Sharif University of Technology

Computer Engineering Department

Software-Defined Networking

Ali Movaghar

Mohammad Hosseini

Data Plane Programming
Part 1

TA: Iman Rahmati & Farbod Shahinfar

Data Plane Programmability

ACM SIGCOMM CCR 2014

P4: Programming Protocol-Independent Packet Processors

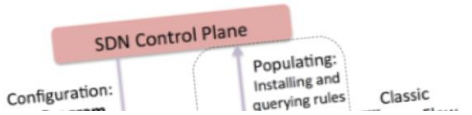
Pat Bosshart[†], Dan Daly[*], Glen Gibb[†], Martin Izzard[†], Nick McKeown[‡], Jennifer Rexford[**], Cole Schlesinger[**], Dan Talayco[†], Amin Vahdat[¶], George Varghese[§], David Walker[**]
[†]Barefoot Networks [*]Intel [‡]Stanford University [**]Princeton University [¶]Google [§]Microsoft Research

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 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 hardware. As an example, we describe how to

multiple stages of 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 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.



Recapping SDN

- SDN gives operators programmatic control over their networks.
- In SDN, the control plane is physically separate from the forwarding plane.
- One control plane controls multiple forwarding devices, by a common, open, vendor-agnostic interface like OpenFlow.

Proliferation of Header Fields

- The OpenFlow interface started simple, with the abstraction of a single table of rules that could match packets on a dozen header fields (e.g., MAC addresses, IP addresses, protocol, TCP/UDP port numbers, etc.).

- Over the years, the specification has grown increasingly more complicated, with many more header fields.

Version	Date	Header Fields
OF 1.0	Dec 2009	12 fields (Ethernet, TCP/IPv4)
OF 1.1	Feb 2011	15 fields (MPLS, inter-table metadata)
OF 1.2	Dec 2011	36 fields (ARP, ICMP, IPv6, etc.)
OF 1.3	Jun 2012	40 fields
OF 1.4	Oct 2013	41 fields

New Headers

❖The proliferation of new header fields shows no signs of stopping.

A New Mechanism

- Rather than repeatedly extending the OpenFlow specification, the paper proposes 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)

Future SDN Switches

- Configurable packet parser
  - Not tied to a specific header format
- Flexible match+action tables
  - Multiple tables (in series and/or parallel)
  - Able to match on all defined fields
- General packet-processing primitives
  - Copy, add, remove, and modify
  - For both header fields and meta-data

