Advanced Networked Systems SS24

Programmable Data Plane

Prof. Lin Wang, Ph.D.
Computer Networks Group
Paderborn University
https://cs.uni-paderborn.de/cn

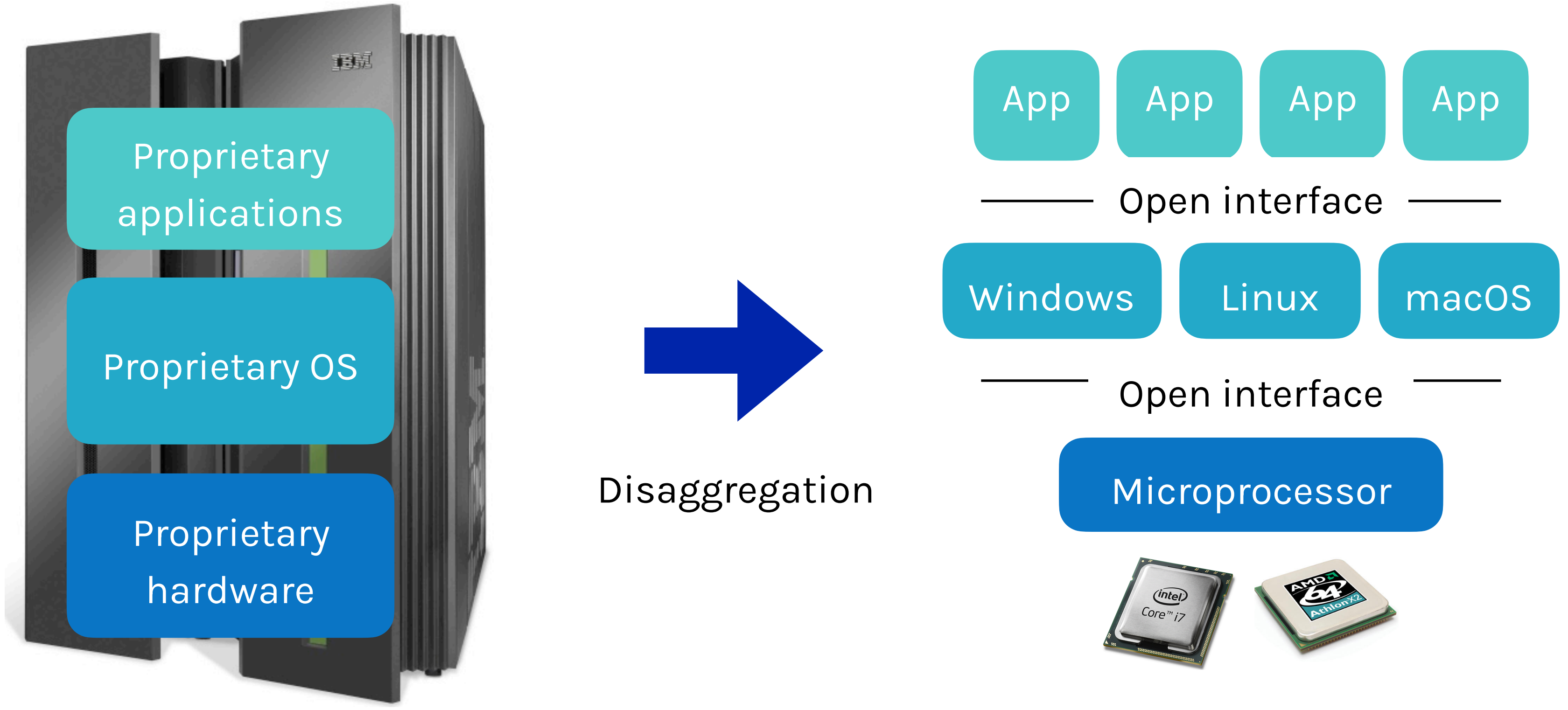
Learning objectives

Why we need programmable data plane?

How to enable data plane programmability?

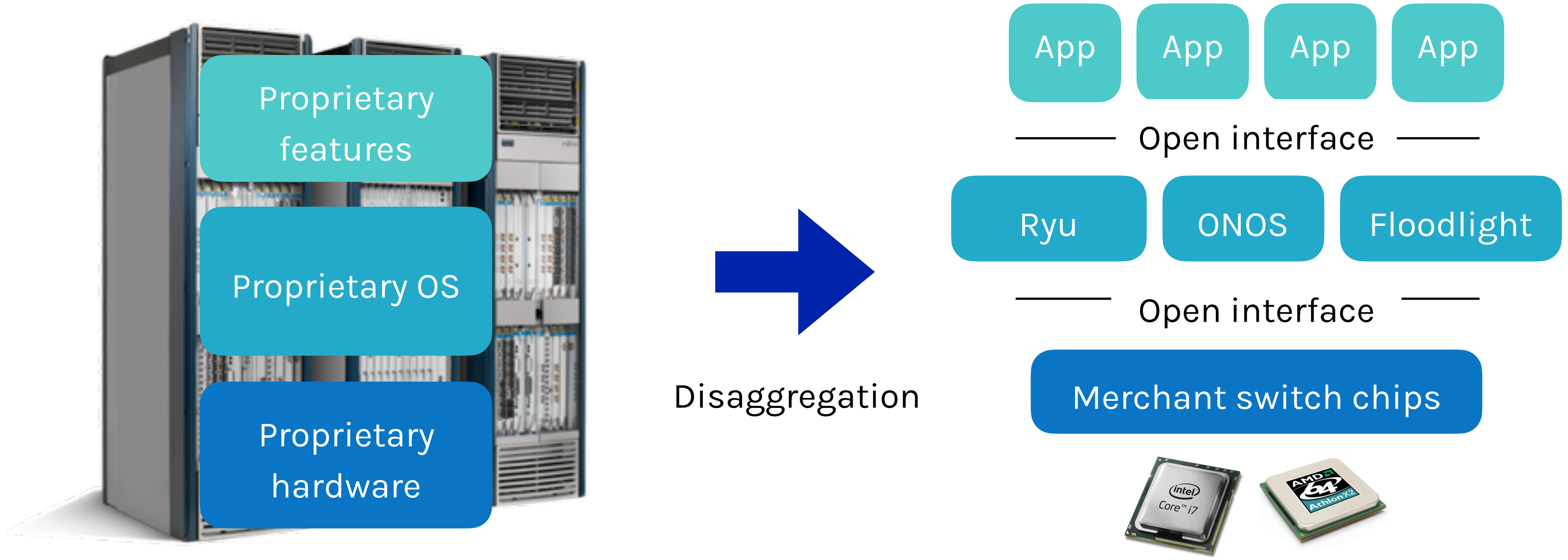
Why do we need data plane programmability?

Evolution of the computer industry



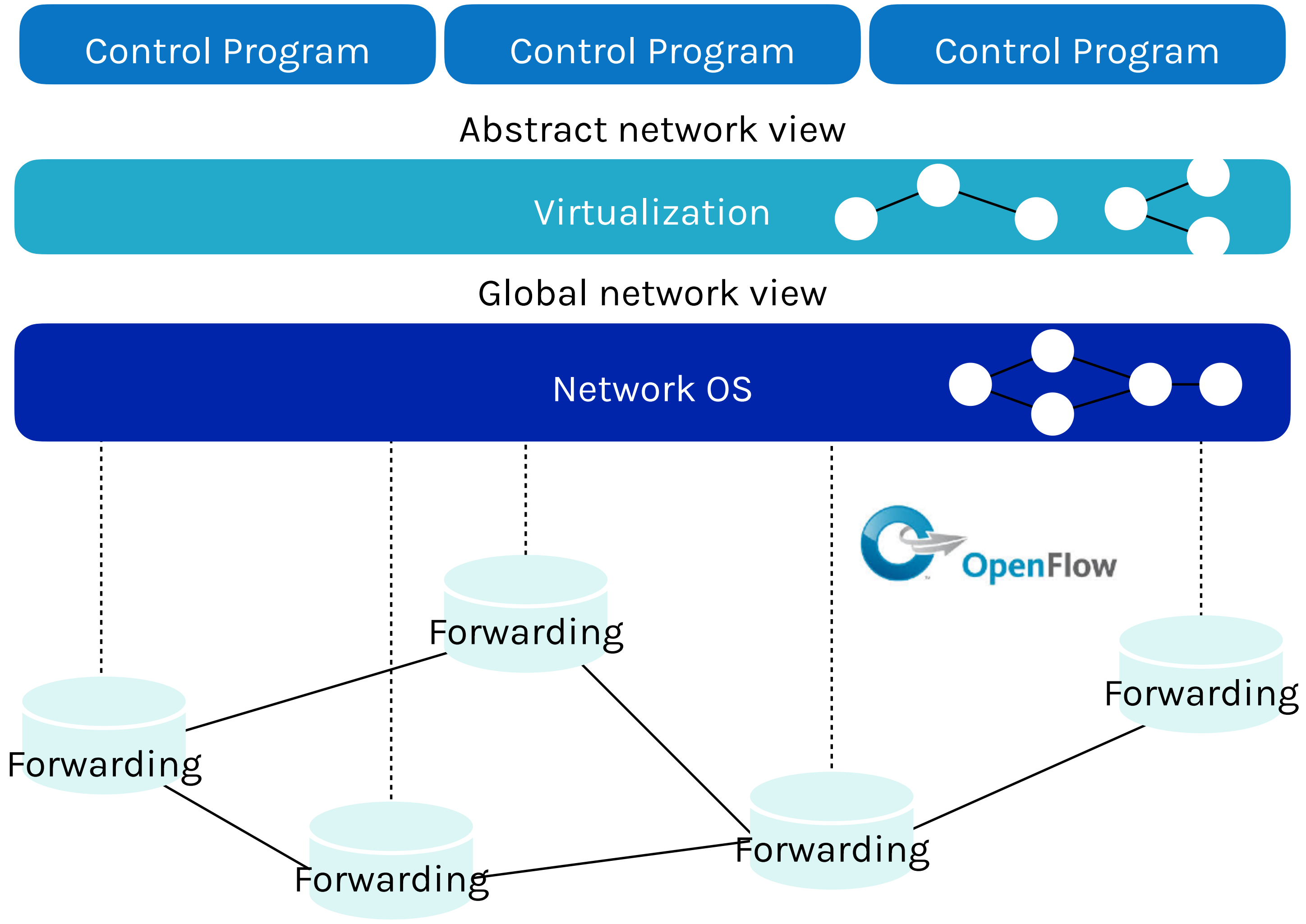
The computing industry has been evolving from proprietary hardware/software towards more **general-purpose** hardware/software with **open standards/interfaces**.

Evolution of networking industry

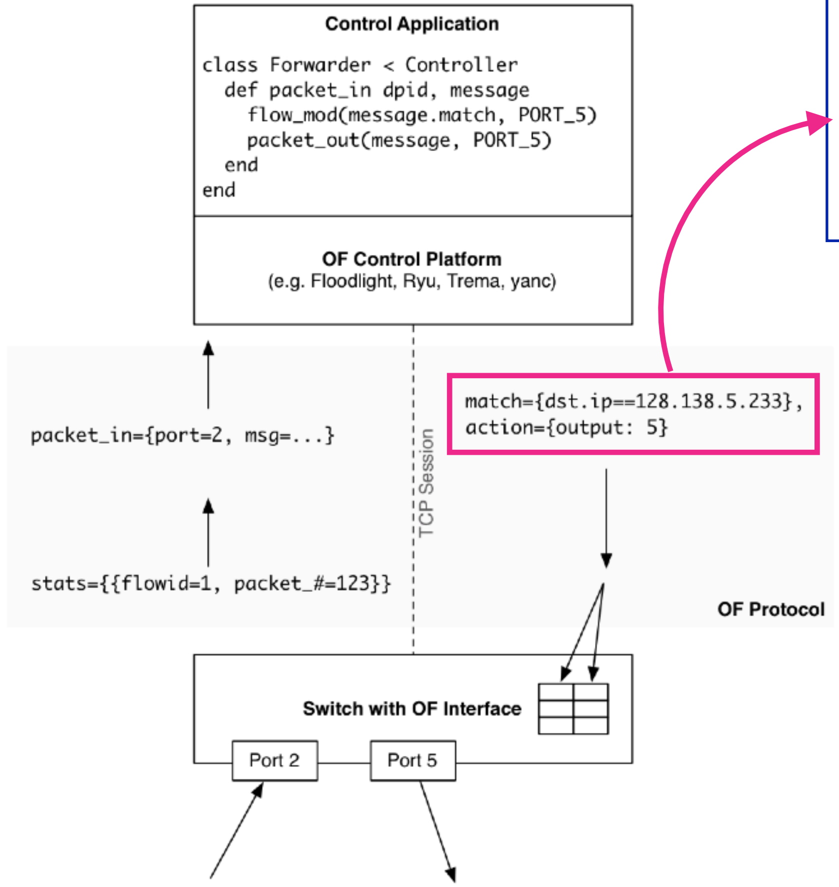


The networking industry has also been evolving from proprietary hardware/software towards more **general-purpose** hardware/software with **open standards/interfaces**.

Recap: software define networking



A deep dive into OpenFlow



```
class Forwarder < Controller
  def packet_in dpid, message
    flow_mod(message.match, PORT_5)
    packet_out(message, PORT_5)
  end
end
```

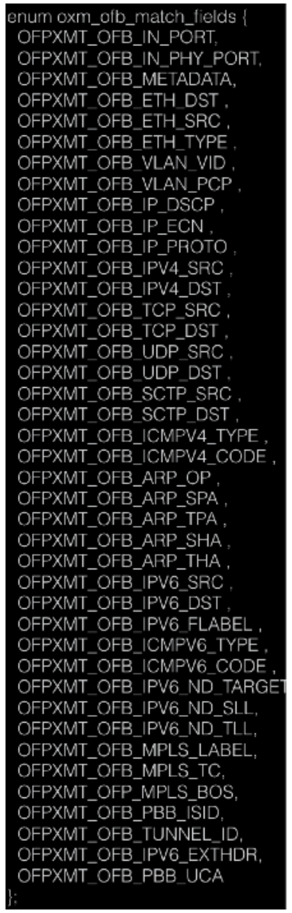
```
packet_in={port=2, msg=...}
stats={{flowid=1, packet_#=123}}
match={dst.ip==128.138.5.233},
action={output: 5}
```

OpenFlow is designed around the **match+action abstraction**: a set of header match fields and forwarding actions

OpenFlow v1.5: 41 match header fields

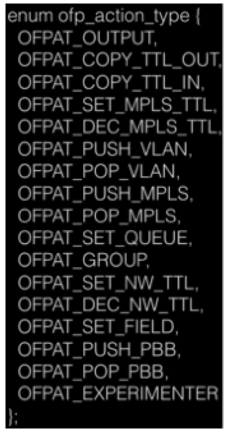
Most hardware/software switches only support limited match/action set (Ethernet, IP, TCP, MPLS) due to ASIC limitations.

