P4 Compiler in SDN

Federico Bruzzone[1], PhD Student

Milan, Italy – 03 November 2024

Slides available at: federicobruzzone.github.io/activities/presentations/p4-compiler-in-SDN.pdf

[1] ADAPT Lab – Università degli Studi di Milano,
Website: federicobruzzone.github.io,
Github: github.com/FedericoBruzzone,
Email: federico.bruzzone@unimi.it

Network Programmability

The ability of the software or the hardware to extecute an externally defined processing algorithm[2]

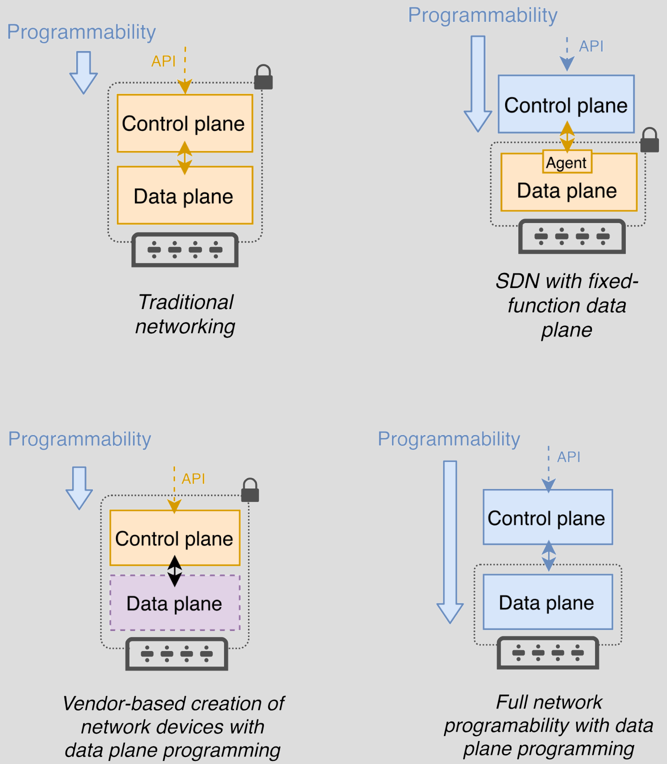
[2] Hauser et al., "A Survey on Data Plane Programming with P4: Fundamentals, Advances, And Applied Research".

Open Networking Foundation (ONF)

- Non-profit consortium founded in 2011
- Promotes networking through **Software Defined Networking** (SDN)
- Standardizes the **OpenFlow** protocol

Software Defined Networking (SDN)

- Born to overcome the limitations of traditional network architectures
- Decouples the control plane from the data plane
- Centralizes the control of the network



OpenFlow Protocol

- Gives access to the **forwarding plane** (data plane) of a network device
- Mainly used by switches and controllers
- Layered on top of the **Transport Control Protocol** (TCP)
- De-facto standard for SDN

OpenFlow Development

- First appeared in 2008[3]
- In April 2012, Google deploys OpenFlow in its internal network with significant improvements (Urs Hölzle promotes it[4])
- In January 2013, NEC rolls out OpenFlow for Microsoft Hyper-V
- Latest version is 1.5.1 (Apr 2015)

---

[3] McKeown et al., "Openflow: Enabling Innovation in Campus Networks".

[4] Inter-Datacenter WAN with centralized TE using SDN and OpenFlow.

Fields in OpenFlow Standard

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 (APR, ICMP, IPv6, etc.)
OF 1.3	Jun 2012	40 fields
OF 1.4	Oct 2013	41 fields

More Details on the OpenFlow v1.0.0 Switch Specification[5]

[5] https://opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf

OpenFlow is protocol-dependent

Fixed set of fields and parser based on standard protocols

(Ethernet, IPv4/IPv6, TCP/UDP)

P4: Programming Protocol-Independent Packet Processors

Bosshart believed that future generations of OpenFlow would have allowed the controller to *tell the switch how to operate*[6]

[6] Bosshart et al., "P4: Programming Protocol-Independent Packet Processors".

