP runtime: 233.84 seconds

<table>
<thead>
<tr>
<th>L2L3 Complex Benchmark on RMT chip</th>
<th># stages</th>
<th>latency (ns)</th>
<th>power (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best greedy</td>
<td>17</td>
<td>130</td>
<td>4.98</td>
</tr>
<tr>
<td>Optimal (ILP)</td>
<td>16</td>
<td>104</td>
<td>4.44</td>
</tr>
</tbody>
</table>
What did we learn?

Greedy:
- Much faster than ILP
- Choosing a heuristic is important

→ If you *need* the program to fit, use ILP

Greedy and ILP can help each other!
What else did we learn?

- Not all heuristics are created equal
  - ILP validates that dependency-based is best
- Use greedy estimates to speed up ILP
- Design decisions from compiler solutions
  - Some programs push the limits of the switch
  - Use compiler to validate memory dimensions based on expected program input
Thanks!

P4: http://p4.org/
P4:

FlexPipe:

Advisors:
Nick McKeown (Stanford), George Varghese (Microsoft Research)