Match



```
enum oxm_ofb_match_fields {
  OFPXMT_OFB_IN_PORT,
  OFPXMT_OFB_IN_PHY_PORT,
  OFPXMT_OFB_METADATA,
  OFPXMT_OFB_ETH_DST ,
  OFPXMT_OFB_ETH_SRC ,
  OFPXMT_OFB_ETH_TYPE ,
  OFPXMT_OFB_VLAN_VID ,
  OFPXMT_OFB_VLAN_PCP ,
  OFPXMT_OFB_IP_DSCP ,
  OFPXMT_OFB_IP_ECN ,
  OFPXMT_OFB_IP_PROTO ,
  OFPXMT_OFB_IPV4_SRC ,
  OFPXMT_OFB_IPV4_DST ,
  OFPXMT_OFB_TCP_SRC ,
  OFPXMT_OFB_TCP_DST ,
  OFPXMT_OFB_UDP_SRC ,
  OFPXMT_OFB_UDP_DST ,
  OFPXMT_OFB_SCTP_SRC ,
  OFPXMT_OFB_SCTP_DST ,
  OFPXMT_OFB_ICMPV4_TYPE ,
  OFPXMT_OFB_ICMPV4_CODE ,
  OFPXMT_OFB_ARP_OP ,
  OFPXMT_OFB_ARP_SPA ,
  OFPXMT_OFB_ARP_TPA ,
  OFPXMT_OFB_ARP_SHA ,
  OFPXMT_OFB_ARP_THA ,
  OFPXMT_OFB_IPV6_SRC ,
  OFPXMT_OFB_IPV6_DST ,
  OFPXMT_OFB_IPV6_FLABEL ,
  OFPXMT_OFB_ICMPV6_TYPE ,
  OFPXMT_OFB_ICMPV6_CODE ,
  OFPXMT_OFB_IPV6_ND_TARGET
  OFPXMT_OFB_IPV6_ND_SLL,
  OFPXMT_OFB_IPV6_ND_TLL,
  OFPXMT_OFB_MPLS_LABEL,
  OFPXMT_OFB_MPLS_TC,
  OFPXMT_OFP_MPLS_BOS,
  OFPXMT_OFB_PBB_ISID,
  OFPXMT_OFB_TUNNEL_ID,
  OFPXMT_OFB_IPV6_EXTHDR,
  OFPXMT_OFB_PBB_UCA
};
```

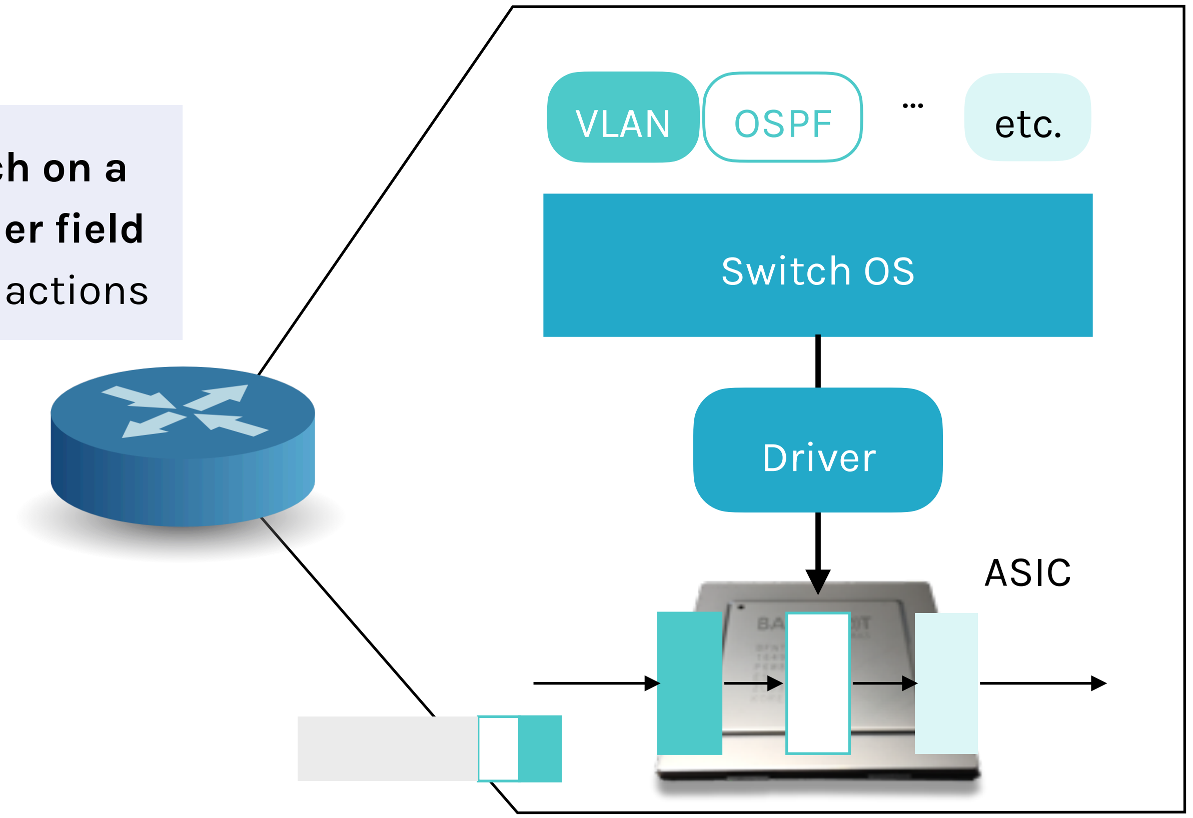
Action



```
enum ofp_action_type {
  OFPAT_OUTPUT,
  OFPAT_COPY_TTL_OUT,
  OFPAT_COPY_TTL_IN,
  OFPAT_SET_MPLS_TTL,
  OFPAT_DEC_MPLS_TTL,
  OFPAT_PUSH_VLAN,
  OFPAT_POP_VLAN,
  OFPAT_PUSH_MPLS,
  OFPAT_POP_MPLS,
  OFPAT_SET_QUEUE,
  OFPAT_GROUP,
  OFPAT_SET_NW_TTL,
  OFPAT_DEC_NW_TTL,
  OFPAT_SET_FIELD,
  OFPAT_PUSH_PBB,
  OFPAT_POP_PBB,
  OFPAT_EXPERIMENTER
};
```

Switch architecture

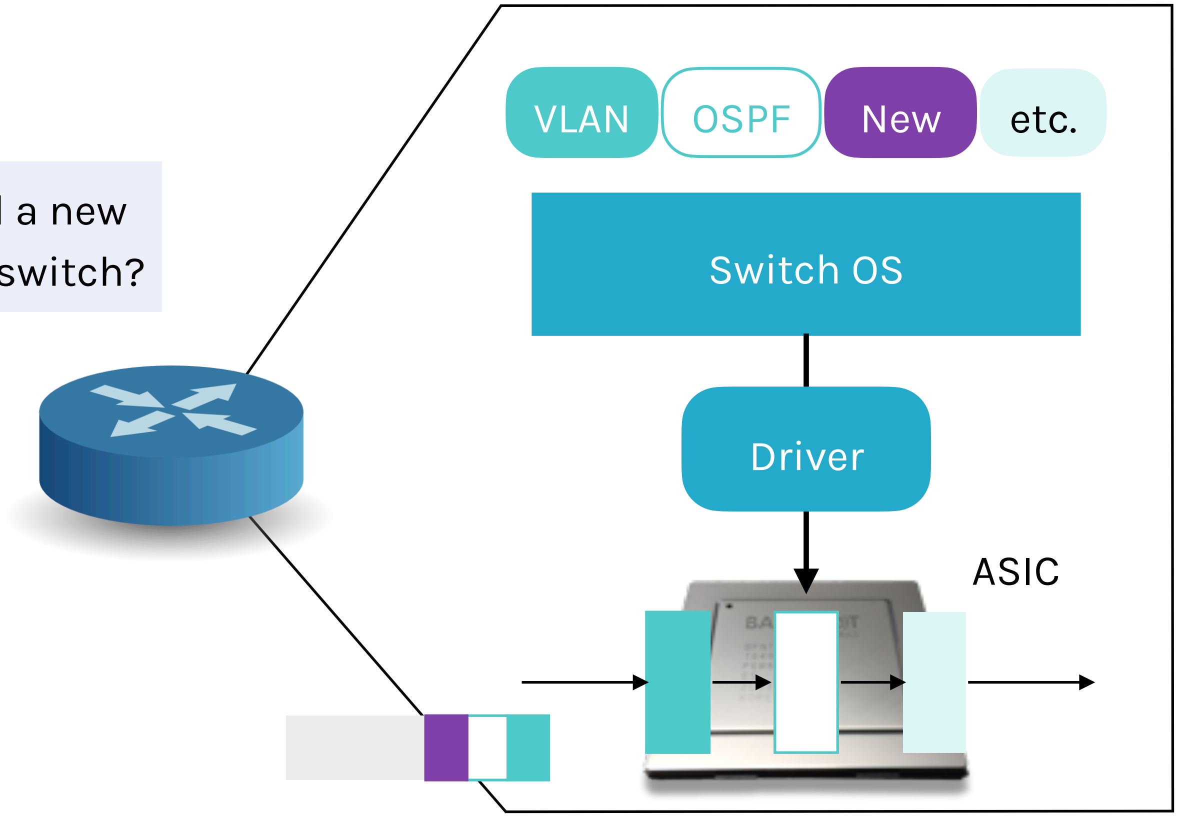
A switch can only **match on a supported packet header field** and take corresponding actions



Packet processing pipeline

Switch architecture

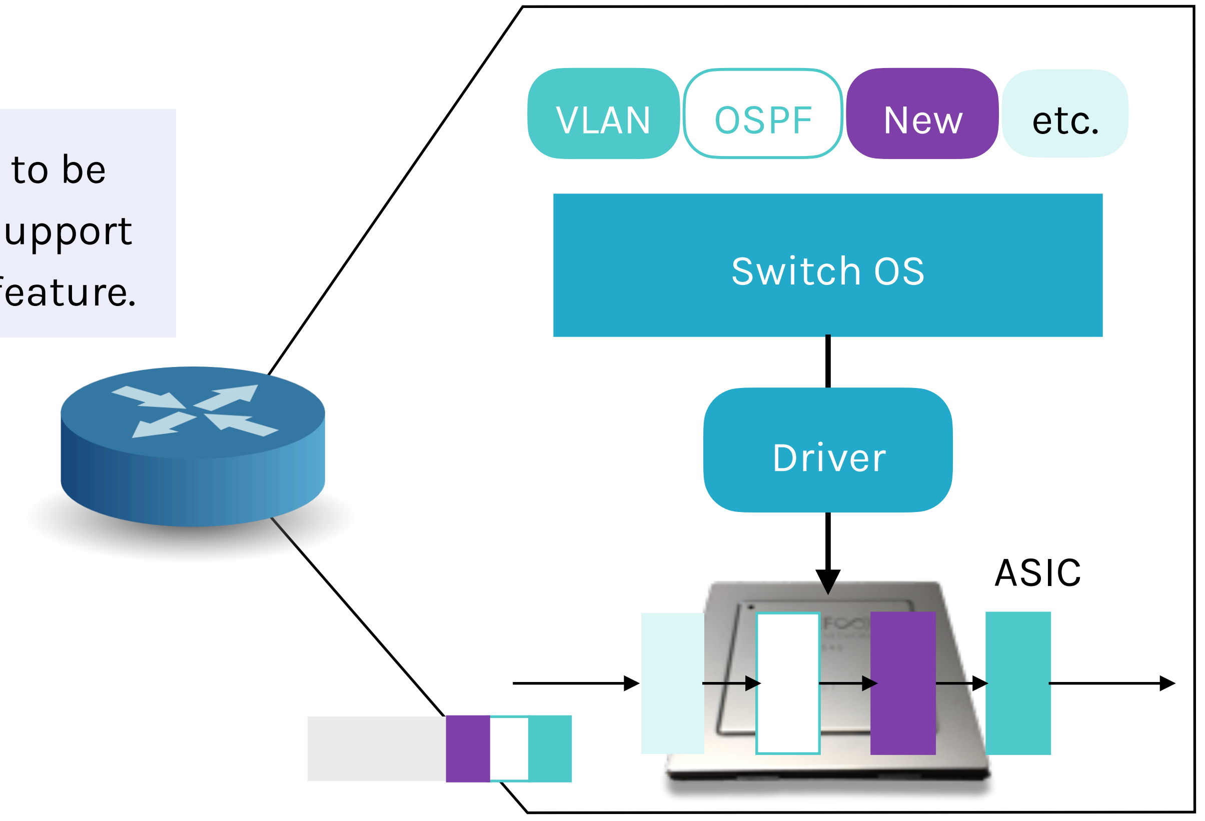
What if we want to add a new protocol/feature to the switch?



Packet processing pipeline

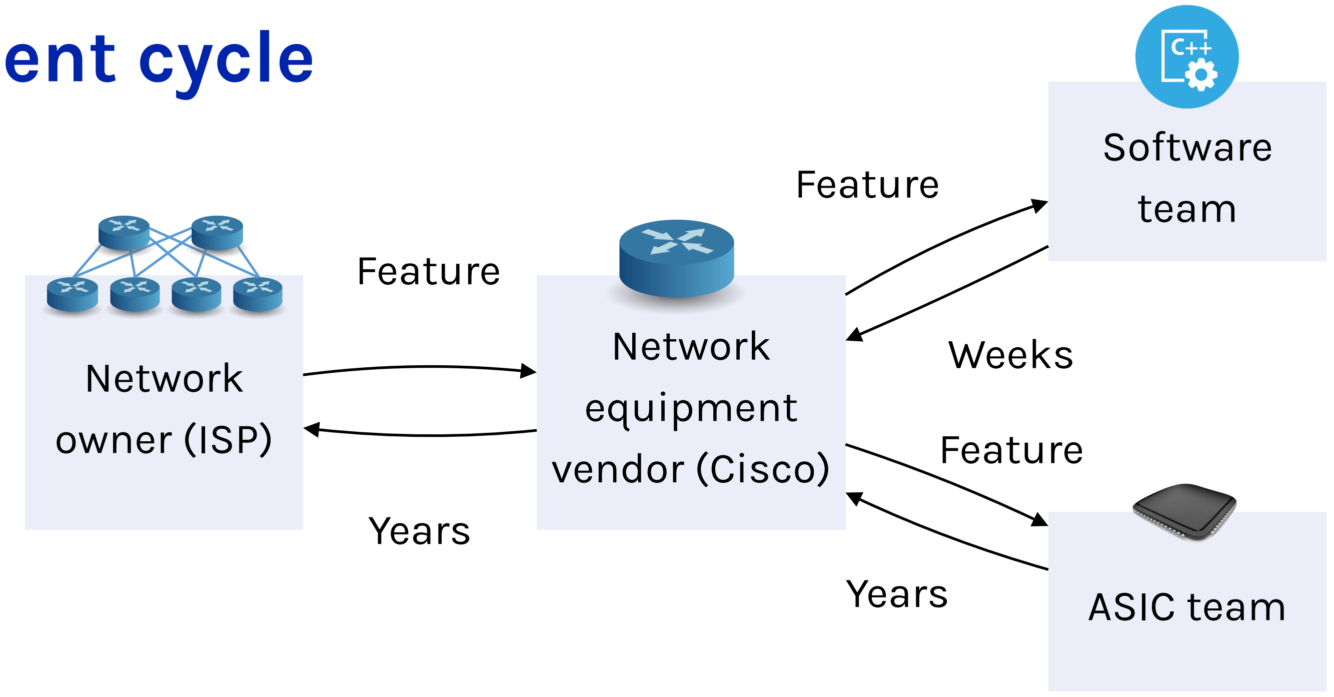
Switch architecture

The switch ASIC has to be modified in order to support such a new protocol/feature.



Packet processing pipeline

Development cycle



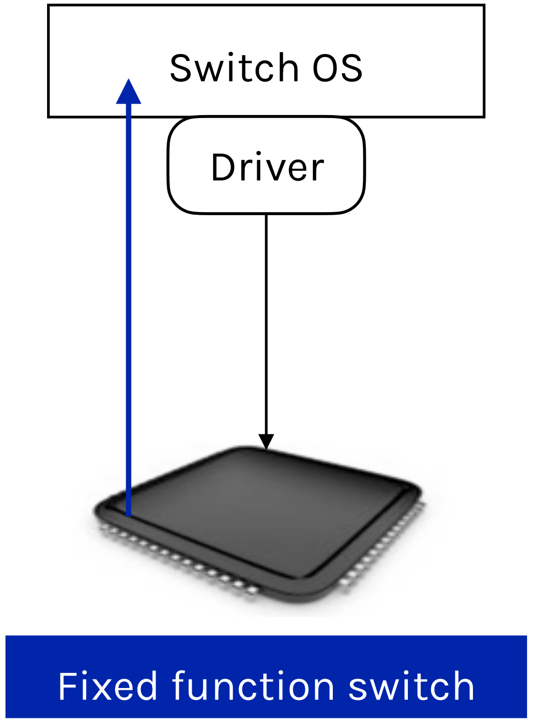
It takes years for the new ASIC to be developed, fully tested, and finally deployed!! When the upgrade is available:

- It either **no longer solves your problem**
- You need **a fork-lift upgrade** at huge expenses

What is the root cause of all this?