Goals and Challenges

Reconfigurability: the controller should be able to redefine the packet parsing and processing in the field

Protocol Independence: the switch should *headers* using parsing and processing using *match+action* tables

Target Independence: a compiler from *target-independent* description to *target-dependent* binary

Goals and Challenges

Reconfigurability: the controller should be able to redefine the packet parsing and processing in the field

Protocol Independence: the switch should *headers* using parsing and processing using *match+action* tables

Target Independence: a compiler from *target-independent* description to *target-dependent* binary

Goals and Challenges

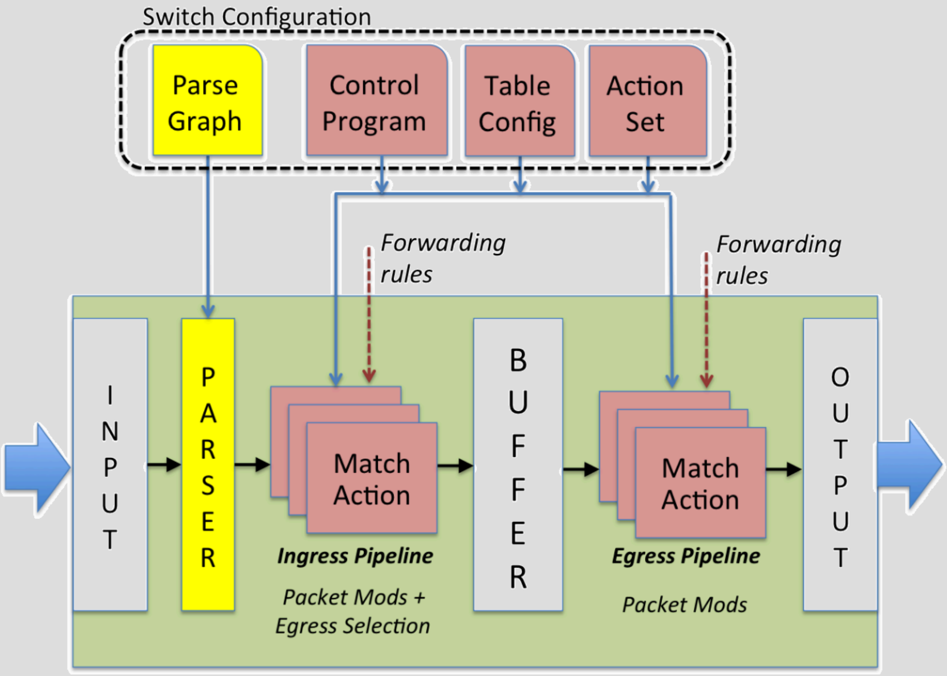
Reconfigurability: the controller should be able to redefine the packet parsing and processing in the field

Protocol Independence: the switch should *headers* using parsing and processing using *match+action* tables

Target Independence: a compiler from *target-independent* description to *target-dependent* binary

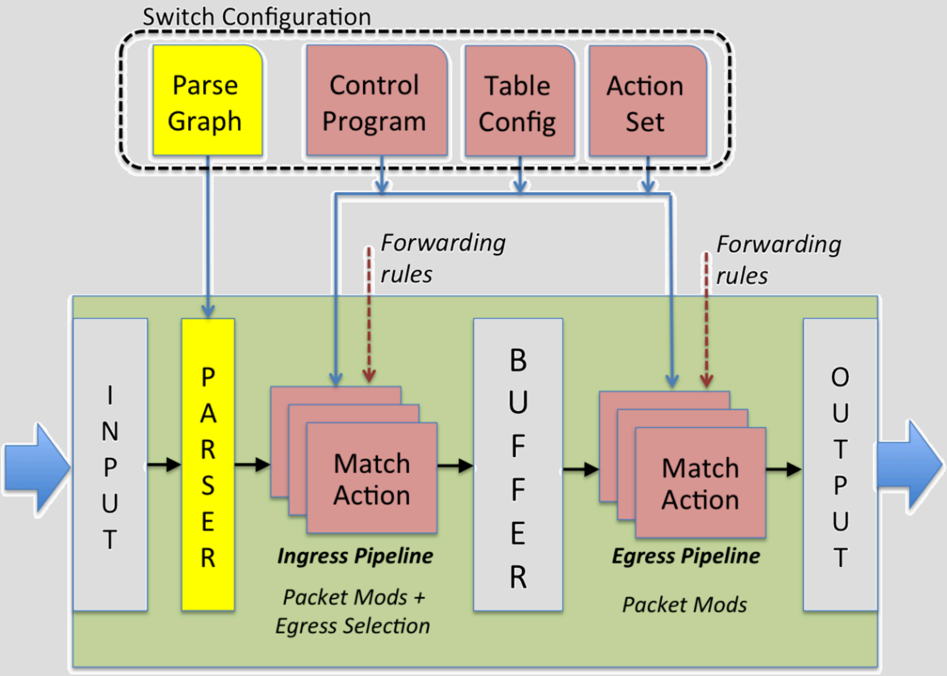
Abstract Forwarding Model (AFM)