It Is Feasible

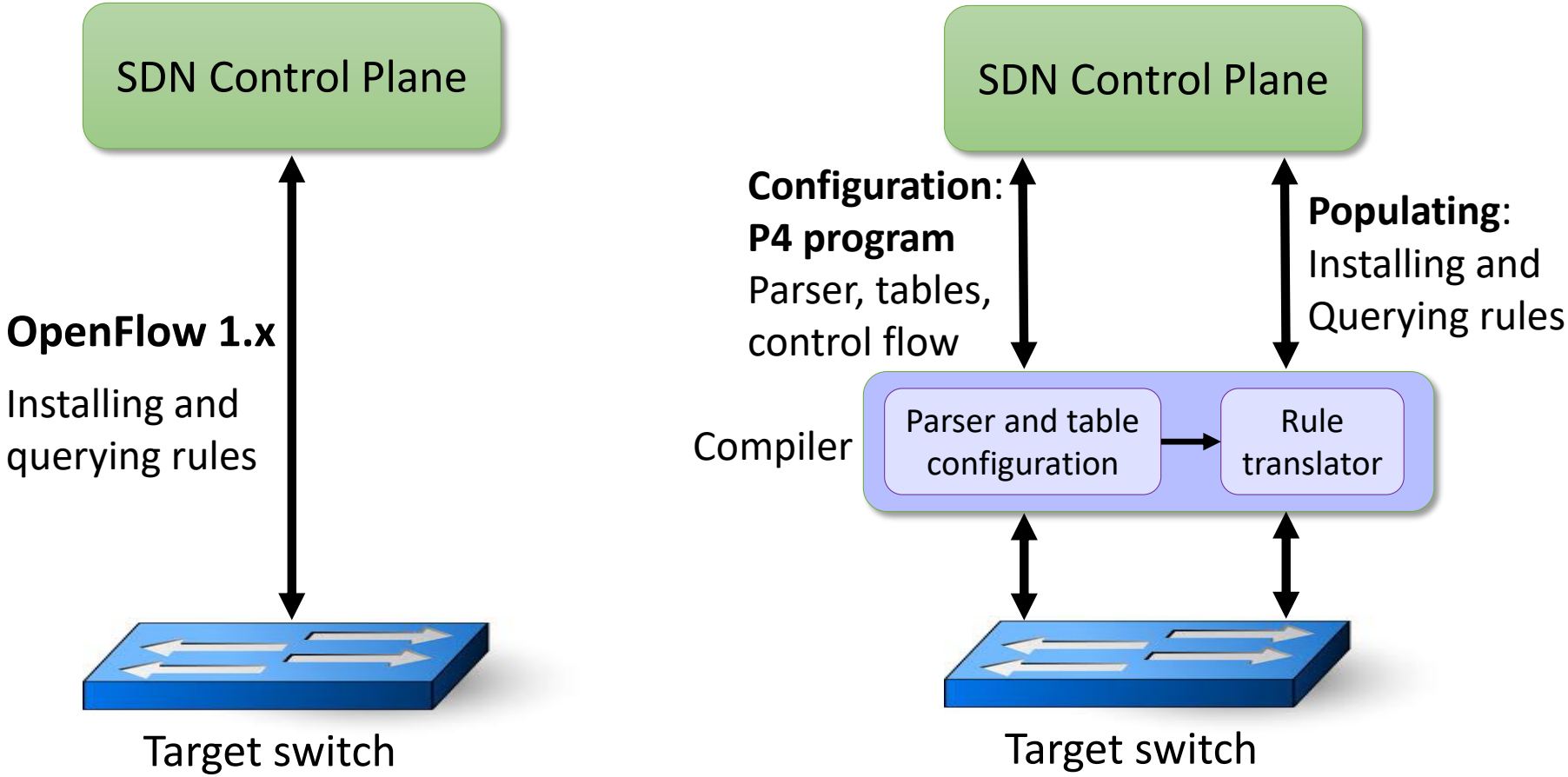
- Recent chip designs demonstrate that such flexibility can be achieved in custom ASICs at terabit speeds [1-3].

[1] “Leaping multiple headers in a single bound: Wire-speed parsing using the Kangaroo system," IEEE INFOCOM, 2010.
[2] Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," ACM SIGCOMM, 2013.
[3] “Intel Ethernet Switch Silicon FM6000” (Intel FlexPipe)

- Custom, vendor-specific interfaces.
- Each chip has its own low-level interface, similar to microcode programming.

P4

❖ The paper proposes a high-level language for **P**rogramming **P**rotocol-independent **P**acket **P**rocessors (P4).