The “bottom-up” mentality

“This is how I process packet…”

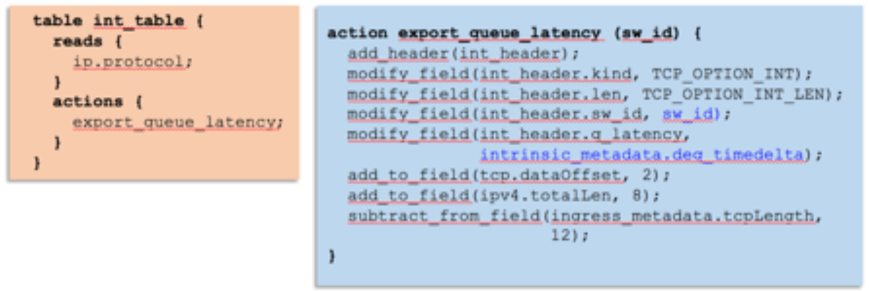


The network systems are built following the bottom-up approach: all network features are centered around **the capabilities of the ASIC**.

How to improve this?

The “top-down” approach

Make the ASIC **programmable**, and let your features to tell the ASIC what to support!

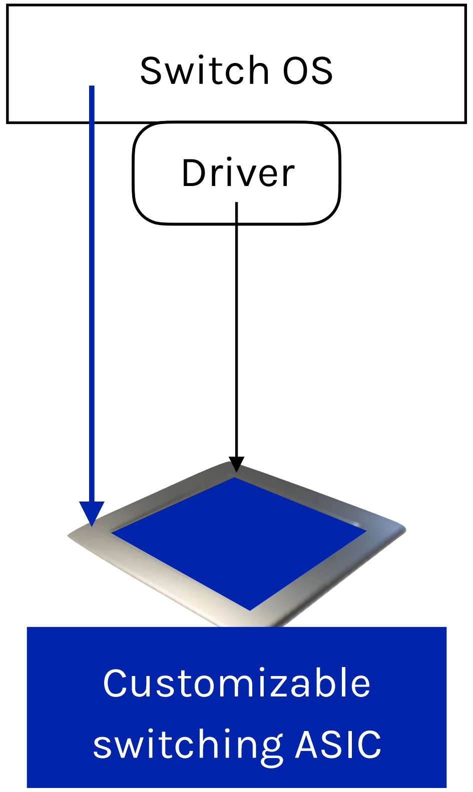


```
table int_table {
  reads {
    ip.protocol;
  }
  actions {
    export_queue_latency;
  }
}
```

```
action export_queue_latency (sw_id) {
  add_header(int_header);
  modify_field(int_header.kind, TCP_OPTION_INT);
  modify_field(int_header.len, TCP_OPTION_INT_LEN);
  modify_field(int_header.sw_id, sw_id);
  modify_field(int_header.q_latency,
               intrinsic_metadata.deq_timedelta);
  add_to_field(tcp.dataOffset, 2);
  add_to_field(ipv4.totalLen, 8);
  subtract_from_field(ingress_metadata.tcpLength,
                      12);
}
```

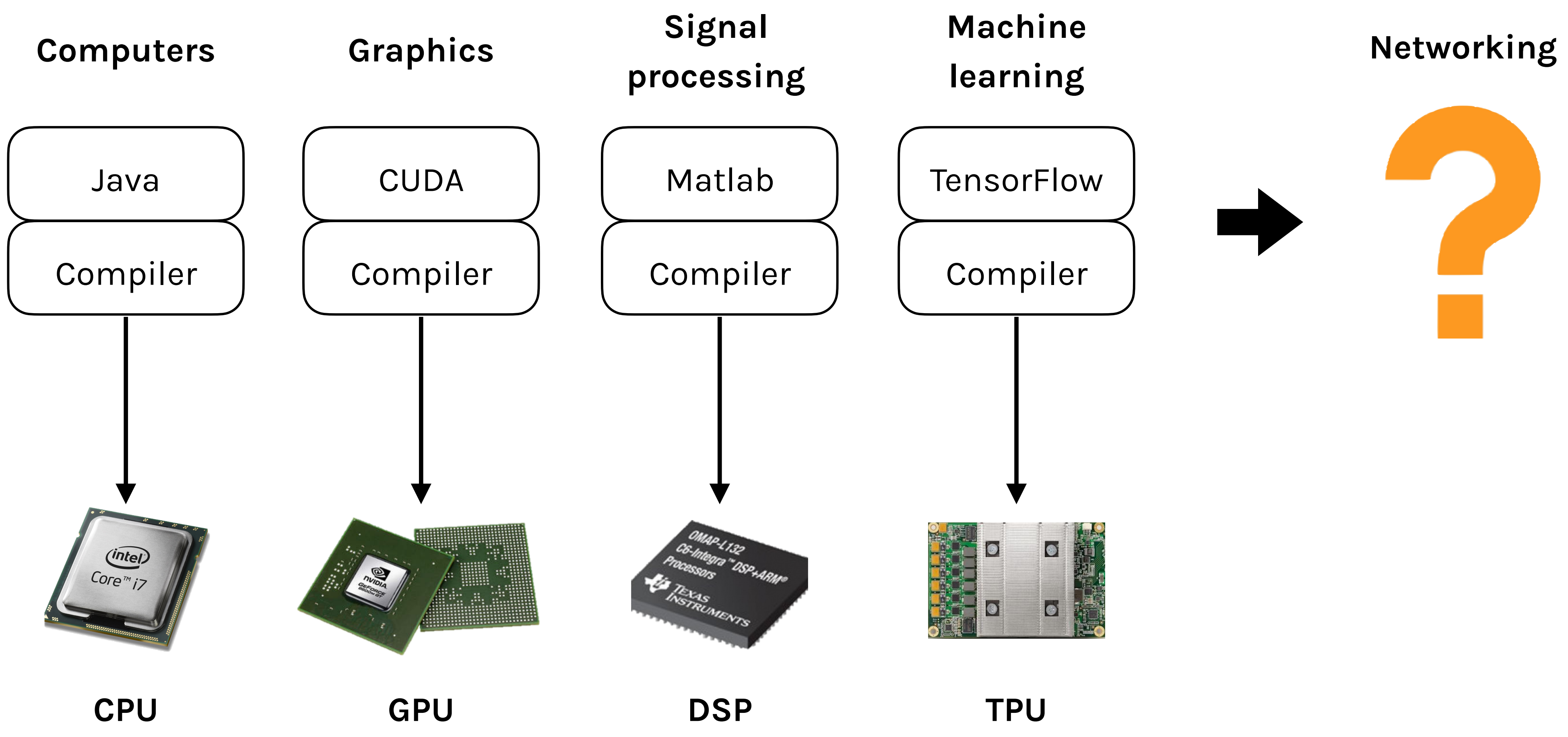
“This is precisely how you must process packets…”

How to support programmability?

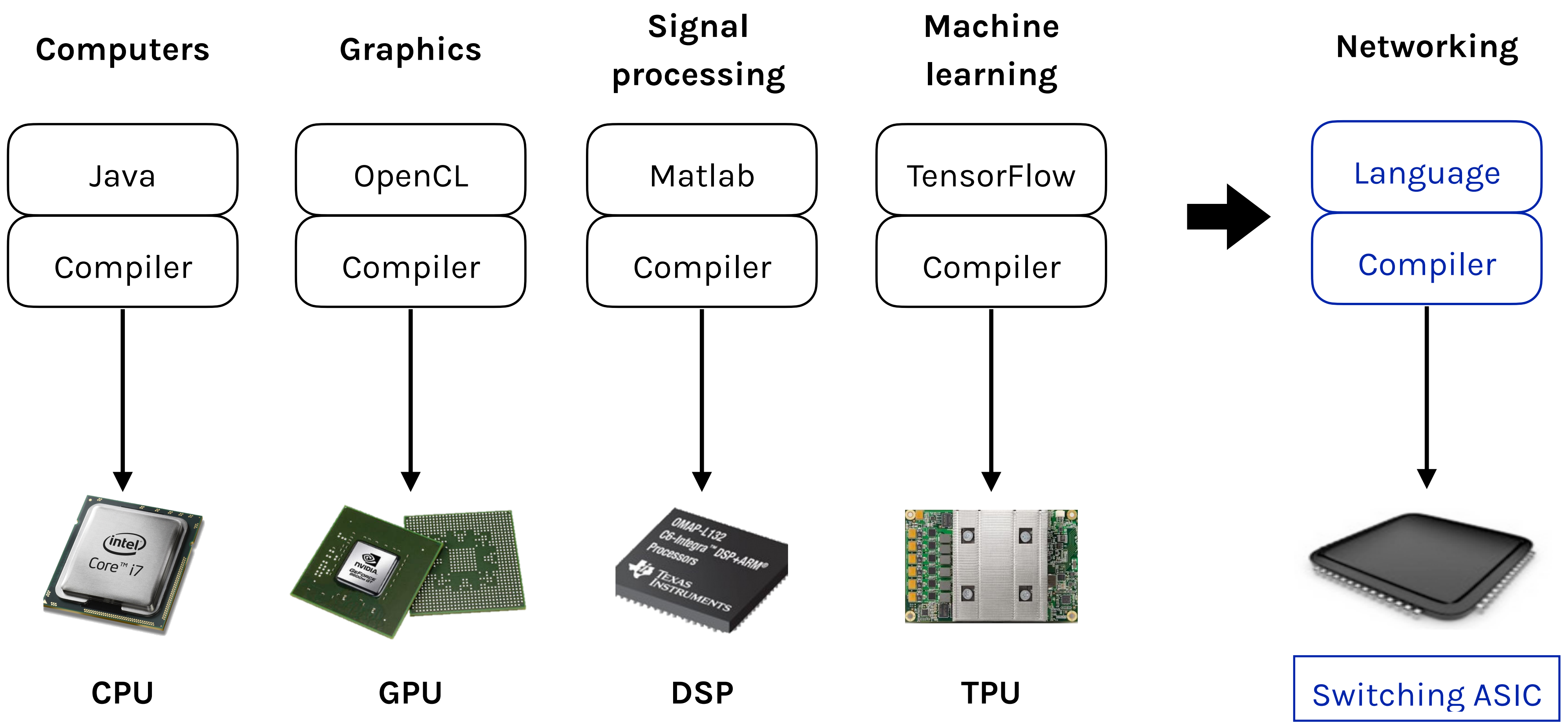


How to enable data plane programmability?

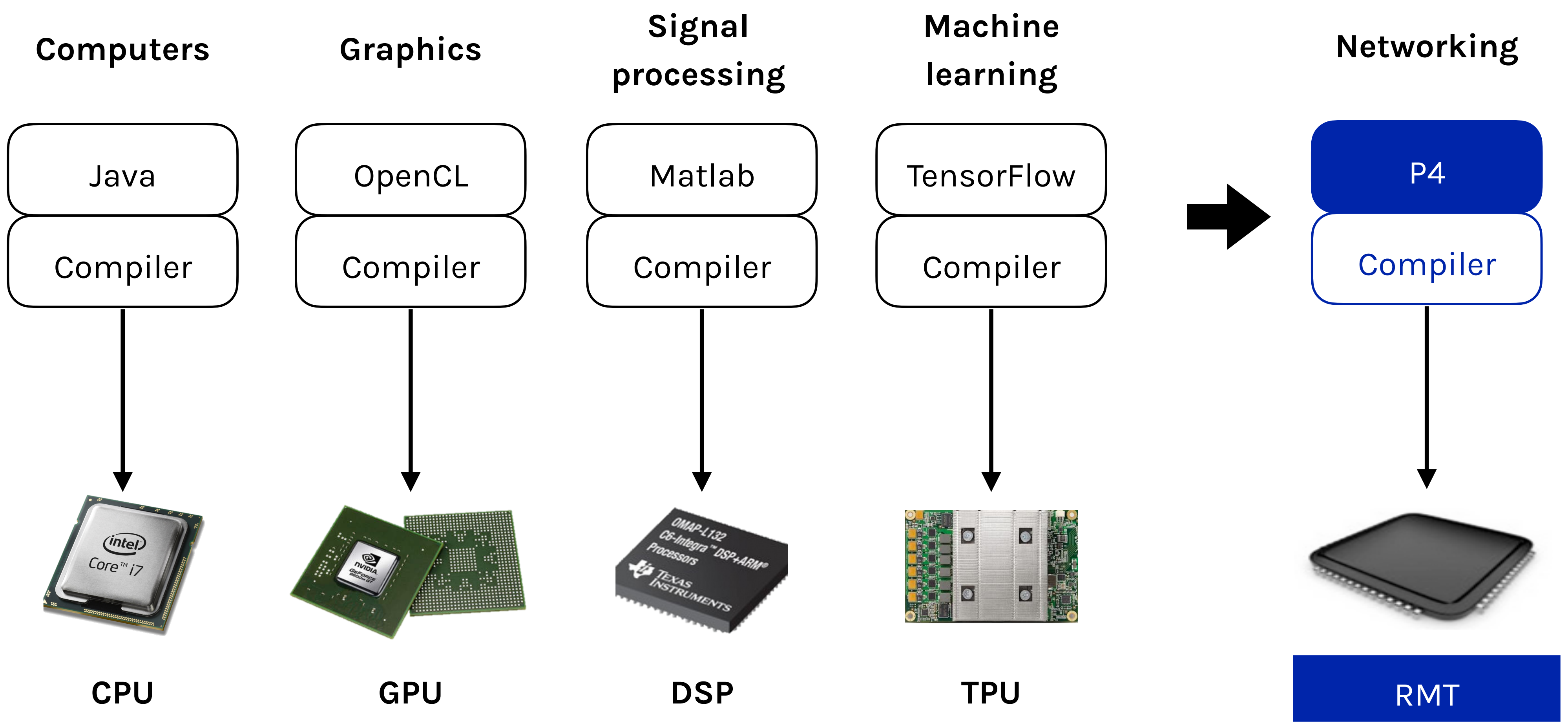
Domain-specific processors



Domain-specific processors



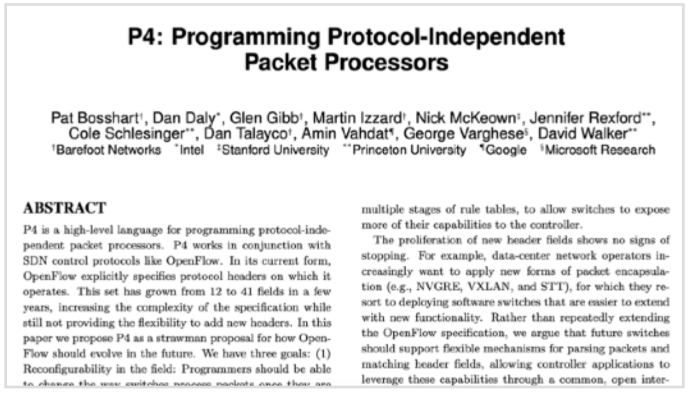
Domain-specific processors



RMT and P4

RMT: reconfigurable match tables model (a RISC-inspired pipelined architecture)

P4: a domain-specific language for programming protocol-independent packet processors



P4: Programming Protocol-Independent Packet Processors

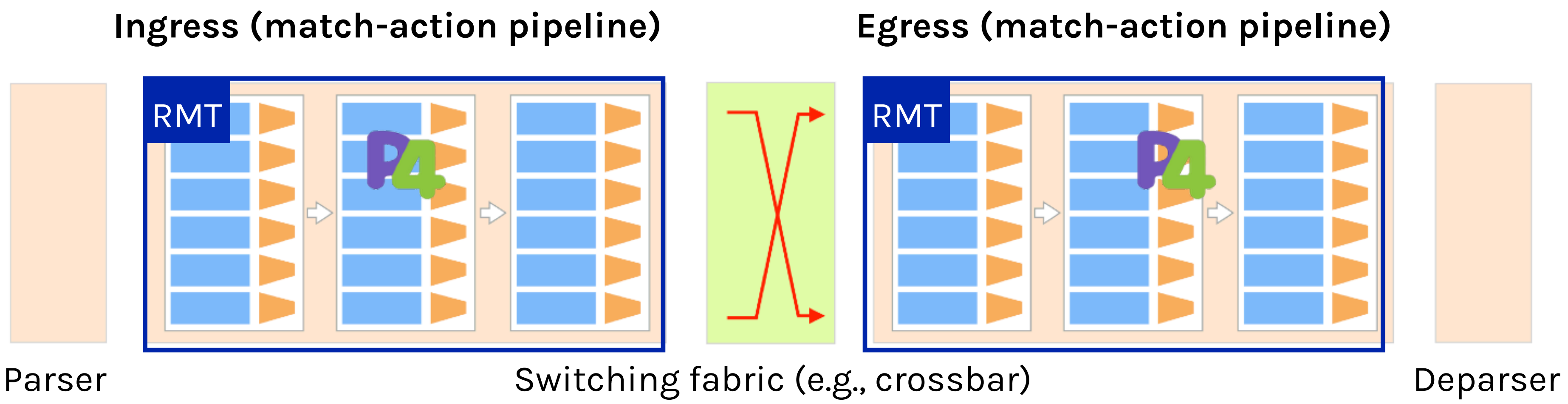
Pat Bosshart[†], Dan Daly[*], Glen Gibb[†], Martin Izzard[†], Nick McKeown[‡], Jennifer Rexford[**], Cole Schlesinger[**], Dan Talayco[†], Amin Vahdat[¶], George Varghese[§], David Walker[**]
[†]Barefoot Networks [*]Intel [‡]Stanford University [**]Princeton University [¶]Google [§]Microsoft Research