1. Parsing the packet headers
2. The fields are passed to the match-action pipeline.
   - **Ingrees**: determines the egress port/queue
   - **Egress**: per-instance header modifications
3. Metadata processing (e.g., timestamp)
4. As in OpenFlow, the queuing discipline is chosen at switch configuration time (e.g., minimum rate)



Abstract Forwarding Model (AFM)

1. Parsing the packet headers
2. The fields are passed to the match-action pipeline.
   - **Ingrees**: determines the egress port/queue
   - **Egress**: per-instance header modifications
3. Metadata processing (e.g., timestamp)
4. As in OpenFlow, the queuing discipline is chosen at switch configuration time (e.g., minimum rate)



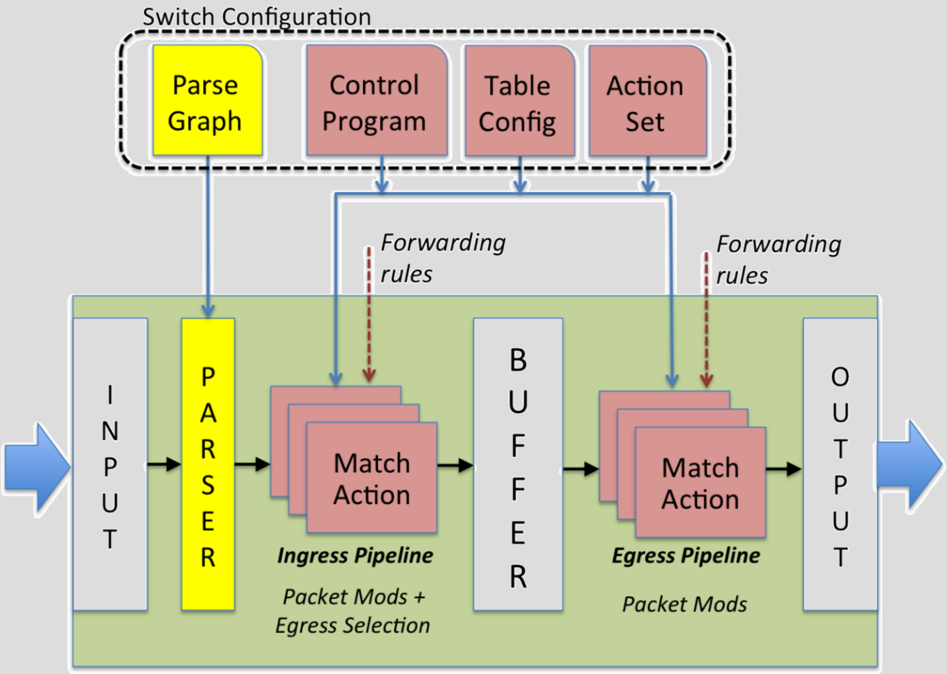
Abstract Forwarding Model (AFM)

1. Parsing the packet headers

2. The fields are passed to the match-action pipeline.
   - **Ingrees**: determines the egress port/queue
   - **Egress**: per-instance header modifications