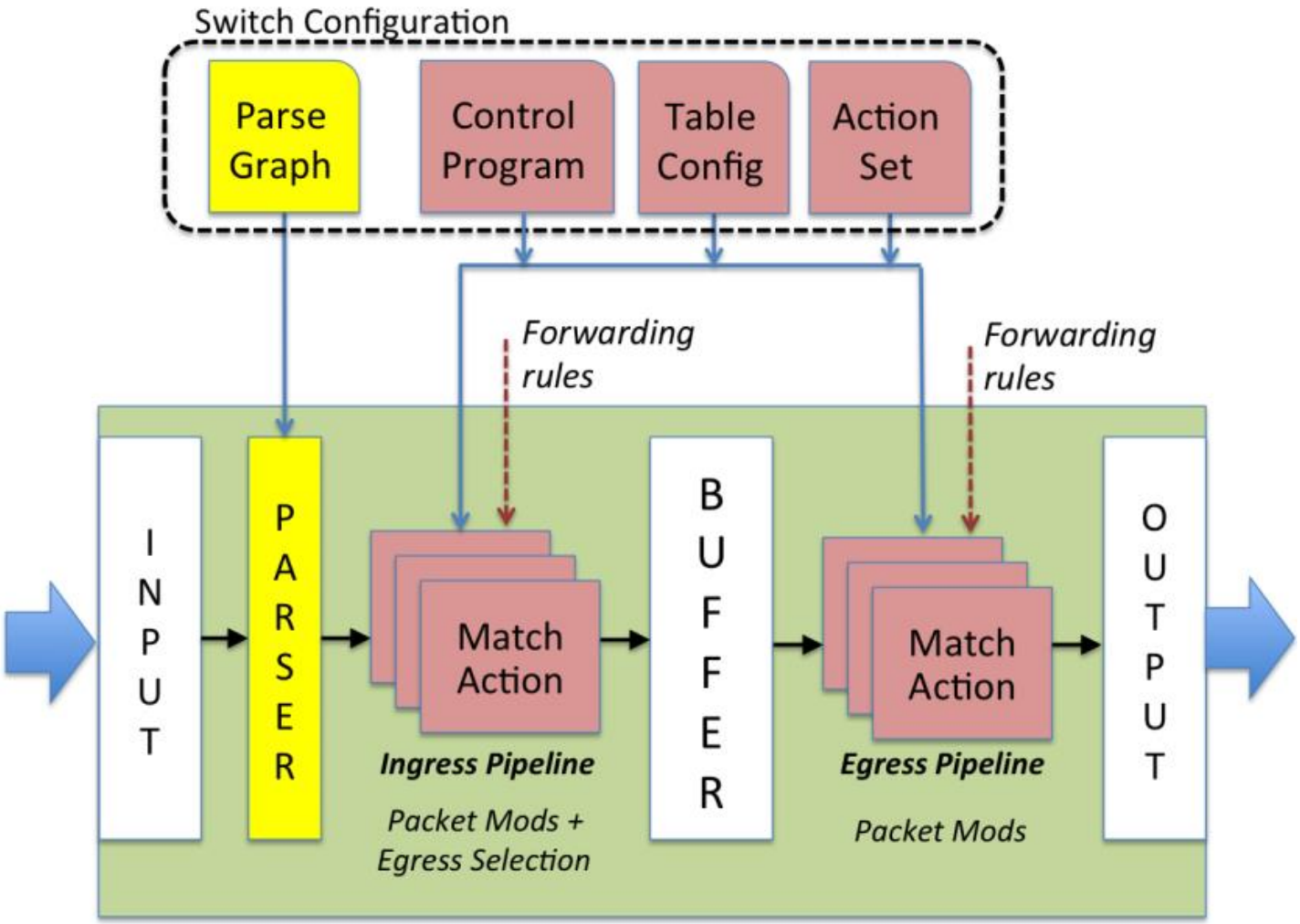
Main Goals of P4

- **Reconfigurability**: The controller should be able to redefine the packet parsing and processing **in the field**.

- **Protocol independence**: The switch should not be tied to specific packet formats.

- **Target independence**: The programmer should not need to know the details of the underlying switch. Instead, a **compiler** should take the switch's capabilities into account when turning a target-independent description (written in P4) into a target-dependent program (used to configure the switch).

Abstract Forwarding Model



Abstract Forwarding Model

❖The abstract model generalizes how packets are processed

in different forwarding devices (e.g., Ethernet switches, loadbalancers, routers)

and

by different technologies (e.g., reconfigurable switches, software switches, FPGAs, NPUs, and even fixed-function switch ASICs,).

