Welcome.
The P4 Community

Nick McKeown
Stanford University
At some level, we all know why we want programmability....
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. A new abstraction: Programming rather than protocols
A long-term aspiration

Declared network forwarding behavior

Partition and generate code

P4 code
P4 code
P4 code
P4 code
P4 code
P4 code

P4 Compiler

NIC

NIC
Why now?
Domain Specific Processors

1. Computers
   - Applications
   - Compiler
   - CPU

2. Graphics
   - Applications
   - Compiler
   - GPU

3. Signal Processing
   - Applications
   - Compiler
   - DSP

4. Networks
   - Applications
   - Compiler
   - ?
But isn’t programmability too expensive in networking?
Packet Forwarding Speeds

Gb/s (per chip)


0.1 1 10 100 1000 10000 3.2Tb/s
Packet Forwarding Speeds

- 1990: 0.1 Gb/s
- 1995: 1 Gb/s
- 2000: 10 Gb/s
- 2005: 100 Gb/s
- 2010: 1000 Gb/s
- 2015: 10000 Gb/s
- 2020: 100000 Gb/s

3.2 Tb/s
50x

Switch Chip
CPU

Gb/s (per chip)

What has been happening

A collective stepping back

– Recognized the value of the “match + action” abstraction

– Picked the basic *plumbing primitives*

– Created more flexible pipelines, less protocol dependent

– Exploited the inherent parallelism

Fulcrum/Intel, Cisco Doppler, Netronome, Xpliant/Cavium, Huawei, “RMT”, etc.
A more general architectural approach is emerging
PISA: Protocol Independent Switch Architecture

Abstract forwarding model (for P4 v1.0)
Compiler target for P4 programs
P4 and PISA

P4 code

Compiler

Compiler Target

Parser

Match Table  Action Macro

Match Table  Action Macro

Match Table  Action Macro

Match Table  Action Macro

Queues

14
P4 and PISA

Parser

```
parse ethernet {
    extract(ethernet);
    select(latest.etherType) {
        0x800 : parse_ipv4;
        0x86DD : parse_ipv6;
    }
}
```

Match Action Tables

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

Control Flow Graph

```
control ingress {
    apply(l2_table);
    if (valid(ipv4)) {
        apply(ipv4_table);
    }
    if (valid(ipv6)) {
        apply(ipv6_table);
    }
    apply(acl);
}
```

Queues
Mapping Control Flow to PISA Target

Control Flow Graph
Switch Pipeline

Parser
L2 Table
L2 Action Macro
IPv4 Table
IPv4 Action Macro
IPv6 Table
IPv6 Action
ACL Table
ACL Action Macro

Queues
Why “Protocol Independence”?

- Programmable in the field
- IP belongs to the programmer
- Breaks the stranglehold of the “lock-in” APIs
- Faster innovation
Can we create a common language?

Existing work: packetC, Netcore, Frenetic, Pyretic, NetKAT, ...
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 protocol. (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 in many ways, having a common, open, vendor-agnostic interface (like OpenFlow) enables a control plane to control forwarding devices from different hardware and software vendors.

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., VXLAN, STT), for which they resort to deploying software switches that are easier to extend with new functionality. 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 microcode programming. In this paper, we sketch the design of a higher-level language for configuring custom packet processors (i.e., a new “OpenFlow 2.0” API). Such a language would simplify, more elegant, and more future-proof than today’s OpenFlow 1.x standard.
P4 Language Spec
P4 tools: compilers, debugging, verification
P4 programs: library

Protocol Independent
P4 programs specify how a switch processes packets.

Target Independent
P4 is suitable for describing everything from high-performance forwarding ASICS to software switches.

Field Reconfigurable
P4 allows network engineers to change the way their switches process packets after they are deployed.
<The End>