ABSTRACT

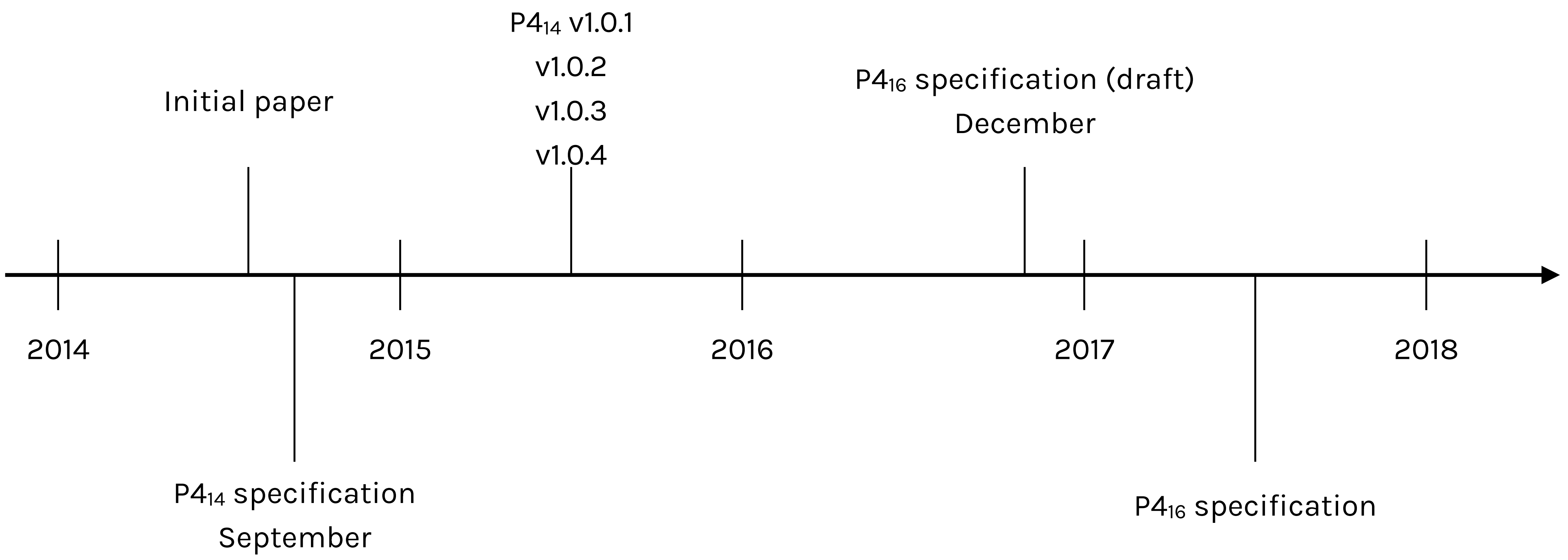
P4 is a high-level language for programming protocol-independent packet processors. P4 works in conjunction with SDN control protocols like OpenFlow. In its current form, OpenFlow explicitly specifies protocol headers on which it operates. This set has grown from 12 to 41 fields in a few years, increasing 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

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 inter-


Ingress (match-action pipeline)
Egress (match-action pipeline)
RMT
RMT
P4
P4
Parser
Switching fabric (e.g., crossbar)
Deparser

P4 development



$P4_{16}$ introduces the concept of architecture

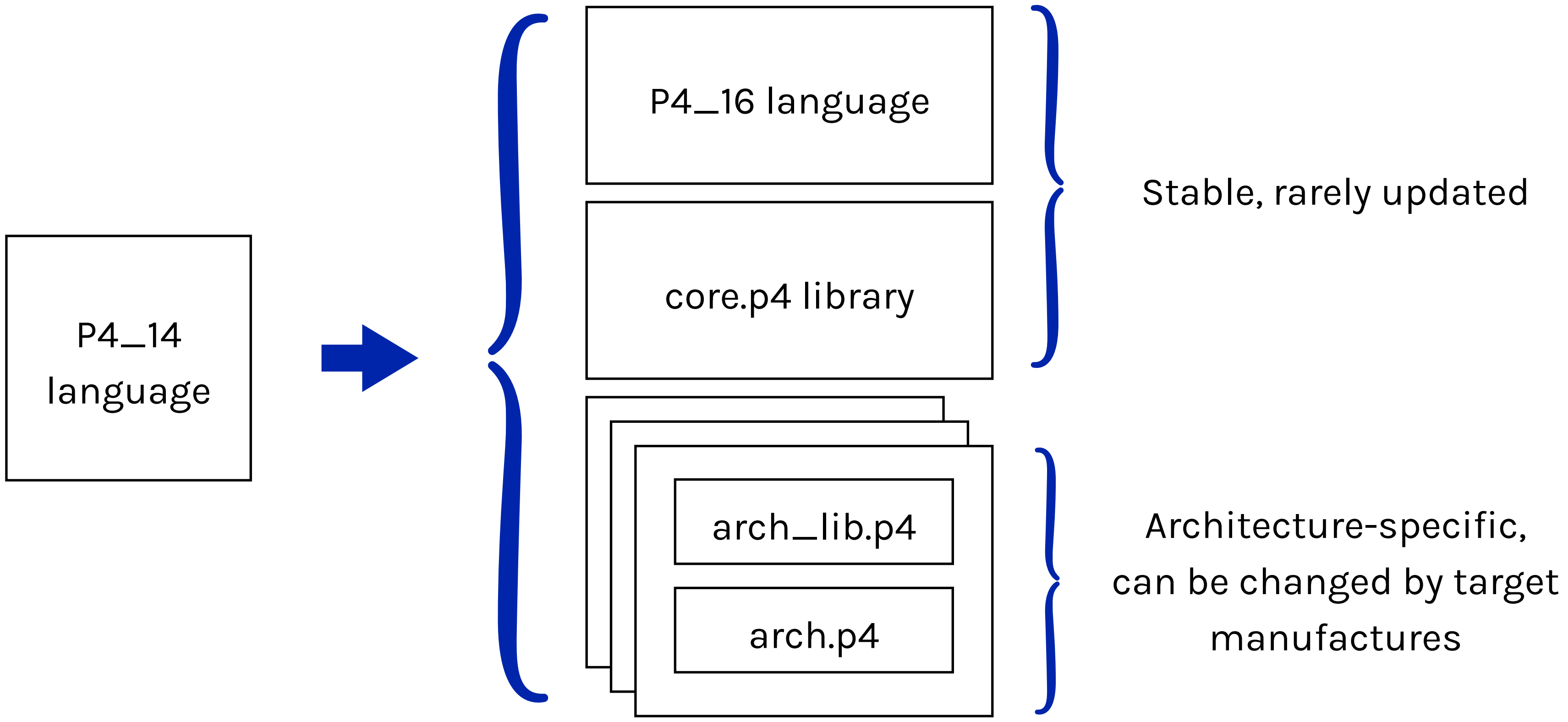
P4 architecture

Specifies the **P4 programmable components** of a target and **data plane interfaces** between them

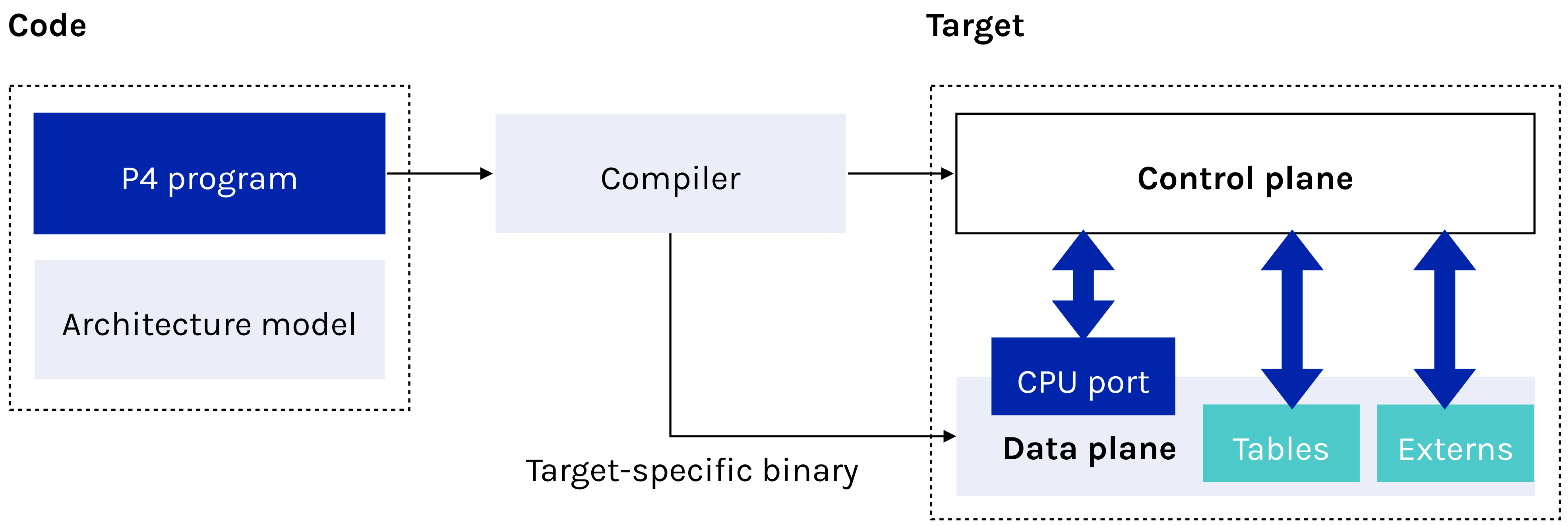
P4 target

A model of a specific hardware implementation

P4 language evolvement

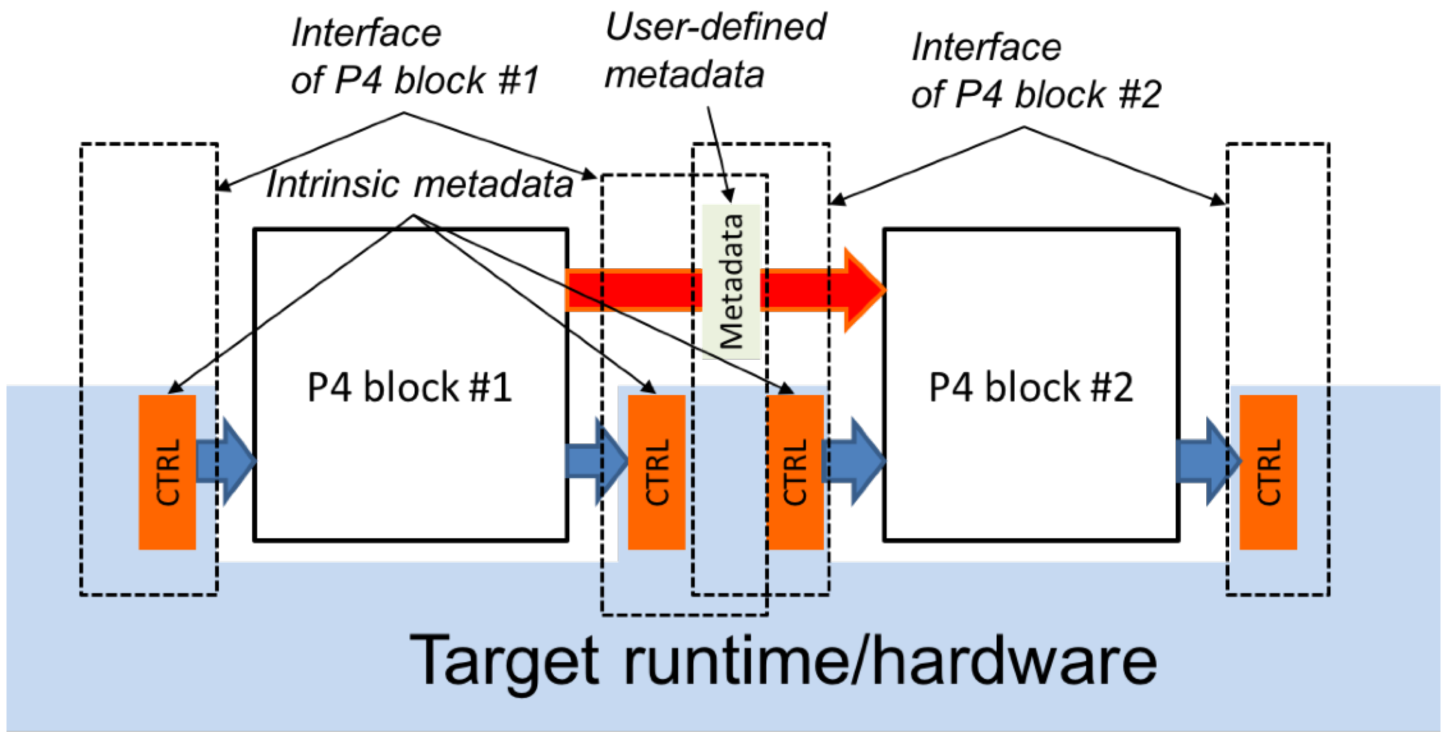


Programming a P4 target



■	User supplied
□	Vendor supplied

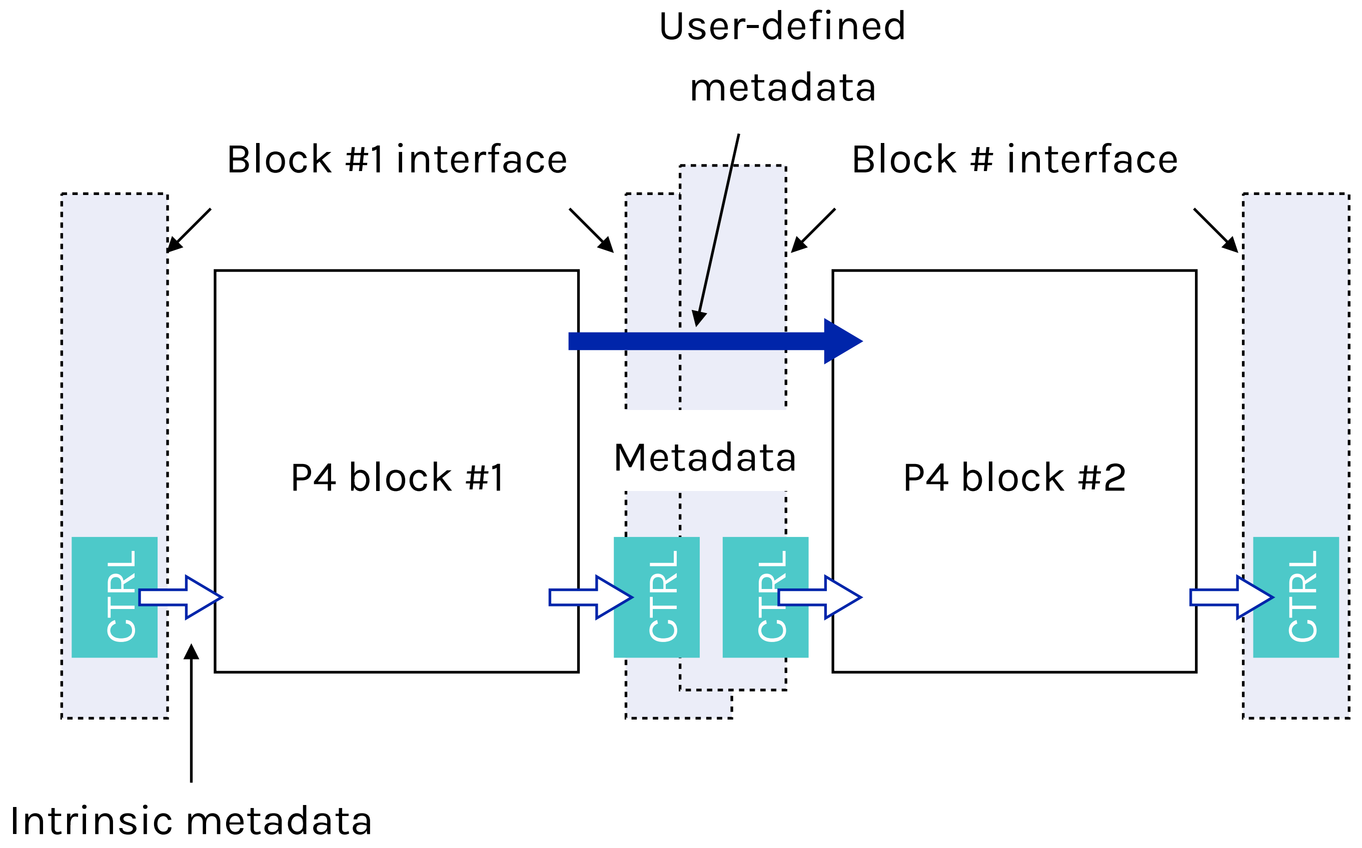
Architecture model



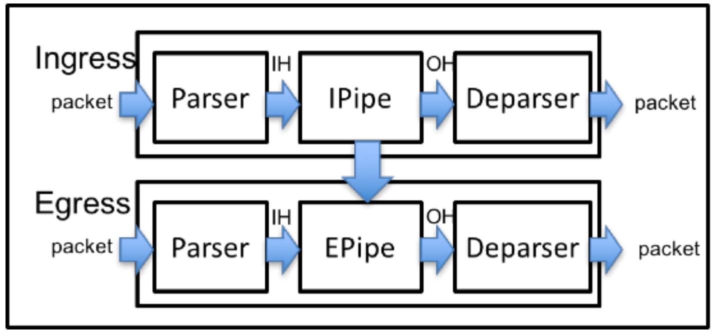
A contract between the P4 program and the target