Abstract Forwarding Model

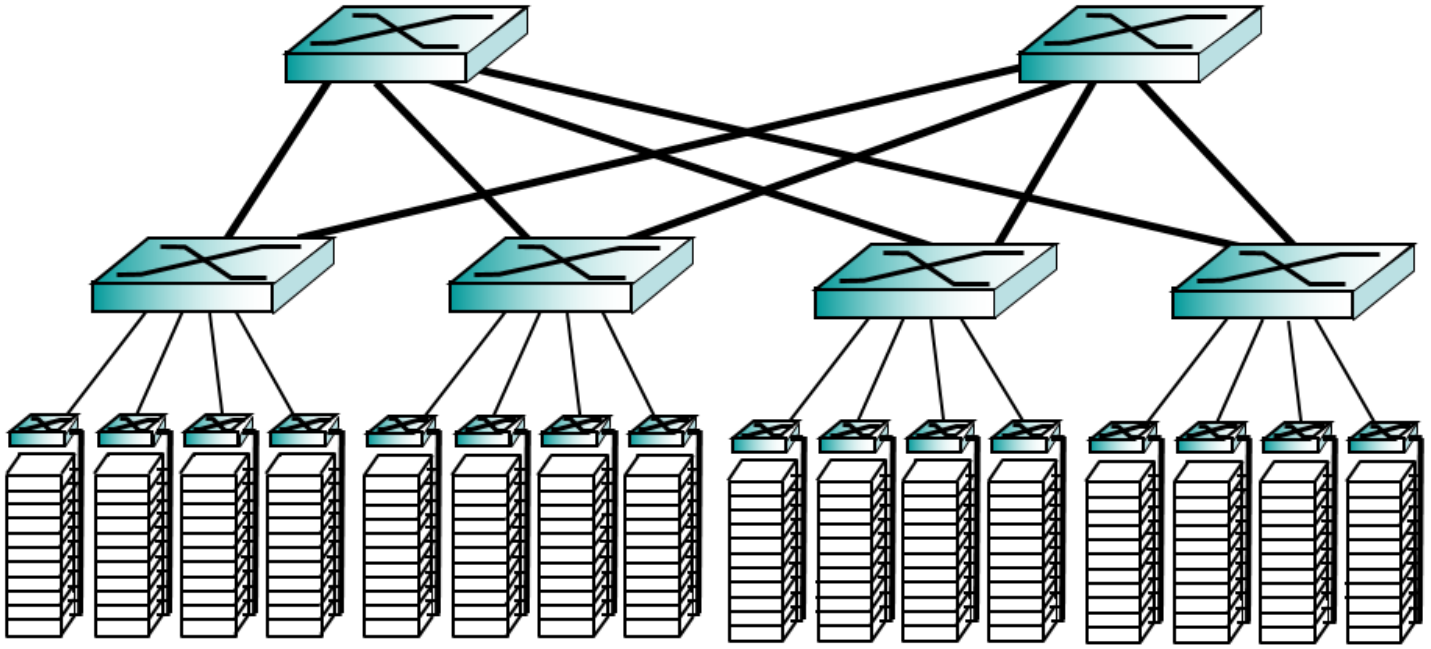
- OpenFlow assumes a fixed parser, whereas the proposed model supports a **programmable parser** to allow new headers to be defined.
- OpenFlow assumes the match+action stages are in series, whereas in the proposed model they can be in parallel or in series.
- The proposed model assumes that actions are composed from protocol-independent primitives supported by the switch.

Two Types of Operations

- Configure: configure operations program the parser, set the order of match+action stages, and specify the header fields processed by each stage.
  - Configuration determines which protocols are supported and how the switch may process packets.

- Populate: populate operations add (and remove) entries to the match+action tables that were specified during configuration.
  - Population determines the policy applied to packets at any given time.

