Programming
The Network Data Plane

CS244, Spring 2017

Changhoon Kim
chang@barefootnetworks.com
For better and easier management
- Allocate h/w resources adaptively
- Ensure visibility for debugging and diagnostics
- Verify network behavior
- Improve OAM capabilities
Tons of beautiful ideas to enhance your network …

- To enable critical new features
  - Introduce custom encapsulations and/or headers
  - Better network-fabric load-balancing
  - Enhanced congestion control
  - Improve robustness
  - Embed some middlebox functions inline
  - any many more …
Extremely limited way of turning beautiful ideas into reality

- No DIY – must work with vendors at **feature** level
- Excruciatingly complicated and involved process to build consensus and pressure for **features**
- Painfully long and unpredictable lead time
- To use new features, you must get new switches
- What you finally get != what you asked for
Ideal world

- We all know how to realize our own ideas by “programming” CPUs
  - Programs used in every phase (implement, test, deploy)
  - Enables extremely fast iteration and differentiation
  - Lets you own your own ideas
  - Builds sustainable ecosystem where all participants benefit

- Let’s replicate this healthy, sustainable ecosystem for networking
Packet forwarding speeds

Gb/s (per chip)


0.1 1 10 100 1000 10000 100000

Switch Chip

6.4Tb/s
Packet forwarding speeds

![Graph showing packet forwarding speeds from 1990 to 2020 with Switch Chip increasing from 0.1 Gb/s to 6.4 Tb/s by 2020, and CPU increasing from 0.1 Gb/s to 80 Gb/s by 2020.](image)
Programmable network devices come to rescue

- CPUs: 10s of Gb/s
- FPGAs, NPUs: 100s of Gb/s
- Protocol-Independent Switch Architecture (PISA) chips: a few Tb/s
  - Initial architecture introduced in RMT [SIGCOMM’13]
  - Packet processing machine with fully programmable parser and generic match-action logic
  - No penalty in size, cost, and power compared to fixed-function ASICs
  - PISA products available now -- in next few years this kind of silicon will dominate
Domain-specific processors

Computers: Java Compiler
Graphics: OpenCL Compiler
Signal Processing: Matlab Compiler
Machine Learning: TensorFlow Compiler
Networking: Language Compiler

CPU
GPU
DSP
TPU
Domain-specific processors

Computers
- Java
- Compiler
- CPU

Graphics
- OpenCL
- Compiler
- GPU

Signal Processing
- Matlab
- Compiler
- DSP

Machine Learning
- TensorFlow
- Compiler
- TPU

Networking
- P4 Program
- Compiler
- PISA
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 still 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.
Turning the tables “upside down”

“This is precisely how you must process packets” in P4

Switch OS

Run-time API

Driver

PISA device

(Protocol-Independent Switch Architecture)
SDN is necessary, but not sufficient

- SDN allows one to “program” the network control plane
- Large DCs already “program” host networking stacks (hypervisors), essentially doing edge-based SDN
- The last missing piece in the puzzle: The programmable data plane
PISA: An architecture for high-speed programmable packet forwarding
PISA: Protocol Independent Switch Architecture
PISA: Protocol Independent Switch Architecture

Programmable Parser → Ingress → Buffer → Egress
PISA: Protocol Independent Switch Architecture

Match Logic
(Mix of SRAM and TCAM for lookup tables, counters, meters, generic hash tables)

Action Logic
(ALUs for standard boolean and arithmetic operations, header modification operations, hashing operations, etc.)

Programmable Packet Generator

Programmable Parser

Ingress match-action stages (pre-switching)

Buffer

Egress match-action stages (post-switching)

Recirculation

CPU (Control plane)
Why we call it protocol-independent packet processing
Device does not understand any protocols until it gets programmed.

Logical Data-plane View (your P4 program)

Switch Pipeline
Mapping logical data-plane design to physical resources

Logical Data-plane View
(your P4 program)

Switch Pipeline

Programmable Parser

L2 Table
L2 Action Macro

IPv4 Table
IPv4 Action Macro

IPv6 Table
IPv6 Action Macro

ACL Table
ACL Action Macro

Queues

CLK
Re-program in the field

Logical Data-plane View
(your P4 program)

Switch Pipeline

Programmable Parser

L2 Table
L2 Action Macro

MyEncap
My Encap
Action o Action

IPv4
IPv4
IPv6
IPv6

ACL Table
ACL Action Macro

ACL
What exactly does the compiler do?