3. Metadata processing (e.g., timestamp)

4. As in OpenFlow, the queuing discipline is chosen at switch configuration time (e.g., minimum rate)



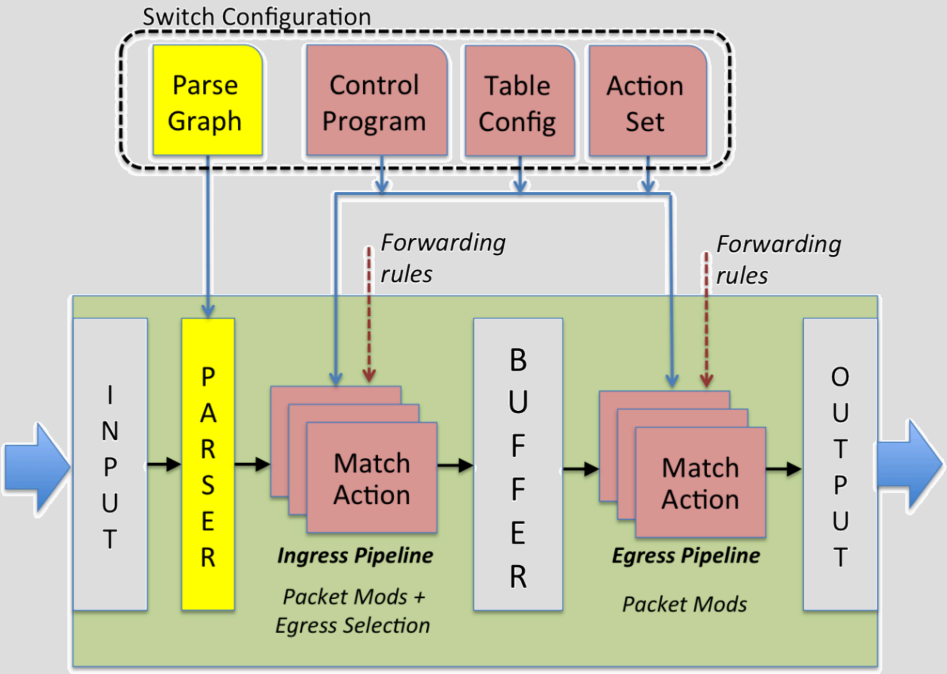
Abstract Forwarding Model (AFM)

1. Parsing the packet headers

2. The fields are passed to the match-action pipeline.
   - **Ingrees**: determines the egress port/queue
   - **Egress**: per-instance header modifications

3. Metadata processing (e.g., timestamp)

4. As in OpenFlow, the queuing discipline is chosen at switch configuration time (e.g., minimum rate)

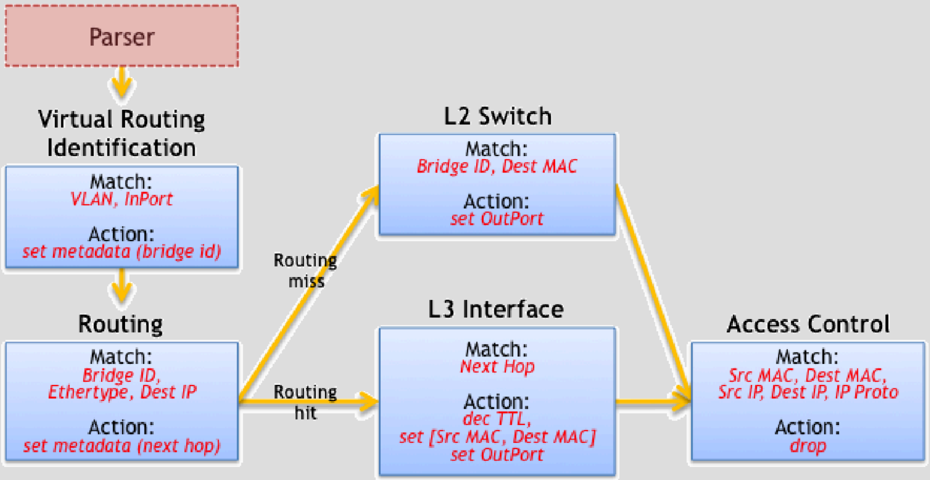


Two-stage Compilation

Imperative control flow program based on **AFM**

1. The compiler translate the P4 program into **TDGs** (Table Dependency Graphs)

2. The **TDGs** are compiled into **target-dependent** code



Real Case Scenario

Setup: **L2 Network Architecture**

- *Edge (top-of-rack switches)*: connect end-hosts to the network
- *Core*: central layer that connects the edge devices

Problem: **Growing End-Hosts and Overflowing Tables**

- The L2 forwarding tables in the *core* are becoming too large → **overflow**
- It can cause *packet loss* and *network congestion*

Solutions: **Muti-protocol Label Switching and PortLand**

- *MPLS*: a technique that uses labels to make data forwarding decisions → **with multiple tags is daunting**
- *PortLand*: a scalable L2 network architecture → **rewrite MAC addresses**

P4: Language Design

TODO

Header

Describes the structure of a series of fields and constraints on values

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

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

Header (Cont.)