Architecture model

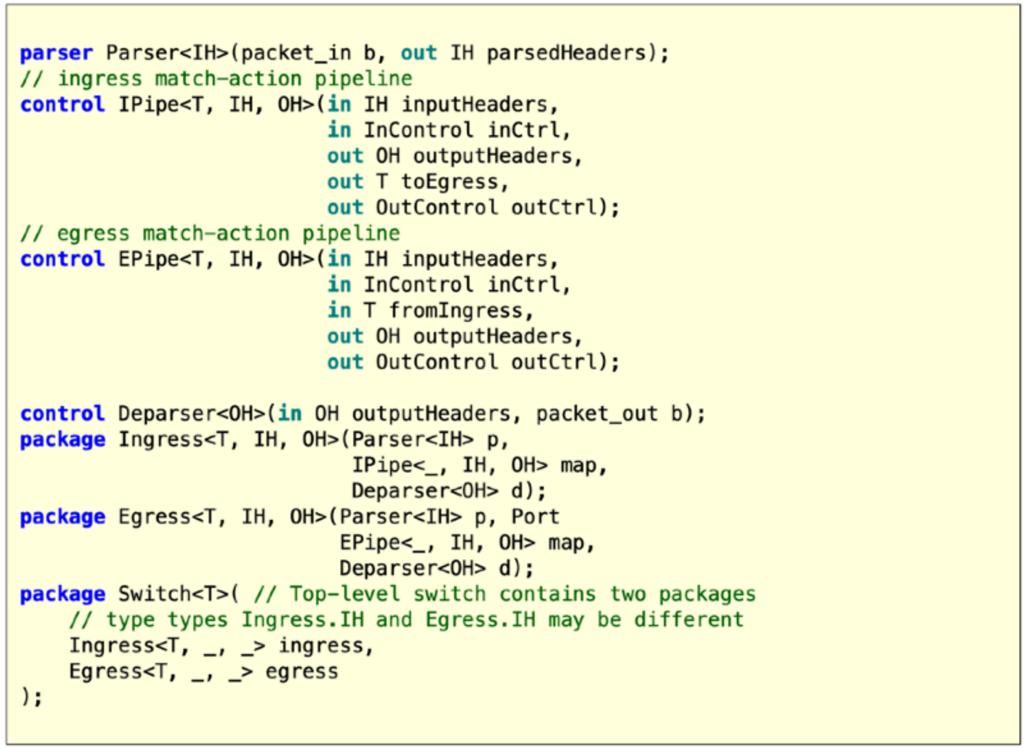
A contract between the P4 program and the target



Switch architecture example



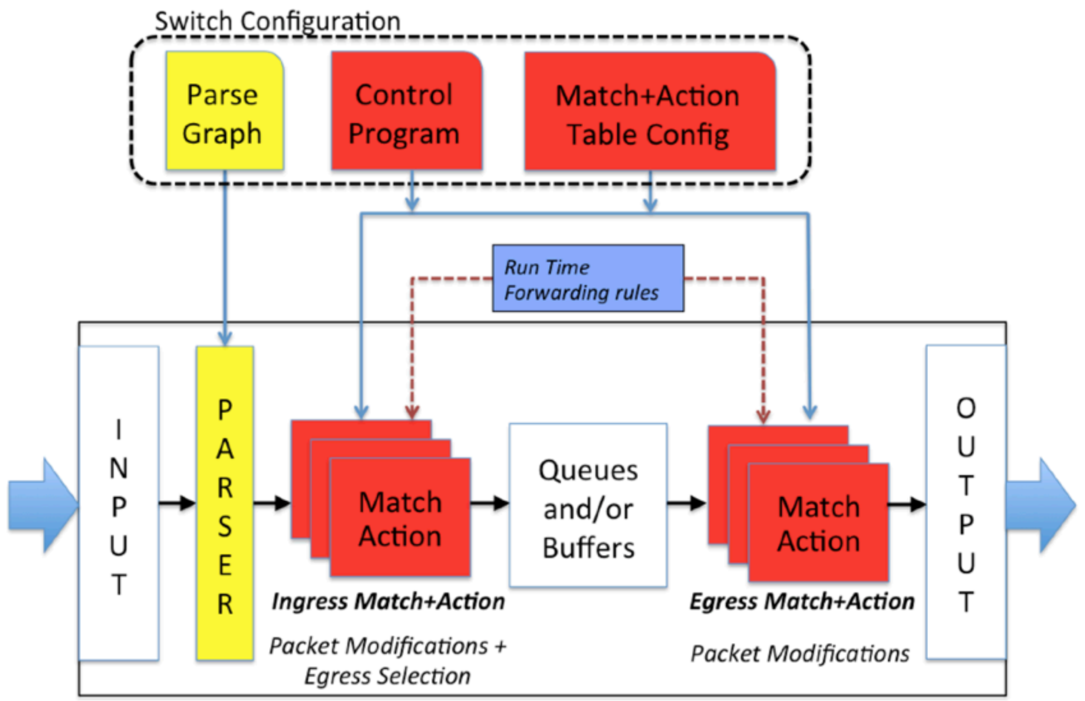
Switch architecture


```
parser Parser<IH>(packet_in b, out IH parsedHeaders);
// ingress match-action pipeline
control IPipe<T, IH, OH>(in IH inputHeaders,
                         in InControl inCtrl,
                         out OH outputHeaders,
                         out T toEgress,
                         out OutControl outCtrl);
// egress match-action pipeline
control EPipe<T, IH, OH>(in IH inputHeaders,
                         in InControl inCtrl,
                         in T fromIngress,
                         out OH outputHeaders,
                         out OutControl outCtrl);

control Deparser<OH>(in OH outputHeaders, packet_out b);
package Ingress<T, IH, OH>(Parser<IH> p,
                           IPipe<_, IH, OH> map,
                           Deparser<OH> d);
package Egress<T, IH, OH>(Parser<IH> p, Port
                          EPipe<_, IH, OH> map,
                          Deparser<OH> d);
package Switch<T>( // Top-level switch contains two packages
    // type types Ingress.IH and Egress.IH may be different
    Ingress<T, _, _> ingress,
    Egress<T, _, _> egress
);
```

Switch architecture description

A simple $P4_{16}$ switch architecture: v1model

Roughly equivalent to Protocol-Independent Switch Architecture (PISA)



https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf

v1model architecture

Defines the metadata it supports, including both intrinsic and user-defined ones

```
struct standard_metadata_t {
    bit<9> ingress_port;
    bit<9> egress_spec;
    bit<9> egress_port;
    bit<32> clone_spec;
    bit<32> instance_type;
    bit<1> drop;
    bit<16> recirculate_port;
    bit<32> packet_length;
    bit<32> enq_timestamp;
    bit<19> enq_qdepth;
    bit<32> deq_timedelta;
    bit<19> deq_qdepth;
    error parser_error;
```

```
    bit<48> ingress_global_timestamp;
    bit<48> egress_global_timestamp;
    bit<32> lf_field_list;
    bit<16> mcast_grp;
    bit<32> resubmit_flag;
    bit<16> egress_rid;
    bit<1> checksum_error;
    bit<32> recirculate_flag;
}
```

Standard intrinsic metadata

https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4

Architecture-specific constructs

Each architecture defines a list of “externs”

- Blackbox functions whose interfaces are known

Most targets contain specialized components, which cannot be expressed in P4

On the other hand, $P4_{16}$ aims to be target-independent

- $P4_{14}$ has almost 1/3 of the constructs target-dependent: not portable to different targets

```
extern register<T> {
   register(bit<32> size);
   void read(out T result, in bit<32> index);
   void write(in bit<32> index, in T value);
}
extern void random<T>(out T result, in T lo, in T hi);
extern void hash<O, T, D, M>(out O result,
   in HashAlgorithm algo, in T base, in D data, in M max);
extern void update_checksum<T, O>(in bool condition,
   in T data, inout O checksum, HashAlgorithm algo);
```

v1model architecture-specific externs

P4 language basics

P4 language overview

```
#include <core.p4>
#include <v1model.p4>
```

Libraries

```
const bit<16> TYPE_IPV4 = 0x800;
typedef bit<32> ip4Addr_t;
header ipv4_t {...}
struct headers {...}
```

Declarations

```
parser MyParser(...) {
   state start {...}
   state parse_ethernet {...}
   state parse_ipv4 {...}
}
```

Packet header parser

```
control MyIngress(...) {
   action ipv4_forward(...) {...}
   table ipv4_lpm {...}
   apply {
      if (...) {...}
   }
}
```

Control flow to modify packets

```
control MyDeparser(...) {...}
```

Assemble modified packet

```
V1Switch(
   MyParser(),
   MyVerifyChecksum(),
   MyIngress(),
   MyEgress(),
   MyComputeChecksum(),
   MyDeparser()
) main;
```

“main()”

P4 language basics: data types

P4_16 is a statically-typed language with base types and operators to derive composed ones

`bool`	Boolean value
`bit<W>`	Bit-string of width W
`int<W>`	Signed integer of width W
`varbit<W>`	Bit-string of dynamic length <= W
`match_kind`	Describes ways to match table keys
`error`	Used to signal errors
`void`	No values, used in few restricted circumstances
✗ `float`	Not supported
✗ `string`	Not supported

P4 language basics: composed data types

Header

```
header Ethernet_h {
  bit<48> dstAddr;
  bit<48> srcAddr;
  bit<16> etherType;
}
```

Header stack

```
header Mpls_h {
  bit<20> label;
  bit<3> tc;
  bit bos;
  bit<8> ttl;
}
Mpls_h[10] mpls;
```

Array of up to 10 MPLS headers

Header union

```
header_union Ip_h {
  IPv4_h v4;
  IPv6_h v6;
}
```

Either IPv4 or IPv6 header is present

A successful `extract()` sets to true the validity bit of the extracted header `hdr.ipv4.isValid()`

Parsing a packet using `extract()` fills in the fields of the header from a network packet

P4 language basics: composed data types

Struct: unordered collection of **named members**

```
struct standard_metadata_t {
  bit<9> ingress_port;
  bit<9> egress_spec;
  bit<9> egress_port;
  …
}
```

Tuple: ordered collection of **unnamed members**

```
tuple<bit<32>, bool> x;
x = {10, false}
```

Other data types:

- enum: `enum Priority {High, Low}`
- Type specification: `typedef bit<48> macAddr_t;`
- extern, parser, control, package…

P4 language basics: operations

P4 operations are similar to C operations and vary depending on the types (unsigned/ signed integers,…)

- Arithmetic operations: +, -, *
- Logical operations:
  - Bitwise complement, and, or, xor: `~`, `&`, `|`, `^`
  - Shifts: >>, <<
- Non-standard operations: `[m:l]` bit slicing, ++ bit concatenation
- **No division and modulo:** can be approximated

P4 language basics: variables and constants

Constants, variable declarations and instantiations are almost the same as in C too

Variable

```
bit<8> x = 123;

typedef bit<8> MyType;
MyType x;
x = 123;
```

Constant

```
const bit<8> x = 123;

typedef bit<8> MyType;
const MyType x = 123;
```

Important

Variables cannot be used to maintain state across different network packets.

Instead, we can only use **two stateful constructs, i.e., tables and extern objects,** to maintain state.

P4 language basics: statements

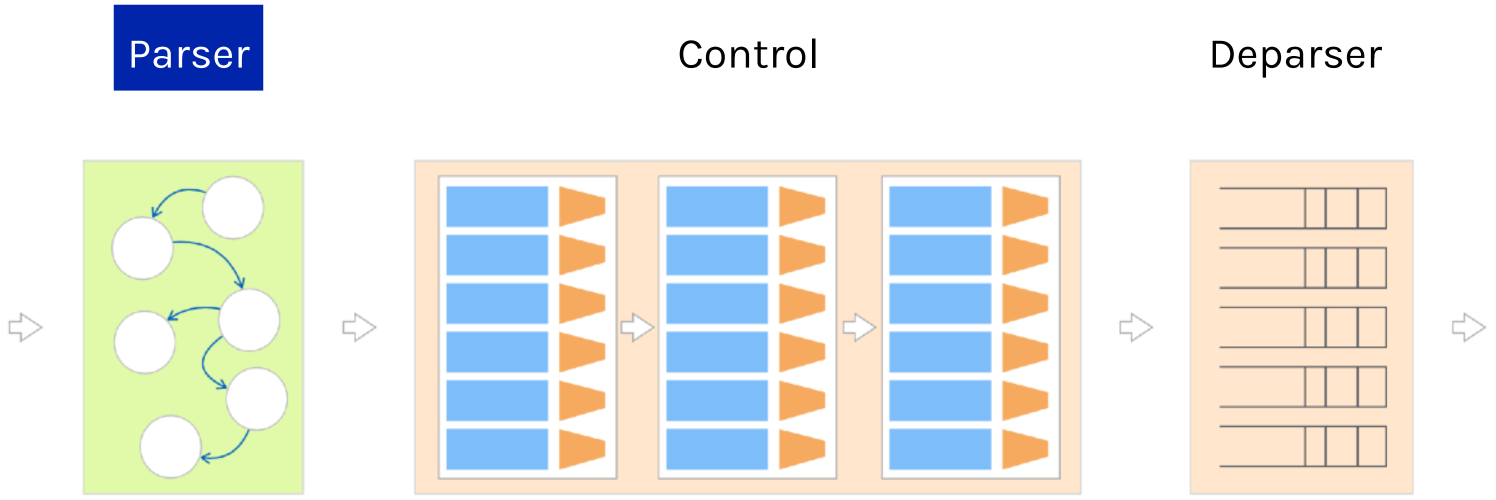
P4 statements are pretty classical too

- Some restrictions may apply depending on the statement location

`return`	Terminates the execution of the action of control containing it	
`exit`	Terminates the execution of all the blocks currently executing	
Conditions	`if (x==123) {…} else {…}`	Not in parser