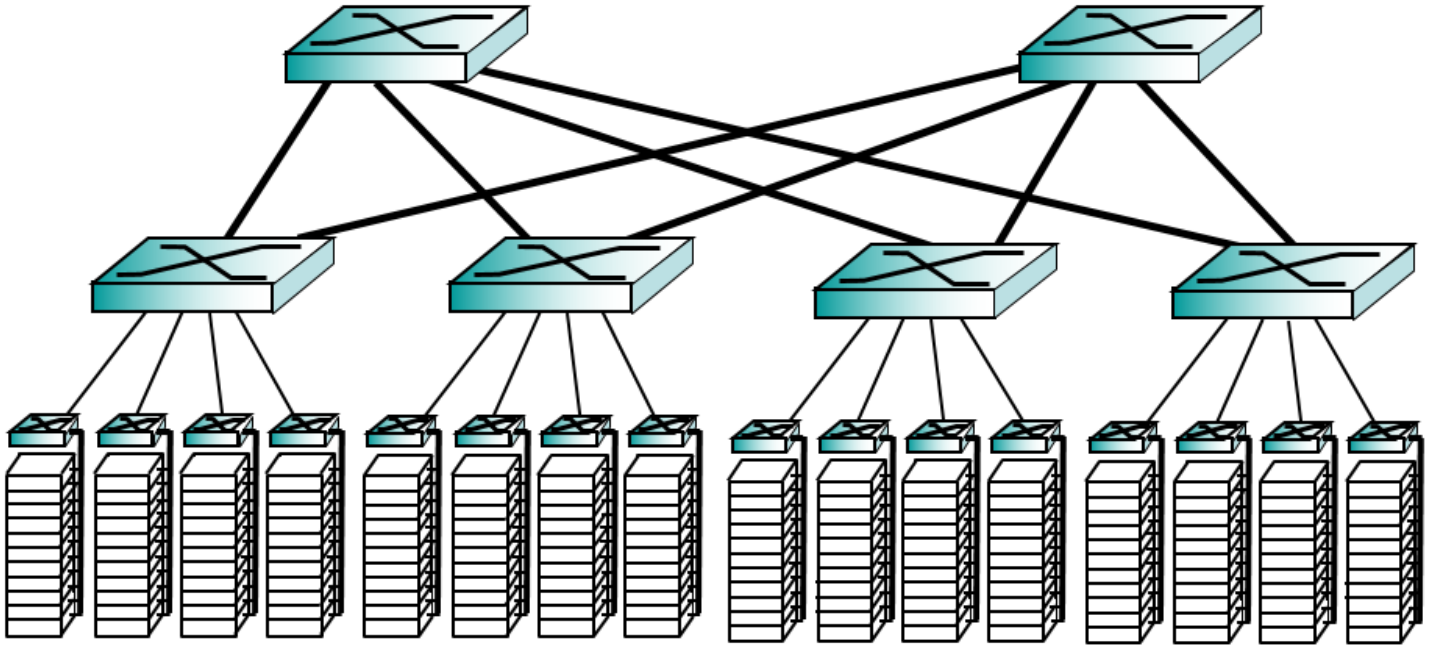
Simple Motivating Example



- L2 network deployment in a data-center
  - Top-of-rack (ToR) switches at the edge connected by a two-tier core

- Assume the number of end-hosts is growing and the core L2 tables are overflowing.

Simple Motivating Example



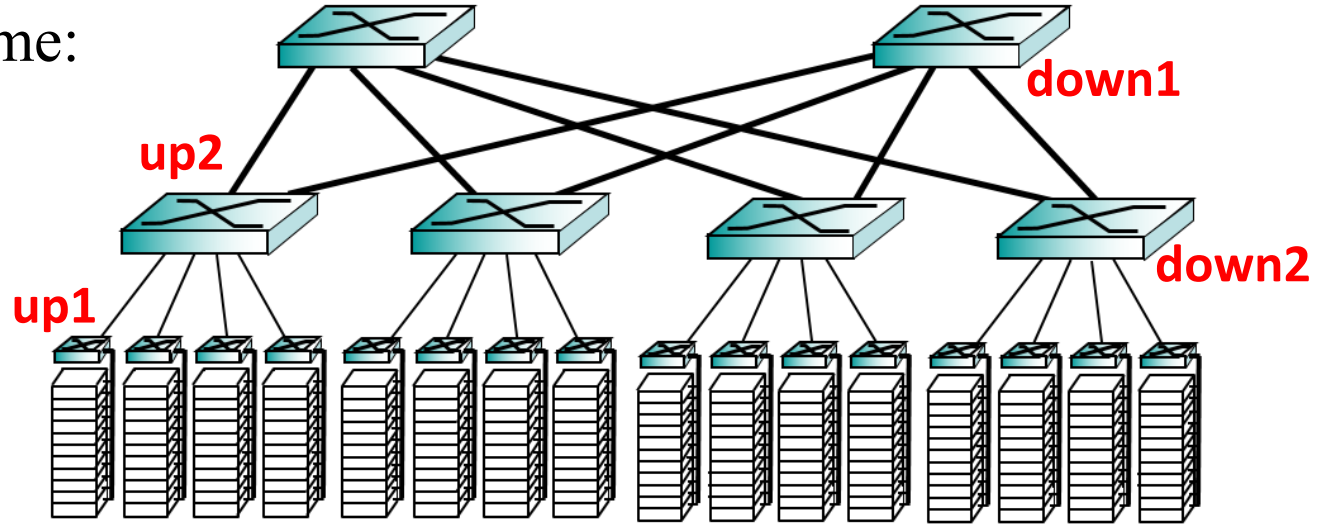
- MPLS
- PortLand [1]