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

- *mTag* can be added without altering the existing headers
- The core has two layers of aggregation
- Each core switch examines on of these bytes detemined by its **location** and the **direction** of the packet

Parser

Specifies how to identify headers and valid header sequences

```
parser start { ethernet; }
```

```
parser ethernet {
  switch(ethertype) {
    case 0x8100: vlan;
    case 0x9100: vlan;
    case 0x800: ipv4;
    // Other cases
  }
}
```

```
parser vlan {
  switch(ethertype) {
    case 0xaaaa: mTag;
    case 0x800: ipv4;
    // Other cases
  }
}
```

Parser (Cont.)

```
parser mTag {
  switch(ethertype) {
    case 0x800: ipv4;
    // Other cases
  }
}
```

- Reached a state for a new header, the State Machine extracts the header and sends it to the match+action pipeline
- The parser for *mTag* is simple, it has only four states

Table

Defines the fields to match on and the actions to take

```
table mTag_table {
  reads {
    ethernet.dst_addr: exact;
    vlan.vid: exact;
  }
  actions {
    // At runtime, entries are
    // programmed with params
    // for the mTag action.
    add_mTag;
  }
  max_size: 20000;
}
```

The compiler knows what memory type use (e.g., TCAM, SRAM) and the amount of memory to allocate

- `reads`: the edge switch matches on the L2 destination address and the VLAN ID
- `actions`: selects an *mTag* to add to the header
- `max_size`: the maximum number of entries

Action

Construction of actions from simpler protocol-independent primitives

```
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);
}
```

- P4 assumes parallel execution
- Parameters are passed from the match table at runtime
- The switch inserts the *mTag* afer the VLAN header

Control Programs

Determines the order of match+action tables that are applied to a packet

```
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);
  }
}
```

- *mTag* should only be seen on ports to the core
- `source_check` strips the *mTag* and records it in the metadata to avoid retagging
- If the `local_switching` table misses, the packet is not destined for a local host
- Both *local* and *core* forwarding control is handled by the `egress_check` table
- If unknown destination, the SDN controller is notified during `egress_check`

P4: Compilation Process

TODO

Compiling Packet Parsers

- For devices with *programmable* parsers, the compiler generates the parser state machine (see PISA architecture)
- For devices with *fixed* parsers, the compiler verifies that the parser description is *consistent* with the device's fixed parser (e.g., ASICs)

Compiling Packet Parsers (Cont.)

Parser state table entries for the `vlan` and `mTag` sections of the parser

Current Version	Lookup Value	Next State
`vlan`	0xaaaa	`mTag`
`vlan`	0x800	`ipv4`
`vlan`	*	stop
`mTag`	0x800	`ipv4`
`mTag`	*	stop

The * is a wildcard that matches any value

The stop state indicates that the parser has finished processing the packet

Compiling Control Programs

The imperative control-flow representation does not call out dependencies between tables or opportunities for concurrency

1. The compiler analyzes the `control` program to determine dependencies between tables and opportunities for concurrency

2. The compiler generates the target configuration for the switch

Is this not familiar?

Two-stage compilation

Compiling Control Programs

The imperative control-flow representation does not call out dependencies between tables or opportunities for concurrency

1. The compiler analyzes the `control` program to determine dependencies between tables and opportunities for concurrency

2. The compiler generates the target configuration for the switch

Is this not familiar?

Two-stage compilation

Compiling Control Programs

The imperative control-flow representation does not call out dependencies between tables or opportunities for concurrency

1. The compiler analyzes the `control` program to determine dependencies between tables and opportunities for concurrency
2. The compiler generates the target configuration for the switch

Is this not familiar?

Two-stage compilation

Compiling Control Programs

The imperative control-flow representation does not call out dependencies between tables or opportunities for concurrency

1. The compiler analyzes the `control` program to determine dependencies between tables and opportunities for concurrency

2. The compiler generates the target configuration for the switch

Is this not familiar?

Two-stage compilation

Software Switches

TODO

Hardware Switches with RAM and TCAM

TODO

Switches supporting parallel tables

TODO

Switches that apply actions at the end of the pipeline

TODO

Switches with a few tables

TODO

Table (Addition)

```
table source_check {
  // Verify mtag only on ports to the core
  reads {
    mtag : valid; // Was mtag parsed?
    metadata.ingress_port : exact;
  }
  actions { // Each table entry specifies *one* action
    // 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; // One rule per port
}
```

Table (Addition)

```
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
}
```