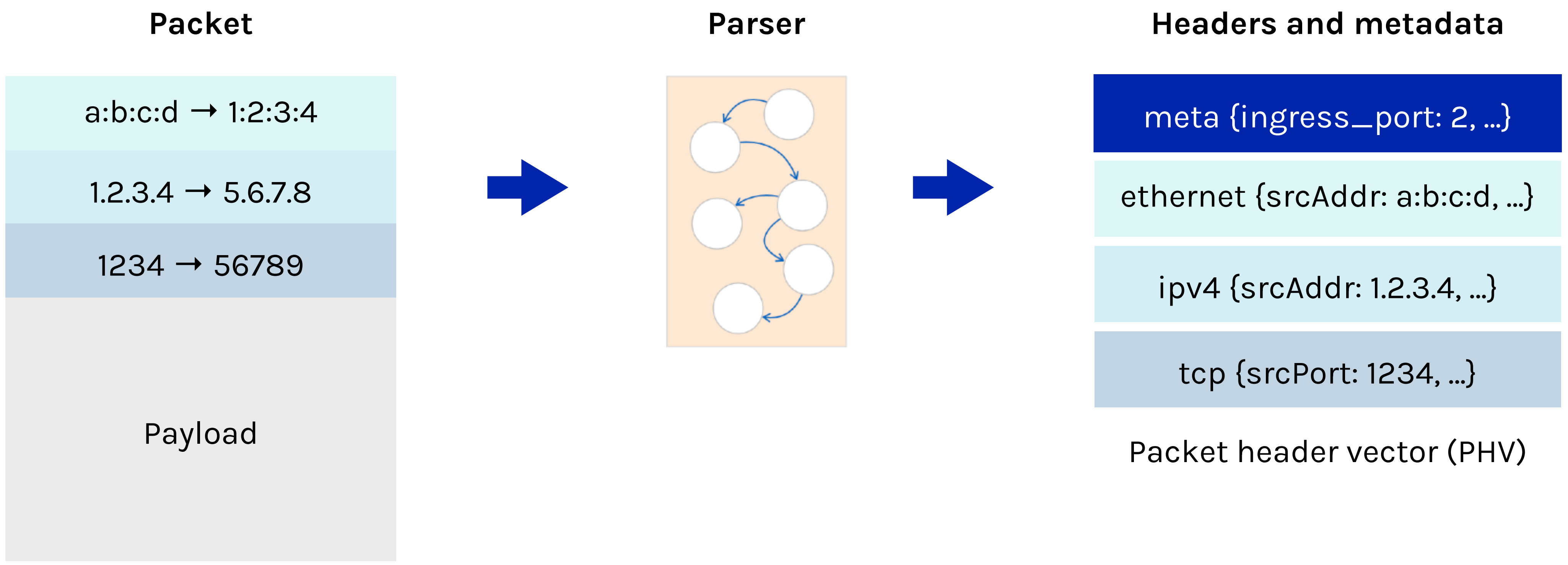
Switch	`switch (t.apply().action_run) {`
	`  action1: {…}`
	`  action2: {…}`
	`}`
		Only in control blocks
		No fall-through if a block statement is present

P4 processing overview



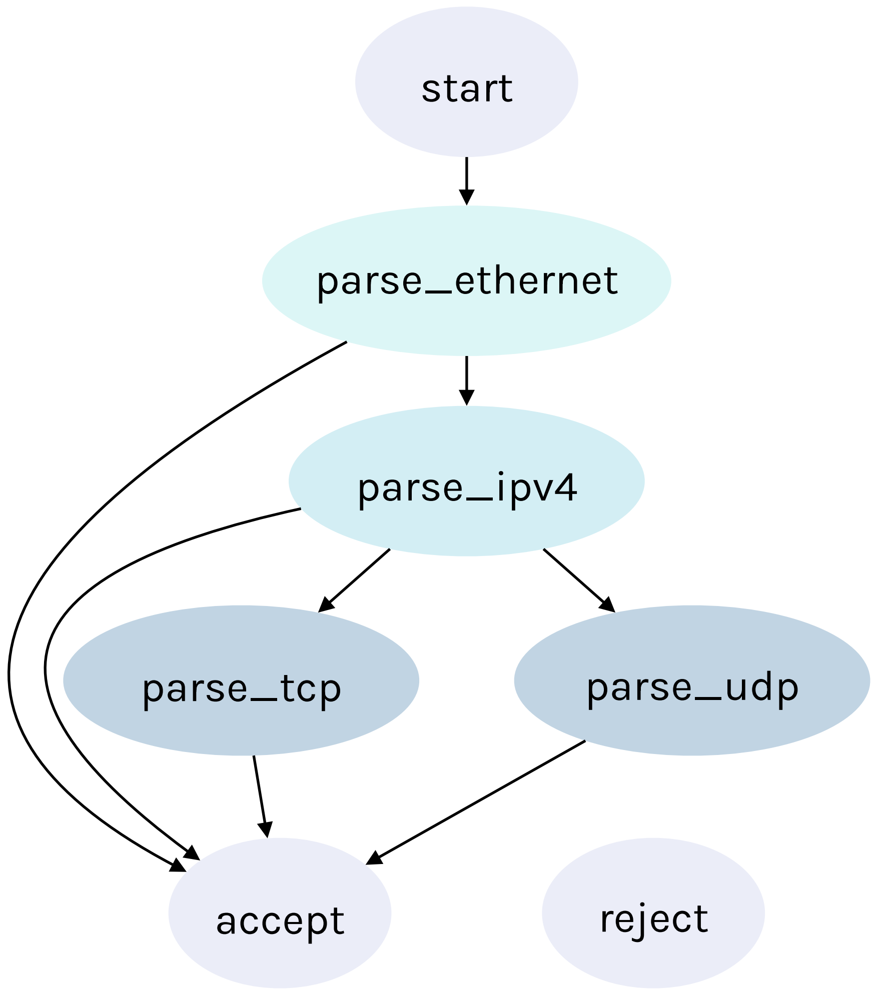
P4 parser

The parser uses a state machine to map packets into headers and metadata



Packet header vector (PHV)

P4 parser: example



```
parser MyParser(…) {
   state start {
      transition parse_ethernet;
   }
   state parse_ethernet {
      packet.extract(hdr.ethernet);
      transition select(hdr.ethernet.etherType) {
         0x800: parse_ipv4;
         default: accept;
      }
   }
   state parse_ipv4 {
      packet.extract(hdr.ipv4)
      transition select(hdr.ipv4.protocol) {
         6: parse_tcp;
         17: parse_udp;
         default: accept;
      }
   }
   state parse_tcp {
      packet.extract(hdr.tcp);
      transition accept;
   }
   state parse_udp {
      packet.extract(hdr.udp);
      transition accept;
   }
}
```

Transition between states

Footnote / references

P4 parser: variable-width header extraction

```
header IPv4_no_options_h {
    …
    bit<32> srcAddr;
    bit<32> dstAddr;
}
```

Fixed-width fields

```
header IPv4_options_h {
    varbit<32> options;
}
```

Variable-width fields

```
parser MyParser(…) {
    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition select(hdr.ipv4.ihl) {
            5: dispatch_on_protocol;
            default: parse_ipv4_options;
        }
    }
    state parse_ipv4_options {
        packet.extract(hdr.ipv4options, (hdr.ipv4.ihl - 5) << 2);
        transition dispatch_on_protocol;
    }
}
```

`ihl` determines the length of options field

P4 parser: more advanced concepts

Parsing a header stack requires the parser to loop

- The only "loops" that are possible in P4 (done implicitly through state transitions)
- Example in source routing: popping up all the headers to determine the next hop

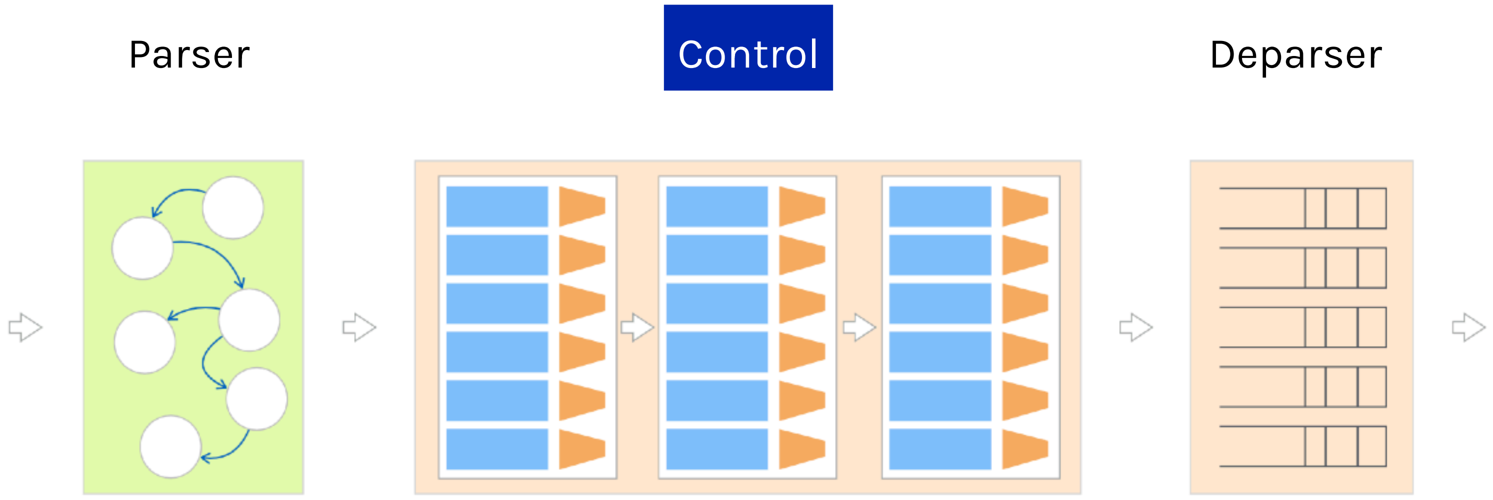
Other concepts in P4 parser:

- Verify: error handling in the parser
- Lookahead: access bits that are not parsed yet
- Sub-parsers: like subroutines

Why should we be cautious about loops?

https://p4.org/p4-spec/docs/P4-16-v1.2.1.html

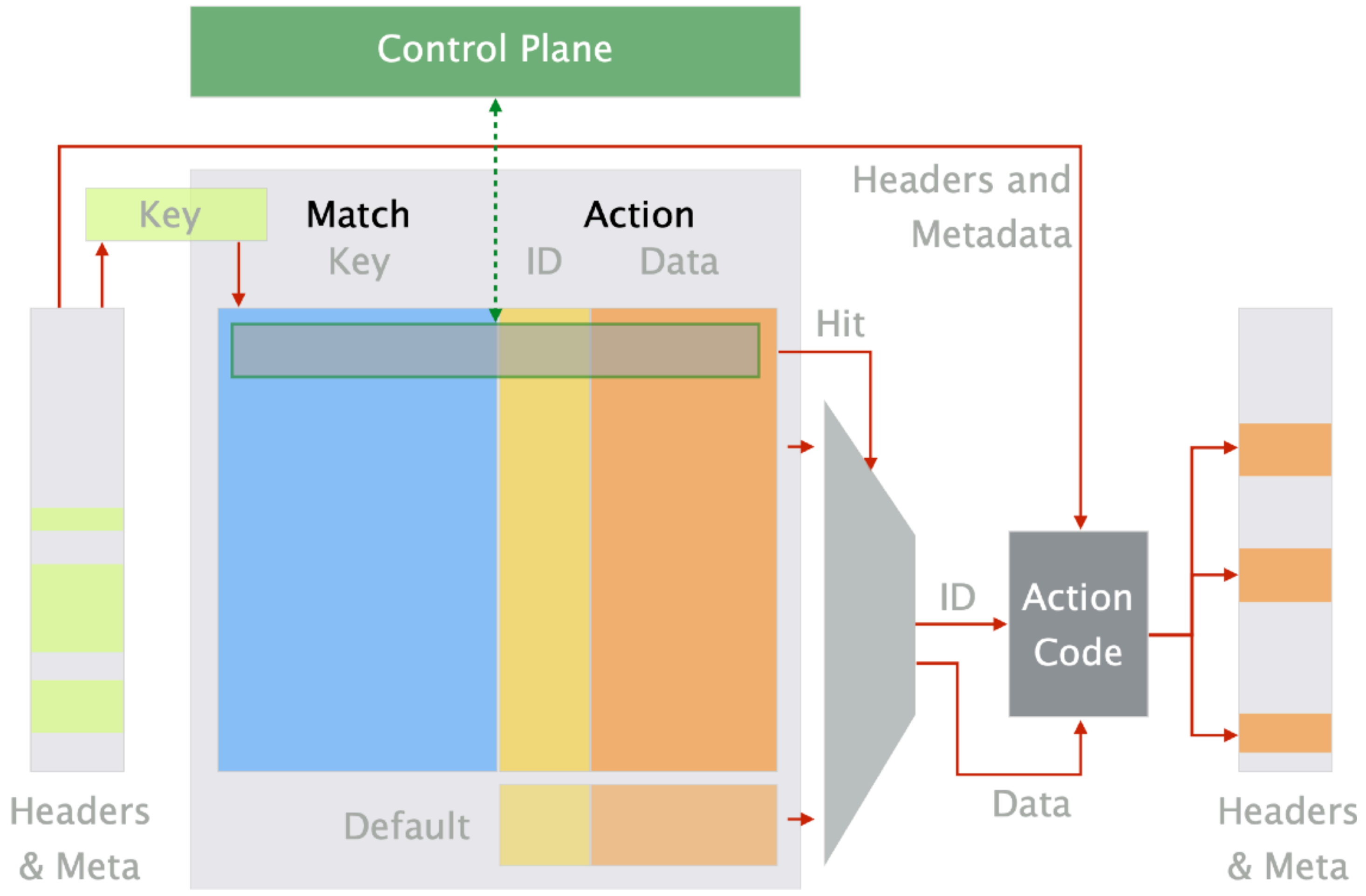
P4 processing overview



P4 control

Tables	Match a key and return an action
Actions	Similar to functions in C
Control flow	Similar to C but without loops

P4 control: tables



P4 control: tables

Table name

Longest prefix match

Possible actions

Max. # of entries in table

Default action

```
table ipv4_lpm {
  key = {
    hdr.ipv4.dstAddr: lpm;
    hdr.ipv4.version: exact;
  }
  actions = {
    ipv4_forward;
    drop;
  }

  size = 1024;
  default_action = drop();
}
```

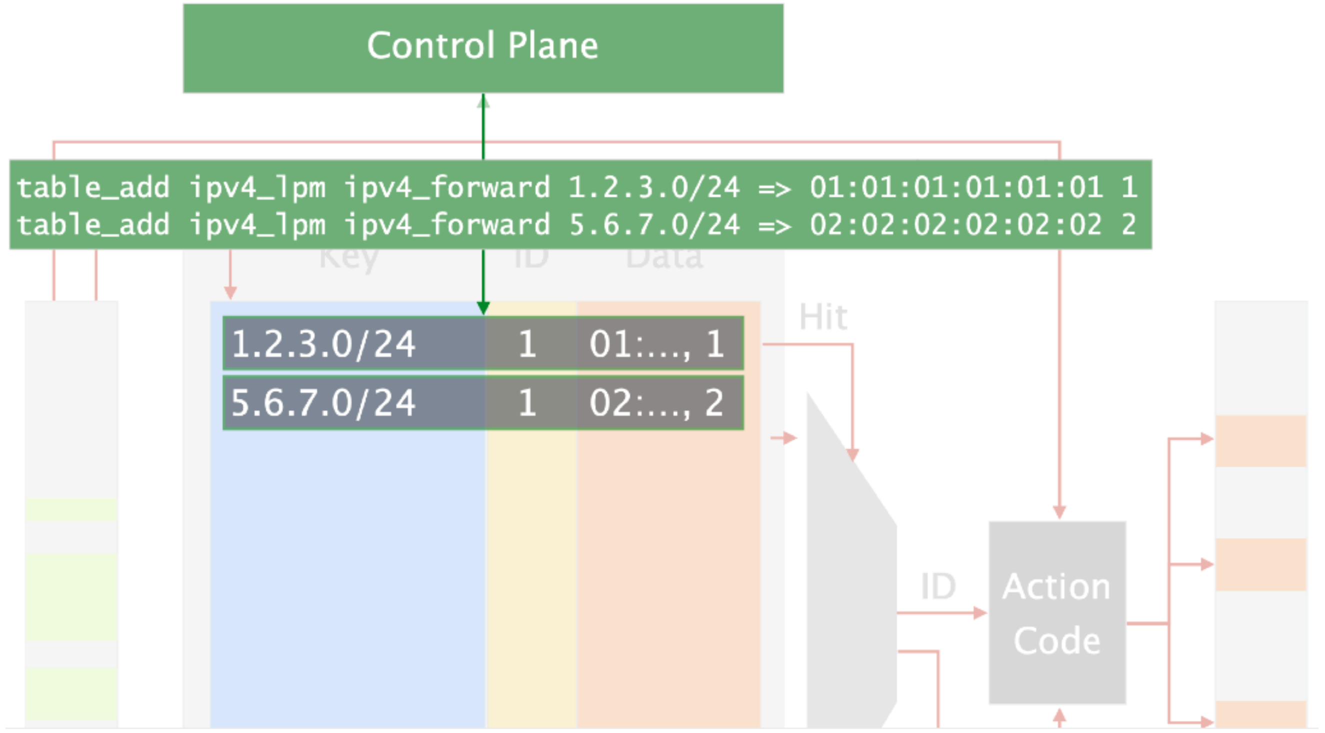
P4 control: match kinds

core.p4	`exact`	Exact comparison: `0x01020304`
	`ternary`	Compare with mask: `0x01020304 & 0x0F0F0F0F`
	`lpm`	Longest prefix match
v1model.p4	`range`	Check if in range: `0x01020304 - 0x010203FF`
Other architectures	…	