[1] “PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric”, ACM SIGCOMM 2009

A Custom Solution Using P4

The solution name:

mTag



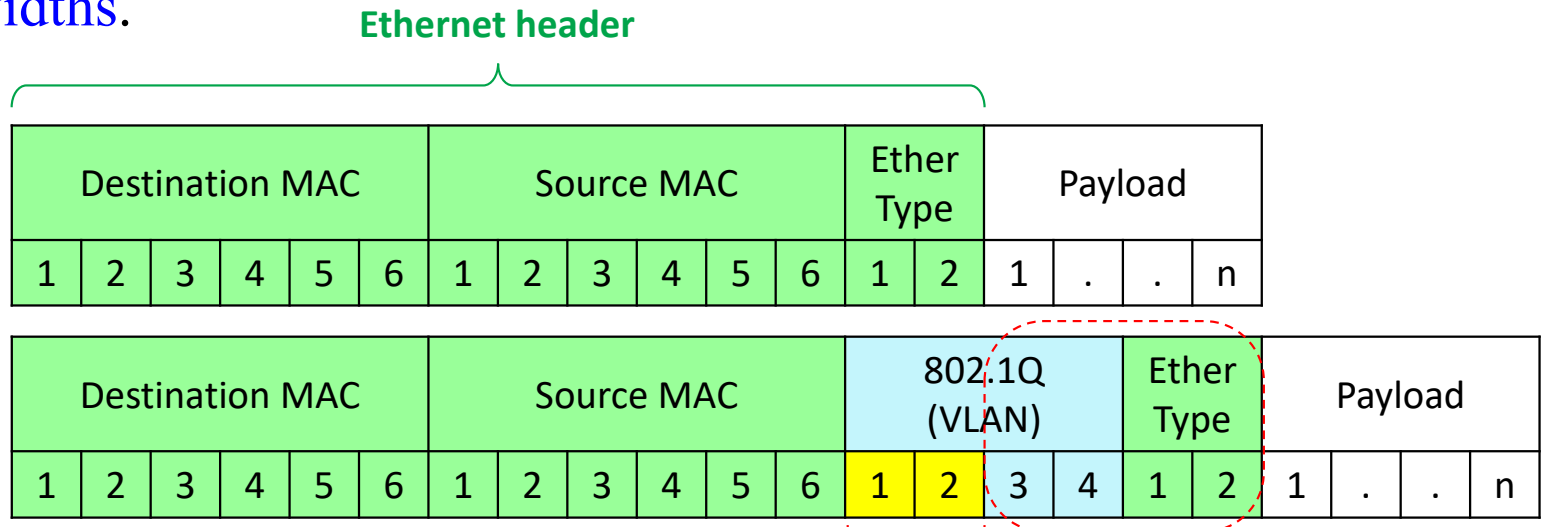
- The routes through the core are encoded by a 32-bit tag composed of four single-byte fields.
- The tag can carry a "**source route**"
- Each core switch need only examine one byte of the tag and switch on that information.
- In the example, the tag is added by the first ToR switch, although it could also be added by the end-host NIC.

P4 Components

- **Headers**: a header definition describes the sequence and structure of a series of fields.
- **Parsers**: a parser definition specifies how to identify headers and valid header sequences within packets.
- **Tables**: match+action tables are the mechanism for performing packet processing.
  - The P4 program defines the fields on which a table may match and the actions it may execute.
- **Actions**: P4 supports construction of complex actions from simpler protocol-independent **primitives**.
  - These complex actions are available within the tables.
- **Control Programs**: The control program determines the order of match+action tables that are applied to a packet.