- PHV
  (Pkt Hdr Vector)
- Cross Bar
- Hash Gen
- Match Table
  (SRAM or TCAM)
- Action & Instr
  Mem
- PHV'
- Programmable
  Parser
- Queues
- CLK
P4 language components

- **Parser Program**
  - State-machine;
  - Field extraction

- **Match Tables + Actions**
  - Table lookup and update;
  - Field manipulation;
  - Control flow

- **Control Flow**

- **Assembly (“deparser”) Program**
  - Field assembly

**No:** memory (pointers), loops, recursion, floating point
What does a P4 program look like?

```
header_type ethernet_t {
  fields {
    dstAddr : 48;
    srcAddr : 48;
  }

  parser parse_ethernet {
    extract(ethernet);
    return select(latest.etherType) {
      0x8100 : parse_vlan;
      ...  
    }
  }
}

header_type my_encap_t {
  fields {
    foo : 12;
    bar : 8;
    baz : 4;
    qux : 4;
    next_protocol : 4;
  }
}
```
What does a P4 program look like?

```p4
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);
}

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

control ingress
{
    apply(l2);
    apply(my_encap);
    if (valid(ipv4) {
        apply(ipv4_lpm);
    } else {
        apply(ipv6_lpm);
    }
    apply(acl);
}
```
What does a P4 program look like?

/* Example: A typical IPv4 routing table */
table ipv4_lpm {
  reads {
    ingress_metadata.vrf : exact;
    ipv4.dstAddr : lpm;
  }
  actions {
    nop;
    l3_l2_switch;
    l3_multicast;
    l3_nexthop;
    l3_ecmp;
    l3_drop;
  }
  size : 65536;
}

These are the only possible actions. Each particular entry in the table is associated with ONE of these.

<table>
<thead>
<tr>
<th>vrf</th>
<th>ipv4.dstAddr / prefix</th>
<th>action</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>192.168.1.0 / 24</td>
<td>l3_l2_switch</td>
<td>port_id=64</td>
</tr>
<tr>
<td>10</td>
<td>10.0.16.0 / 22</td>
<td>l3_ecmp</td>
<td>ecmp_index=12</td>
</tr>
<tr>
<td>1</td>
<td>192.168.0.0 / 16</td>
<td>l3_nexthop</td>
<td>nexthop_index=451</td>
</tr>
<tr>
<td>1</td>
<td>0.0.0.0 / 0</td>
<td>l3_nexthop</td>
<td>nexthop_index=1</td>
</tr>
</tbody>
</table>
P4 development environment

- Open-source P4 development tools
  - P4 compilers & dev tools, reference P4 programs, P4-programmable S/W switch, test framework, etc.
  - Apache 2.0 license
  - Available at [http://github.com/p4lang](http://github.com/p4lang)

- Several other programmable forwarding targets
  - Switches, NICs, etc.
  - Not just for hardware-based, but also software-based P4 programmable devices (P4 BM, OVS, eBPF, VPP, etc.)
What kind of cool things can you do by programming data planes?
Advanced network monitoring, analysis, and diagnostics
- Inband Network Telemetry, Packet history (Postcards)

Custom traffic monitoring and filtering
- FlowRadar [NSDI’16]

Various modes of congestion control
- RCP, XCP, TeXCP, DCQCN++, Timely++

Dynamic source routing
- Flowlet switching, CONGA [SIGCOMM’15], HULA [SOSR’16]

Embedding middlebox functions into switches
- L4 connection load balancing, TCP SYN authentication, etc.

Offloading parts of the distributed apps
- SwitchPaxos [SOSR’15, ACM CCR‘16], SwitchKV [NSDI’16]

Jointly optimizing network and the apps running on it
- Mostly-ordered Multicast [NSDI’15, SOSP’15]

And many more … -- we’re just starting to scratch the surface!
Any questions and critiques ...?

- What about switching, queueing, scheduling, and congestion control?
- How do you update a table in the data plane?
Lessons

- Take feedback from the PL community
  - Stronger types, expressions, lexical scoping, etc.
- P4 is not just for plain, old, boring switches 😊
  - Language should be separated from target architecture
  - Language should be simple & stable and yet extensible !!!?