P4 control: table entries

Table entries are added through the control plane

- Recall the SDN control plane for flow rule installation



```
table_add ipv4_lpm ipv4_forward 1.2.3.0/24 => 01:01:01:01:01:01 1
table_add ipv4_lpm ipv4_forward 5.6.7.0/24 => 02:02:02:02:02:02 2
```

P4 control: actions

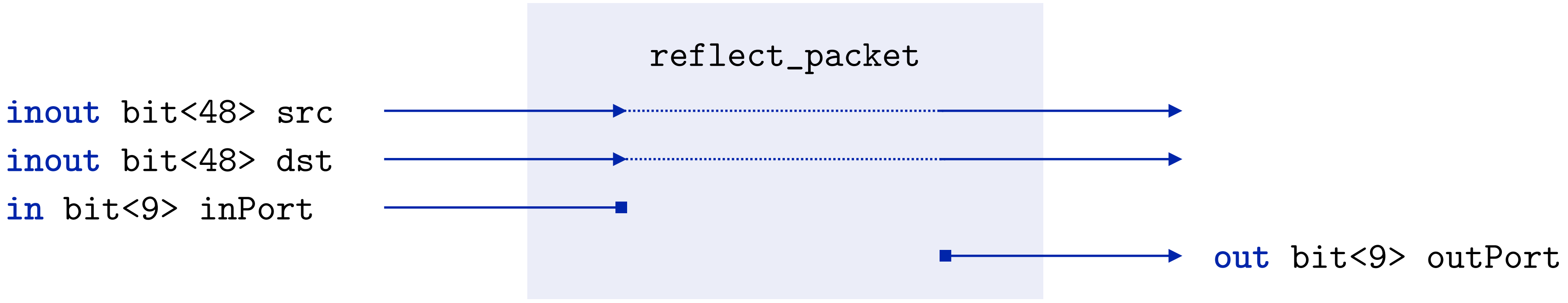
Actions are

- Blocks of statements that possibly modify the packets
- Usually take directional parameters indicating how the corresponding value is treated within the block

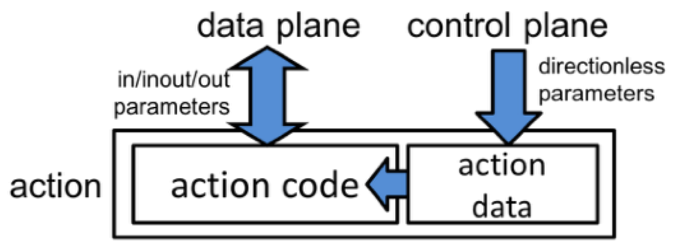
```
action reflect_packet(inout bit<48> src,
                      inout bit<48> dst,
                      in bit<9> inPort;
                      out bit<9> outPort; ) {
  bit<48> tmp = src;
  src = dst;
  dst = tmp;
  outPort = inPort;}

reflect_packet(hdr.ethernet.srcAddr, hdr.ethernet.dstAddr,
  standard_metadata.ingress_port, standard_metadata.egress_spec);
```

in: read only inside the action
out: uninitiated, write inside the action
inout: combination of in and out

P4 control: actions



```
action set_egress_port(bit<9> port) {
  standard_metadata.egress_spec = port;
}
```



Action parameters resulting from a table lookup do not take a direction

P4 control: control flow

Apply a table
```
ipv4_lpm.apply()
```

Check if there was a hit
```
if (ipv4_lpm.apply().hit) {…}
else {…}
```

Check which action was executed
```
switch (ipv4_lpm.apply().action_run) {
  ipv4_forward: {…}
}
```

v1model.p4

```
extern void verify_checksum<T, O>(
       in bool condition,
       in T data,
       inout O checksum,
       HashAlgorithm algo);

extern void update_checksum<T, O>(
       in bool condition,
       in T data,
       inout O checksum,
       HashAlgorithm algo);
```

P4 control: re-computing checksum

```
control MyComputeChecksum {
  apply {
    update_checksum(
      hdr.ipv4.isValid(),
      { hdr.ipv4.version,
        hdr.ipv4.ihl,
        hdr.ipv4.diffserv,
        hdr.ipv4.totalLen,
        hdr.ipv4.identification,
        hdr.ipv4.flags,
        hdr.ipv4.fragOffset,
        hdr.ipv4.ttl,
        hdr.ipv4.protocol,
        hdr.ipv4.srcAddr,
        hdr.ipv4.dstAddr },
      hdr.ipv4.hdrChecksum,
      HashAlgorithm.csum16);
  }
}
```

Pre-condition

Fields list

Checksum field

Checksum algorithm

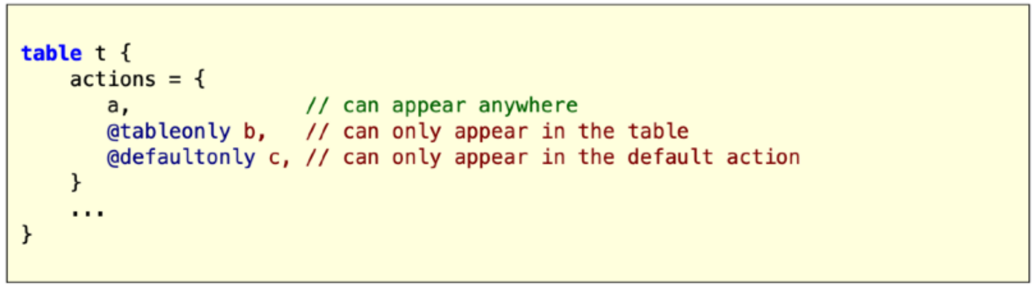
P4 control: more advanced concepts

Cloning packets	Create a clone of a packet
Sending packets to control plane	Use dedicated Ethernet port, or target-specific mechanisms
Recirculating	Send packet through pipeline multiple times

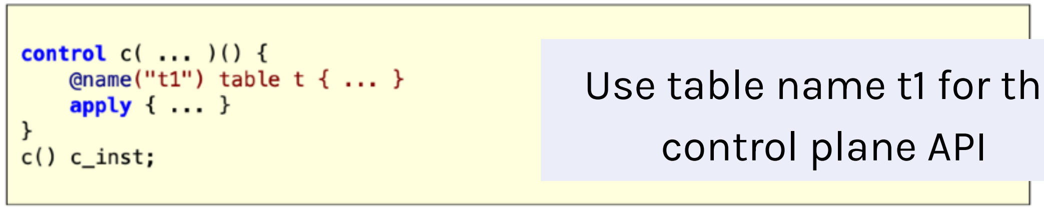
Be cautious about recirculating!

Annotations

Additional information given to the compiler or the control plane

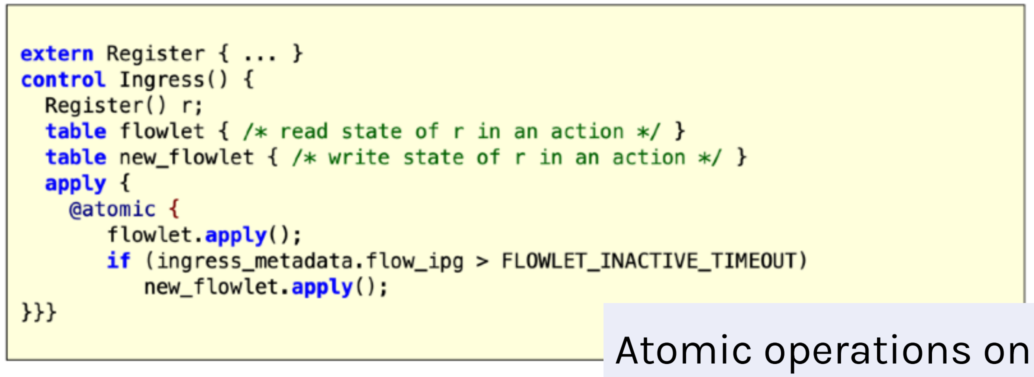


```
table t {
    actions = {
       a,              // can appear anywhere
       @tableonly b,   // can only appear in the table
       @defaultonly c, // can only appear in the default action
    }
    ...
}
```



```
control c( ... )() {
    @name("t1") table t { ... }
    apply { ... }
}
c() c_inst;
```

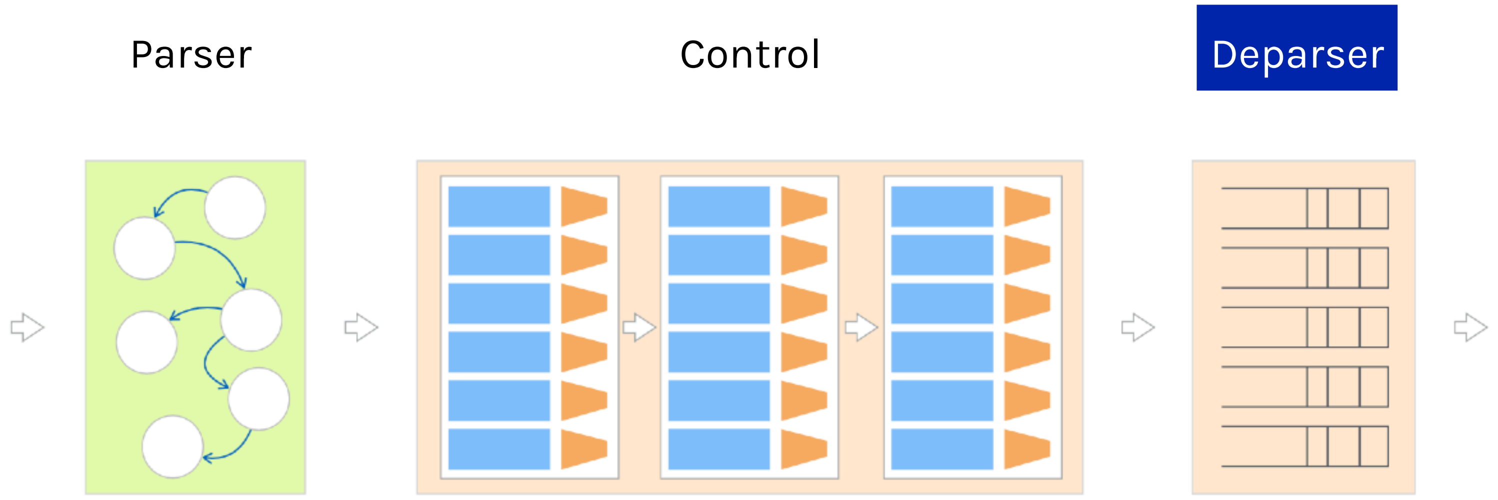
Use table name t1 for the control plane API



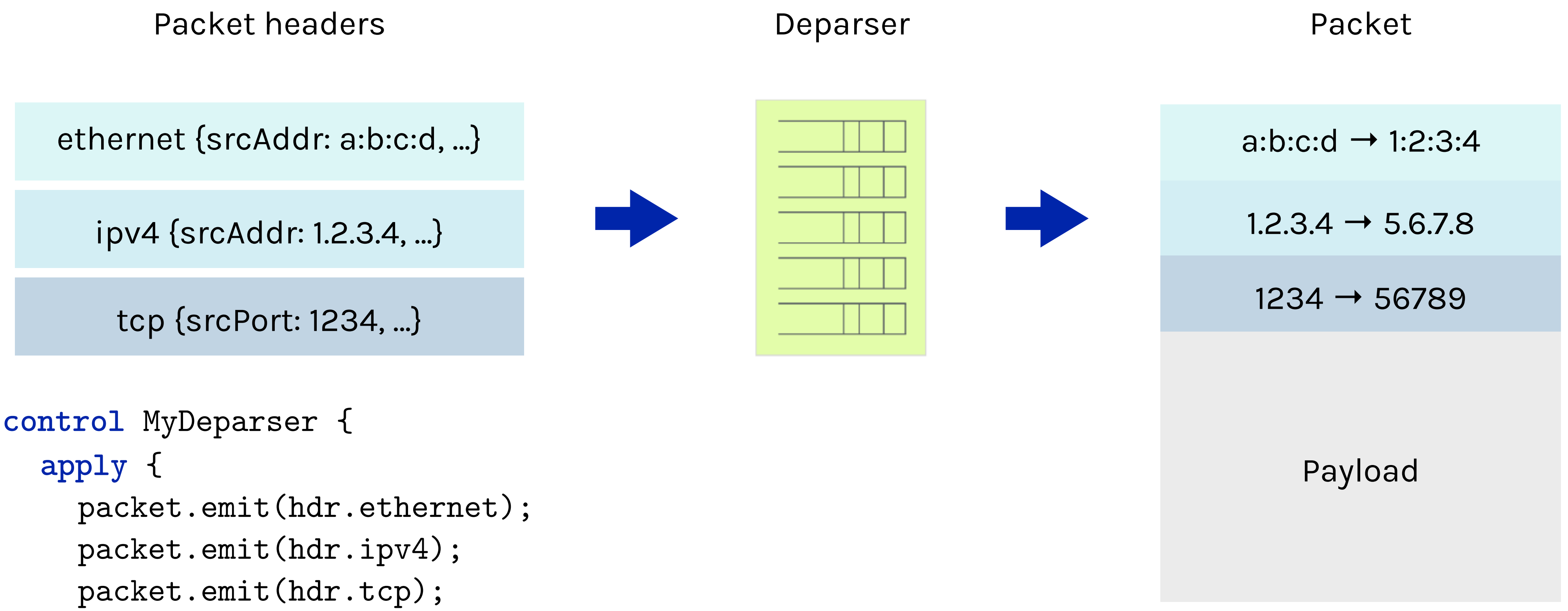
```
extern Register { ... }
control Ingress() {
  Register() r;
  table flowlet { /* read state of r in an action */ }
  table new_flowlet { /* write state of r in an action */ }
  apply {
    @atomic {
       flowlet.apply();
       if (ingress_metadata.flow_ipg > FLOWLET_INACTIVE_TIMEOUT)
          new_flowlet.apply();
}}}
```