Headers

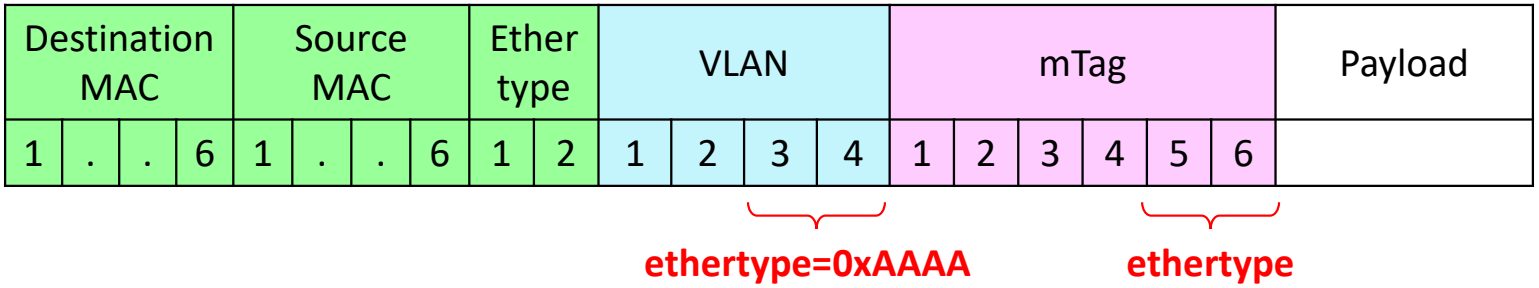
❖ Each header is specified by declaring an ordered list of field names together with their widths.




```
header ethernet {
  fields {
    dst_addr : 48;
    src_addr : 48;
    ethertype : 16;
  }
}
```

```
header vlan {
  fields {
    pcp : 3;
    cfi : 1;
    vid : 12;
    ethertype : 16;
  }
}
```

Headers




```
header ethernet {
  fields {
    dst_addr : 48;
    src_addr : 48;
    ethertype : 16;
  }
}
```

```
header vlan {
  fields {
    pcp : 3;
    cfi : 1;
    vid : 12;
    ethertype : 16;
  }
}
```

```
header mTag {
  fields {
    up1 : 8;
    up2 : 8;
    down1 : 8;
    down2 : 8;
    ethertype : 16;
  }
}
```

- Each core switch is programmed with rules to examine one of these bytes determined by its location in the hierarchy and the direction of travel (up or down).

Packet Parser

```
parser start {
    ethernet;
}
parser ethernet {
    switch(ethertype) {
        case 0x8100 : vlan;
        case 0x9100 : vlan;
        case 0x800 : ipv4;
    }
}
parser vlan {
    switch(ethertype) {
        case 0xaaaa : mTag;
        case 0x800 : ipv4;
    }
}
parser mTag {
    switch(ethertype) {
        case 0x800 : ipv4;
    }
}
```

- P4 assumes the underlying switch can implement a state machine that traverses packet headers from start to finish, extracting field values as it goes.
- The extracted field values are sent to the match+action tables for processing.
- P4 describes this state machine directly as the set of transitions from one header to the next.

Tables

- Describe each packet-processing stage
  - What fields are matched, and in what way (exact, range, wildcard)
  - What action functions are performed
  - (Optionally) a hint about max number of rules
- In the simple *mTag* example, edge switches match on the L2 destination and VLAN ID, and select an mTag to add to the header.
- The programmer defines a table to match on these fields and apply an action to add the *mTag* header.
- The table specification allows a compiler to decide how much memory it needs (width and height), and the memory type (e.g., TCAM or SRAM) to implement the table.

```
table mTag_table {
  reads {
    ethernet.dst_addr : exact;
    vlan.vid : exact;
  }
  actions {
    add_mTag;
  }
  max_size : 20000;
}
```

Tables

```
table source_check {
    // Verify mtag only on ports to the core
    reads {
        mtag : valid; // Was mtag parsed?
        metadata.ingress_port : exact;
    }
    actions {
        // If inappropriate mTag, send to CPU
        fault_to_cpu;
        // If mtag found, strip and record in metadata
        strip_mtag;
        // Otherwise, allow the packet to continue
        pass;
    }
    max_size : 64;
}
```

Tables

```
table local_switching {
    // Reads destination and checks if local
    // If miss occurs, goto mtag table.
}
```

```
table egress_check {
    // Verify egress is resolved
    // Do not retag packets received with tag
    // Reads egress and whether packet was mTagged
}
```

