Programming the Forwarding Plane

Nick McKeown
PISA: Protocol Independent Switch Architecture
[Sigcomm 2013]
PISA: Protocol Independent Switch Architecture
ABSTRACT

P4 is a high-level language for programming protocol-independent packet processors. P4 works in conjunction with SDN control protocols like OpenFlow. In its current form, OpenFlow explicitly specifies protocol headers on which it operates. This set has grown from 12 to 41 fields in a few years, increasing the complexity of the specification while not providing the flexibility to add new headers. In this paper we propose P4 as a strawman proposal for how OpenFlow should evolve in the future. We have three goals: (1) Reconfigurability in the field: Programmers should be able to change the way switches process packets once they are deployed. (2) Protocol independence: Switches should not be tied to any specific network protocols. (3) Target independence: Programmers should be able to describe packet-processing functionality independently of the specifics of the underlying hardware. As an example, we describe how to use P4 to configure a switch to add a new hierarchical label.

1. INTRODUCTION

Software-Defined Networking (SDN) gives operators programmatic control over their networks. In SDN, the control plane is physically separate from the forwarding plane, and one control plane controls multiple forwarding devices. While forwarding devices could be programmed at the same time, OpenFlow, the de-facto standard for controlling OpenFlow-enabled switches, typically consists of a single stage of rule tables that the forwarding plane follows. While forwarding devices could be programmed with multiple rule tables, to allow switches to expose more of their capabilities to the controller. The proliferation of new header fields shows no signs of stopping, for example, data center network operators increasingly want to apply new forms of packet encapsulation (e.g., NVGRE, VXLAN, and STT), for which they require new header fields. Rather than repeatedly extending the OpenFlow specification, we argue that future switches should support flexible mechanisms for parsing packets and matching header fields, allowing controller applications to leverage these capabilities through a common, open interface (i.e., a new “OpenFlow 2.0” API). Such a general, extensible approach would be simpler, more elegant, and more future-proof than today’s OpenFlow 1.x standard.

Recent chip designs demonstrate that such flexibility can be achieved in custom ASICs at terabit speeds [1, 2, 3]. Programming this new generation of switch chips is far from easy. Each chip has its own low-level interface, akin to a microcode programming model. In this paper, we sketch the design of a higher-level language for Programming Protocol-Independent Packet Processors (P4). Figure 1 shows the relationship between P4—used to configure a switch, telling it how packets are to be processed—and existing APIs (such as OpenFlow) that are designed to populate the forwarding tables in fixed function switches. P4 raises the level of abstraction for programming the network, and can serve as a common, open, vendor-agnostic interface.
Update on P4 Language Ecosystem
P4.org – P4 Language Consortium

BOARD MEMBERS
Two Board members oversee the consortium:

Nick McKeown
Stanford University

Jennifer Rexford
Princeton University

Field Reconfigurable
P4 allows network engineers to change the way their switches process packets after they are deployed.
P4.org – P4 Language Consortium

- Regular P4 meetings
- Full-day tutorial at Sigcomm 2015
- 2nd P4 Workshop at Stanford
- 1st P4 Boot camp for PhD students November 19–20
- 1st P4 Developers Day November 19

Open for free to any individual or organization

Field Reconfigurable
P4 allows network engineers to change the way their switches process packets after they are deployed.
Mapping P4 programs to compiler target
Lavanya Jose, Lisa Yan, George Varghese, NM

[NSDI 2015]
Naïve Mapping: Control Flow Graph

Control Flow

Switch Pipeline

Programmable Parser → L2 Table → IPv4 Table → IPv6 Table → ACL Table → Queues
Table Dependency Graph (TDG)
Efficient Mapping: TDG

Switch Pipeline

Control Flow Graph

Programmable Parser → L2 Table → IPv4 Table → IPv6 Table → ACL Table → Queues
Example Use Case: Typical TDG

Configuration for 16-stage PISA
Mapping Techniques
[NSDI 2015]

Compare: Greedy Algorithm versus Integer Linear Programming (ILP)