Atomic operations on registers

P4 processing overview

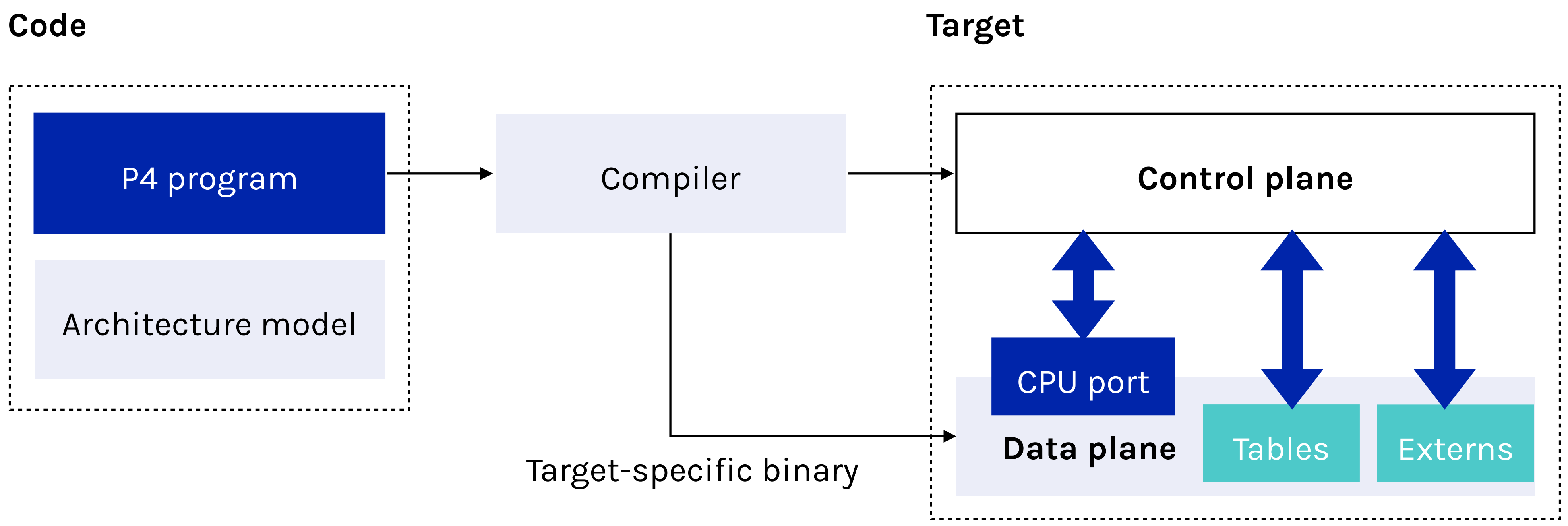


P4 deparser



```
control MyDeparser {
  apply {
    packet.emit(hdr.ethernet);
    packet.emit(hdr.ipv4);
    packet.emit(hdr.tcp);
  }
}
```

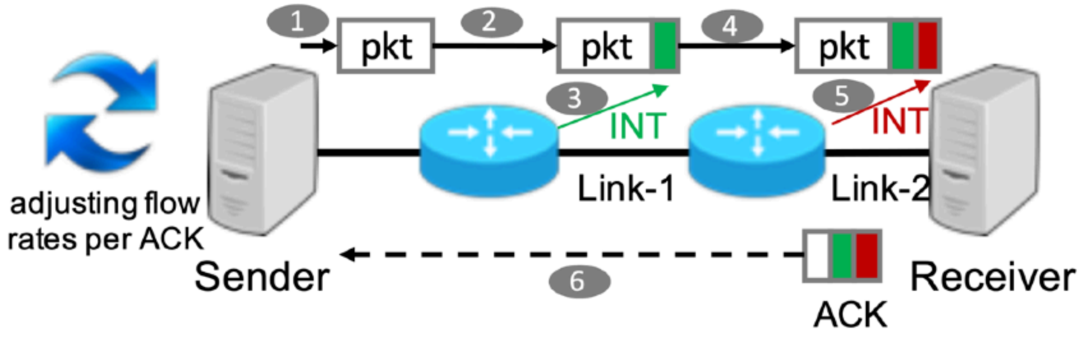
P4 workflow



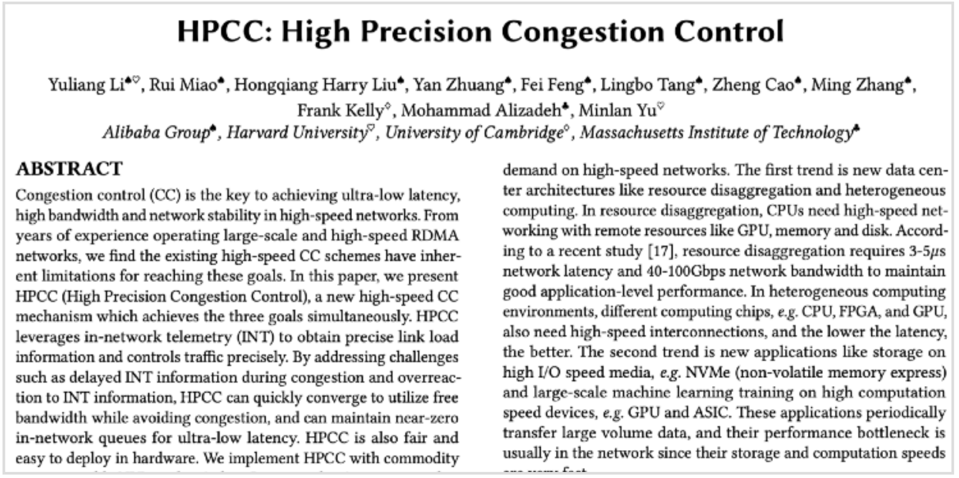
🟦 User supplied

⬜ Vendor supplied

Application: congestion control



Use INT to **obtain precise network link status information** and adjust sending rate based on such information



HPCC: High Precision Congestion Control

Yuliang Li[♠♡], Rui Miao[♠], Hongqiang Harry Liu[♠], Yan Zhuang[♠], Fei Feng[♠], Lingbo Tang[♠], Zheng Cao[♠], Ming Zhang[♠], Frank Kelly[◇], Mohammad Alizadeh[♣], Minlan Yu[♡]
Alibaba Group[♠], Harvard University[♡], University of Cambridge[◇], Massachusetts Institute of Technology[♣]

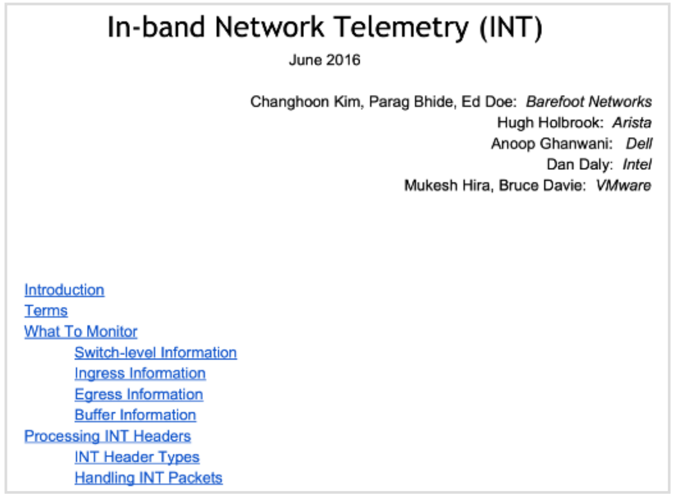
ABSTRACT

Congestion control (CC) is the key to achieving ultra-low latency, high bandwidth and network stability in high-speed networks. From years of experience operating large-scale and high-speed RDMA networks, we find the existing high-speed CC schemes have inherent limitations for reaching these goals. In this paper, we present HPCC (High Precision Congestion Control), a new high-speed CC mechanism which achieves the three goals simultaneously. HPCC leverages in-network telemetry (INT) to obtain precise link load information and controls traffic precisely. By addressing challenges such as delayed INT information during congestion and overreaction to INT information, HPCC can quickly converge to utilize free bandwidth while avoiding congestion, and can maintain near-zero in-network queues for ultra-low latency. HPCC is also fair and easy to deploy in hardware. We implement HPCC with commodity

demand on high-speed networks. The first trend is new data center architectures like resource disaggregation and heterogeneous computing. In resource disaggregation, CPUs need high-speed networking with remote resources like GPU, memory and disk. According to a recent study [17], resource disaggregation requires 3-5μs network latency and 40-100Gbps network bandwidth to maintain good application-level performance. In heterogeneous computing environments, different computing chips, *e.g.* CPU, FPGA, and GPU, also need high-speed interconnections, and the lower the latency, the better. The second trend is new applications like storage on high I/O speed media, *e.g.* NVMe (non-volatile memory express) and large-scale machine learning training on high computation speed devices, *e.g.* GPU and ASIC. These applications periodically transfer large volume data, and their performance bottleneck is usually in the network since their storage and computation speeds

Think about the difference to ECN

Other PDP applications

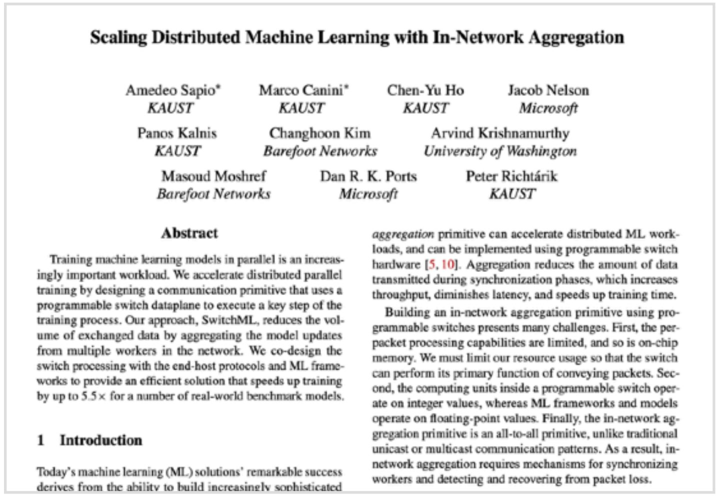


In-band Network Telemetry (INT)

June 2016

Changhoon Kim, Parag Bhide, Ed Doe: *Barefoot Networks*
Hugh Holbrook: *Arista*
Anoop Ghanwani: *Dell*
Dan Daly: *Intel*
Mukesh Hira, Bruce Davie: *VMware*

Network monitoring



Scaling Distributed Machine Learning with In-Network Aggregation

Amedeo Sapio* *KAUST* · Marco Canini* *KAUST* · Chen-Yu Ho *KAUST* · Jacob Nelson *Microsoft*
Panos Kalnis *KAUST* · Changhoon Kim *Barefoot Networks* · Arvind Krishnamurthy *University of Washington*
Masoud Moshref *Barefoot Networks* · Dan R. K. Ports *Microsoft* · Peter Richtárik *KAUST*

Abstract

Training machine learning models in parallel is an increasingly important workload. We accelerate distributed parallel training by designing a communication primitive that uses a programmable switch dataplane to execute a key step of the training process. Our approach, SwitchML, reduces the volume of exchanged data by aggregating the model updates from multiple workers in the network. We co-design the switch processing with the end-host protocols and ML frameworks to provide an efficient solution that speeds up training by up to 5.5× for a number of real-world benchmark models.

1 Introduction

Today's machine learning (ML) solutions' remarkable success derives from the ability to build increasingly sophisticated

aggregation primitive can accelerate distributed ML workloads, and can be implemented using programmable switch hardware [5, 10]. Aggregation reduces the amount of data transmitted during synchronization phases, which increases throughput, diminishes latency, and speeds up training time.

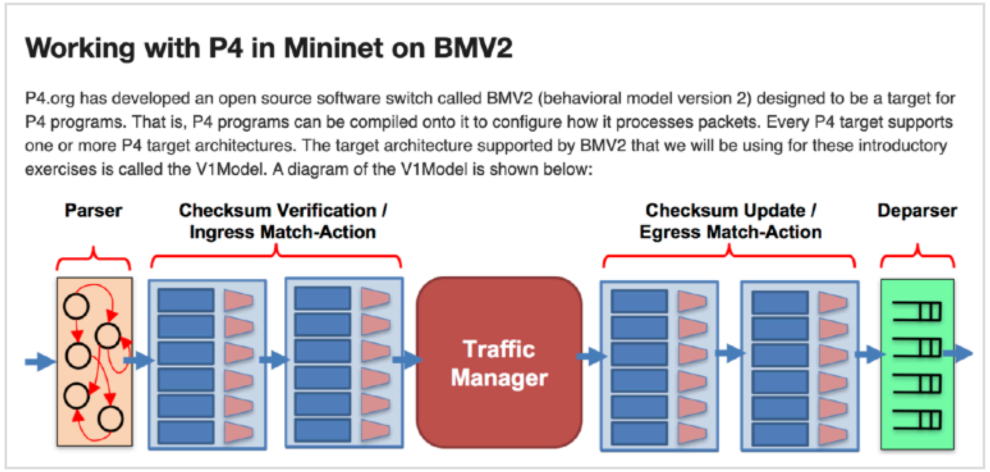
Building an in-network aggregation primitive using programmable switches presents many challenges. First, the per-packet processing capabilities are limited, and so is on-chip memory. We must limit our resource usage so that the switch can perform its primary function of conveying packets. Second, the computing units inside a programmable switch operate on integer values, whereas ML frameworks and models operate on floating-point values. Finally, the in-network aggregation primitive is an all-to-all primitive, unlike traditional unicast or multicast communication patterns. As a result, in-network aggregation requires mechanisms for synchronizing workers and detecting and recovering from packet loss.

In-network computing

Try out P4

P4 hands-on

- Use Mininet to set up the network environment
- Use software switches **bmv2**: https://github.com/p4lang/behavioral-model
- See P4 tutorials: https://github.com/p4lang/tutorials

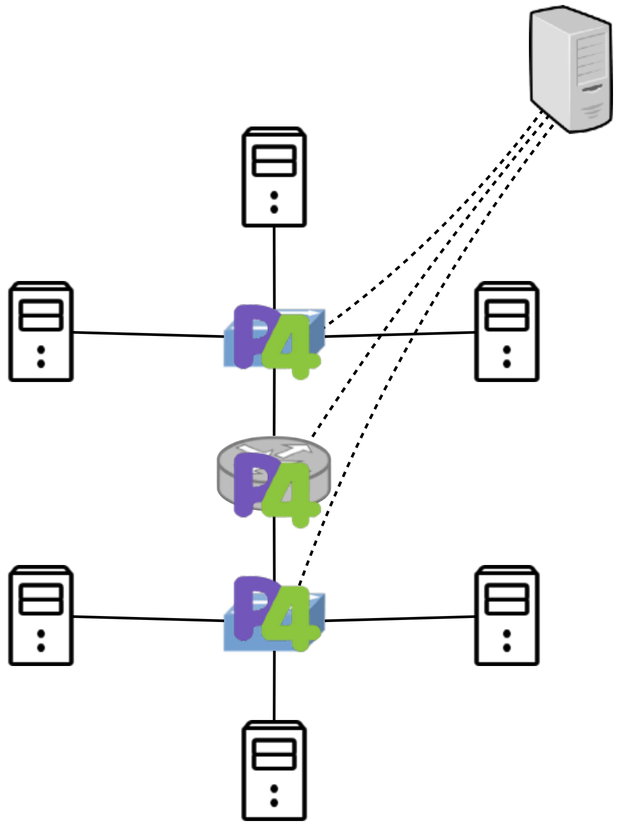


Working with P4 in Mininet on BMV2

P4.org has developed an open source software switch called BMV2 (behavioral model version 2) designed to be a target for P4 programs. That is, P4 programs can be compiled onto it to configure how it processes packets. Every P4 target supports one or more P4 target architectures. The target architecture supported by BMV2 that we will be using for these introductory exercises is called the V1Model. A diagram of the V1Model is shown below:

Parser
Checksum Verification / Ingress Match-Action
Traffic Manager
Checksum Update / Egress Match-Action
Deparser

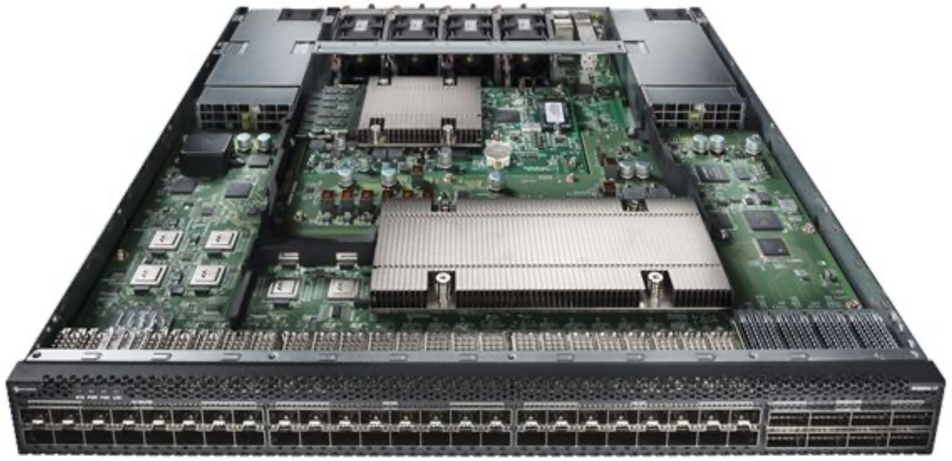
Summary



Data plane programmability needed by the demand of more flexible network configurations

RMT abstracts the data plane architecture and P4 enables data plane programmability

Next time: programmable switch architecture



How does a programmable switch work from the inside out?