Experts collaborated via P4.org and introduced P4\textsubscript{16}
Architecture-language separation in P4

Target Consumer (P4 Programmer)

P4.org

Standard Architecture

Standard Library
• Primitive actions
  • extern types

Target-specific Architecture(s)

Target-specific Library
• Primitive actions
  • extern types

Target Provider

Compile

P4 program

Target-specific Library
• Primitive actions
  • extern types

User Library
• P4 code

Auto-generated API

Target Data-plane Configuration
More lessons

- Parallel action-execution semantics really doesn’t fly
- Particularly painful when writing **stateful** networking apps
- Unfortunately lots of exciting novel apps are stateful
  - Measurement, congestion control, active queue management, load balancing, sketch/streaming algorithms, etc.
- Meanwhile, sequential execution semantics introduces new challenges
Under the hood …

pipeline

Stage 1

Stage 2

Stage 16
A machine model for line-rate switches

Packet Header

Stage 1

Stage 2

Stage 16
A machine model for line-rate switches

pipeline

state action unit

state action unit

state action unit

Typical requirement: 1 pkt / nanosecond
A machine model for line-rate switches
A machine model for line-rate switches

A switch’s atoms constitute its instruction set
Stateless vs. stateful operations

Stateless operation: \( \text{pkt.f4} = \text{pkt.f1} + \text{pkt.f2} - \text{pkt.f3} \)

\[
\begin{align*}
\text{pkt.tmp} &= \text{pkt.f1} + \text{pkt.f2} \\
\text{pkt.f4} &= \text{pkt.tmp} - \text{pkt.f3}
\end{align*}
\]

Can pipeline stateless operations
Stateless vs. stateful operations

Stateful operation: \( x = x + 1 \)

\[ \text{pkt.tmp} = x \]
\[ \text{tmp} = 0 \]
\[ \text{pkt.tmp}++ \]
\[ x = \text{pkt.tmp} \]

\[ X = 0 \]

\[ X = 1 \]

\[ X \text{ should be 2, not 1!} \]
Stateless vs. stateful operations

Stateful operation: \( x = x + 1 \)

Cannot pipeline, need atomic operation in h/w
What Domino offers

- Packet transaction: High-level abstraction for data-plane algorithms
  - Lightweight C: C without pointer, loop, and floating point
  - Examples of several algorithms as packet transactions

- Atoms: A representation for switch instruction sets
  - Seven concrete stateful instructions

- Compiler from packet transactions to atoms
  - Allows us to iteratively design switch instruction sets
Packet transactions: The Domino Language

- Packet transaction: block of imperative code
- Transaction runs to completion, one packet at a time, serially

```python
if (count == 9):
    pkt.sample = pkt.src
    count = 0
else:
    pkt.sample = 0
    count++
```

```
p1.sample = 0
p2.sample = 0
p10.sample = 1.2.3.4
```
Stateful atoms can be fairly involved

Update state in one of four ways based on four predicates.

Each predicate can itself depend on the state.
if (count == 9):
    pkt.sample = pkt.src
    count = 0
else:
    pkt.sample = 0
    count++
Designing programmable switches

Focus on stateful atoms, stateless operations are easily pipelined
## Stateful atoms for programmable switches

<table>
<thead>
<tr>
<th>Atom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W</td>
<td>Read or write state</td>
</tr>
<tr>
<td>RAW</td>
<td>Read, add, and write back</td>
</tr>
<tr>
<td>PRAW</td>
<td>Predicated version of RAW</td>
</tr>
<tr>
<td>IfElseRAW</td>
<td>2 RAWs, one each when a predicate is true or false</td>
</tr>
<tr>
<td>Sub</td>
<td>IfElseRAW with a stateful subtraction capability</td>
</tr>
<tr>
<td>Nested</td>
<td>4-way predication (nests 2 IfElseRAWs)</td>
</tr>
<tr>
<td>Pairs</td>
<td>Update a pair of state variables</td>
</tr>
</tbody>
</table>
## Expressiveness of packet transactions

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bloom filter</td>
<td>29</td>
</tr>
<tr>
<td>Heavy hitter detection</td>
<td>35</td>
</tr>
<tr>
<td>Rate-Control Protocol</td>
<td>23</td>
</tr>
<tr>
<td>Flowlet switching</td>
<td>37</td>
</tr>
<tr>
<td>Sampled NetFlow</td>
<td>18</td>
</tr>
<tr>
<td>HULL</td>
<td>26</td>
</tr>
<tr>
<td>Adaptive Virtual Queue</td>
<td>36</td>
</tr>
<tr>
<td>CONGA</td>
<td>32</td>
</tr>
<tr>
<td>CoDel</td>
<td>57</td>
</tr>
</tbody>
</table>
## Compilation results