Greedy Algorithm runs 100-times faster
ILP Algorithm uses 30% fewer stages

Recommendations:
1. If enough time, use ILP
2. Else, run ILP offline to find best parameters for Greedy algorithm

P4 code, switch models and compilers available at: http://github.com/p4lang
Problem: Adding new protocol feature to OVS is complicated

- Requires domain expertise in kernel programming and networking
- Many modules affected
- Long QA and deployment cycle: typically 9 months

Approach: Specify forwarding behavior in P4; compile to modify OVS

Question: How does the PISCES switch performance compare to OVS?
PISCES Architecture

- P4 Program
- P4 Compiler
- C Code
  - Parse
  - Match
  - Action
- OVS Source Code
- Runtime Flow Rules
  - Flow Rule Type Checker
  - Match-Action Rules
- Slow Path Configuration
- OVS Executable
Native OVS expressed in P4

VLAN Ingress Processing
Match: ingress_port
Action: add_vlan
Action: no_op

MAC Learning
Match: eth.src
Action: learn
Action: no_op

Routing
Match: ip.dst
Action: nexthop
Action: drop

Switching
Match: eth.dst
Match: vlan.vid
Action: forward
Action: bcast

ACL
Match: ip.src,ip.dst
Match: ip.prtcl,
port.src,port.dst
Action: no_op
Action: drop

Routable
Match: eth.src
Match: eth.dst
Match: vlan.vid
Action: no_op

VLAN Egress Processing
Match: egress_port
Action: remove_vlan
Action: no_op
**PISCES vs Native OVS**

Throughput (Gbps) vs Packet Size (Bytes)

- PISCES
- PISCES (Optimized)
- OVS
Complexity Comparison

<table>
<thead>
<tr>
<th></th>
<th>LOC</th>
<th>Methods</th>
<th>Method Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native OVS</td>
<td>14,535</td>
<td>106</td>
<td>137.13</td>
</tr>
<tr>
<td>ovs.p4</td>
<td>341</td>
<td>40</td>
<td>8.53</td>
</tr>
</tbody>
</table>

40x reduction in LOC
20x reduction in method size

<table>
<thead>
<tr>
<th></th>
<th>Files Changed</th>
<th>Lines Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection Label</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OVS ovs.p4</td>
<td>28</td>
<td>411</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Tunnel OAM Flag</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OVS ovs.p4</td>
<td>18</td>
<td>170</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>TCP Flags</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OVS ovs.p4</td>
<td>20</td>
<td>370</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Code mastery no longer needed
Next Steps

1. Make PISCES available as open-source (May 2016)
2. Accumulate experience, measure reduction in deployment time
3. Develop P4-to-eBPF compiler for kernel forwarding
PERC: Proactive Explicit Rate Control
Lavanya Jose, Stephen Ibanez, Mohammad Alizadeh, George Varghese, Sachin Katti, NM

**Problem**: Congestion control algorithms in DCs are “reactive”
• Typically takes 100 RTTs to converge to fair-share rates (e.g. TCP, RCP, DCTCP)
• The algorithm it doesn’t know the answer; it uses successive approximation

**Approach**: Explicitly calculate the fair-share rates in the forwarding plane

**Question**: Does it converge much faster? Is it practical?

[Hotnets 2015]
Reactive vs Proactive Algorithms

![Graph showing transmission rate over time for different algorithms and flows.]

- PERC (solid)
- RCP (dashed)
- Ideal (dotted)

Flow 1 (start=0)
Flow 2 (start=600us)
### Performance Results

Convergence time determined by dependency chain

<table>
<thead>
<tr>
<th></th>
<th>RCP</th>
<th>PERC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Median</td>
<td>14 RTTs</td>
<td>4 RTTs</td>
</tr>
<tr>
<td>Tail (99th)</td>
<td>71 RTTs</td>
<td>10 RTTs</td>
</tr>
</tbody>
</table>
Next Steps

Convergence time
• Proof that convergence time equals length of dependency chain
• Reduce measured time to provable minimum

Develop practical algorithm
• Resilient to imperfect and lost update information
• Calculated in PISA-style forwarding plane
<The End>