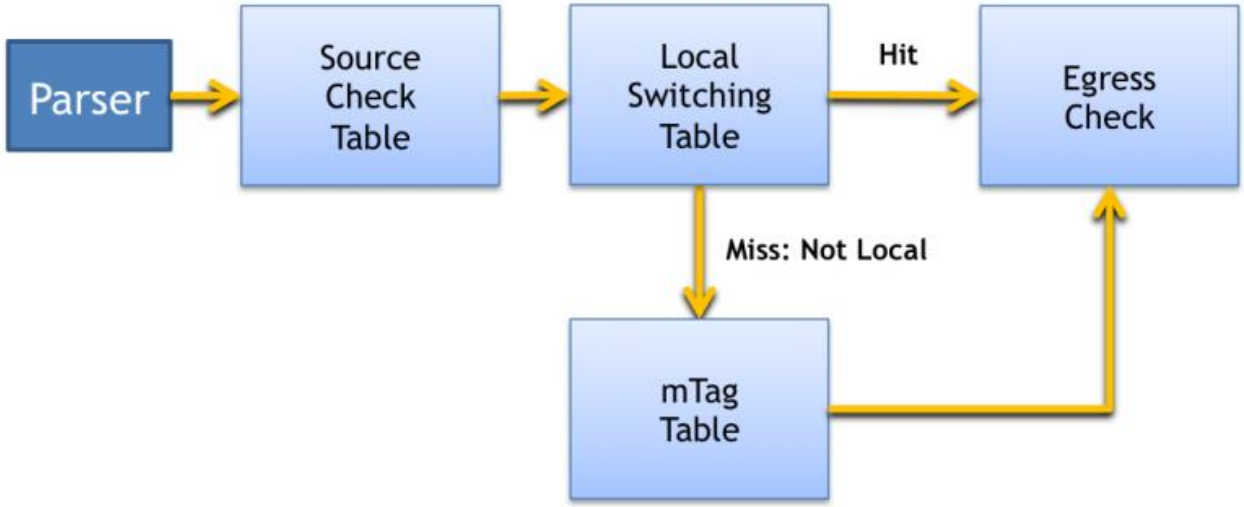
Actions

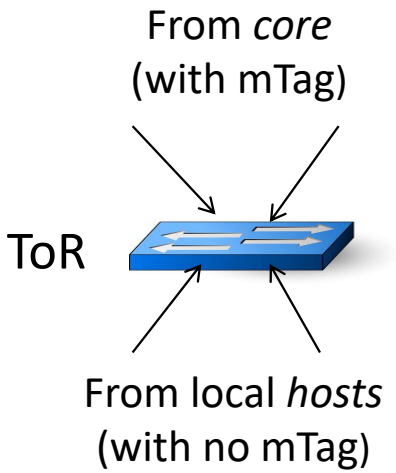
- Custom actions built from primitives
  - add_header, remove_header, copy_field, set_field, increment, checksum

```
action add_mTag(up1, up2, down1, down2, egr_spec) {

    add_header(mTag);

    // Copy VLAN ethertype to mTag
    copy_field(mTag.ethertype, vlan.ethertype);
    // Set VLAN's ethertype to signal mTag
    set_field(vlan.ethertype, 0xaaaa);

    set_field(mTag.up1, up1);
    set_field(mTag.up2, up2);
    set_field(mTag.down1, down1);
    set_field(mTag.down2, down2);

    // Set the destination egress port as well
    set_field(metadata.egress_spec, egr_spec);
}
```

The Control Program

- Flow of control from one table to the next
  - Collection of functions, conditionals, and tables
- For ToR (edge) switches:





The Control Program

❖ For ToR (edge) switches:

```
control main() {
    // Verify mTag state and port are consistent
    table(source_check);
    // If no error from source_check, continue
    if (!defined(metadata.ingress_error)) {
        // Attempt to switch to end hosts
        table(local_switching);

        if (!defined(metadata.egress_spec)) {
            // Not a known local host; try mtagging
            table(mTag_table);
        }
        // Check for unknown egress state or
        // bad retagging with mTag
        table(egress_check);
    }
}
```

P4 Compilation

- Parser
  - Programmable parser: the compiler translates the parser description into a parsing state machine.
  - Fixed parser: the compiler merely verifies that the parser description is consistent with the target's parser.
- Control program
  - In the first stage, the compiler converts the P4 control program into an intermediate *table dependency graph* representation.
  - In the second stage, a target-specific back-end maps this graph onto the switch's specific resources.
- Rule translation
  - Verify that rules agree with the table types
  - Translate the rules to the physical tables