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>LOC</th>
<th>Most expressive stateful atom required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bloom filter</td>
<td>29</td>
<td>R/W</td>
</tr>
<tr>
<td>Heavy hitter detection</td>
<td>35</td>
<td>RAW</td>
</tr>
<tr>
<td>Rate-Control Protocol</td>
<td>23</td>
<td>PRAW</td>
</tr>
<tr>
<td>Flowlet switching</td>
<td>37</td>
<td>PRAW</td>
</tr>
<tr>
<td>Sampled NetFlow</td>
<td>18</td>
<td>IfElseRAW</td>
</tr>
<tr>
<td>HULL</td>
<td>26</td>
<td>Sub</td>
</tr>
<tr>
<td>Adaptive Virtual Queue</td>
<td>36</td>
<td>Nested</td>
</tr>
<tr>
<td>CONGA</td>
<td>32</td>
<td>Pairs</td>
</tr>
<tr>
<td>CoDel</td>
<td>57</td>
<td>Doesn’t map</td>
</tr>
</tbody>
</table>
## Compilation results

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>LOC</th>
<th>Most expressive stateful atom required</th>
<th>Pipeline Depth</th>
<th>Pipeline Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bloom filter</td>
<td>29</td>
<td>R/W</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Heavy hitter detection</td>
<td>35</td>
<td>RAW</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>Rate-Control Protocol</td>
<td>23</td>
<td>PRAW</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Flowlet switching</td>
<td>37</td>
<td>PRAW</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Sampled NetFlow</td>
<td>18</td>
<td>IfElseRAW</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>HULL</td>
<td>26</td>
<td>Sub</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Adaptive Virtual Queue</td>
<td>36</td>
<td>Nested</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>CONGA</td>
<td>32</td>
<td>Pairs</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>CoDel</td>
<td>57</td>
<td>Doesn’t map</td>
<td>15</td>
<td>3</td>
</tr>
</tbody>
</table>

~100 atom instances are sufficient
Modest cost for programmability

- All atoms meet timing at 1 GHz in a 32-nm library.
- They occupy modest additional area relative to a switching chip.

<table>
<thead>
<tr>
<th>Atom</th>
<th>Description</th>
<th>Atom area (micro m^2)</th>
<th>Area for 100 atoms relative to 200 mm^2 chip</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W</td>
<td>Read or write state</td>
<td>250</td>
<td>0.0125%</td>
</tr>
<tr>
<td>RAW</td>
<td>Read, add, and write back</td>
<td>431</td>
<td>0.022%</td>
</tr>
<tr>
<td>PRAW</td>
<td>Predicated version of RAW</td>
<td>791</td>
<td>0.039%</td>
</tr>
<tr>
<td>IfElseRAW</td>
<td>2 RAWs, one each when a predicate is true or false</td>
<td>985</td>
<td>0.049%</td>
</tr>
<tr>
<td>Sub</td>
<td>IfElseRAW with a stateful subtraction capability</td>
<td>1522</td>
<td>0.076%</td>
</tr>
<tr>
<td>Nested</td>
<td>4-way predication (nests 2 IfElseRAWs)</td>
<td>3597</td>
<td>0.179%</td>
</tr>
<tr>
<td>Pairs</td>
<td></td>
<td></td>
<td>&lt;1 % additional area for 100 atom instances</td>
</tr>
</tbody>
</table>
P4\textsubscript{16} adopts the Domino concepts

• Expressions
• Sequential action-execution semantics
  • It’s compiler’s responsibility to recognize and take advantage of parallelizable opportunities in the code
• Annotations to express atomic-execution (non-reentrant) semantics
Questions and critiques ...?
Recapping: Why is data-plane programmability a big deal?
Key benefits of programmable forwarding

1. **New features**: Add new protocols
2. **Reduce complexity**: Remove unused protocols
3. ** Efficient use of resources**: Flexible use of tables
4. **Greater visibility**: New diagnostics, telemetry, OAM etc.
5. **Modularity**: Compose forwarding behavior from libraries
6. **Portability**: Specify forwarding behavior once; compile to many devices
7. **Own your own IP**: No need to tell the chip vendor your features