Advanced Computer Networks

CS740

Programmable Switch

https://pages.cs.wisc.edu/~mgliu/CS740/F25/index.html

Ming Liu
mgliu@cs.wisc.edu

Outline

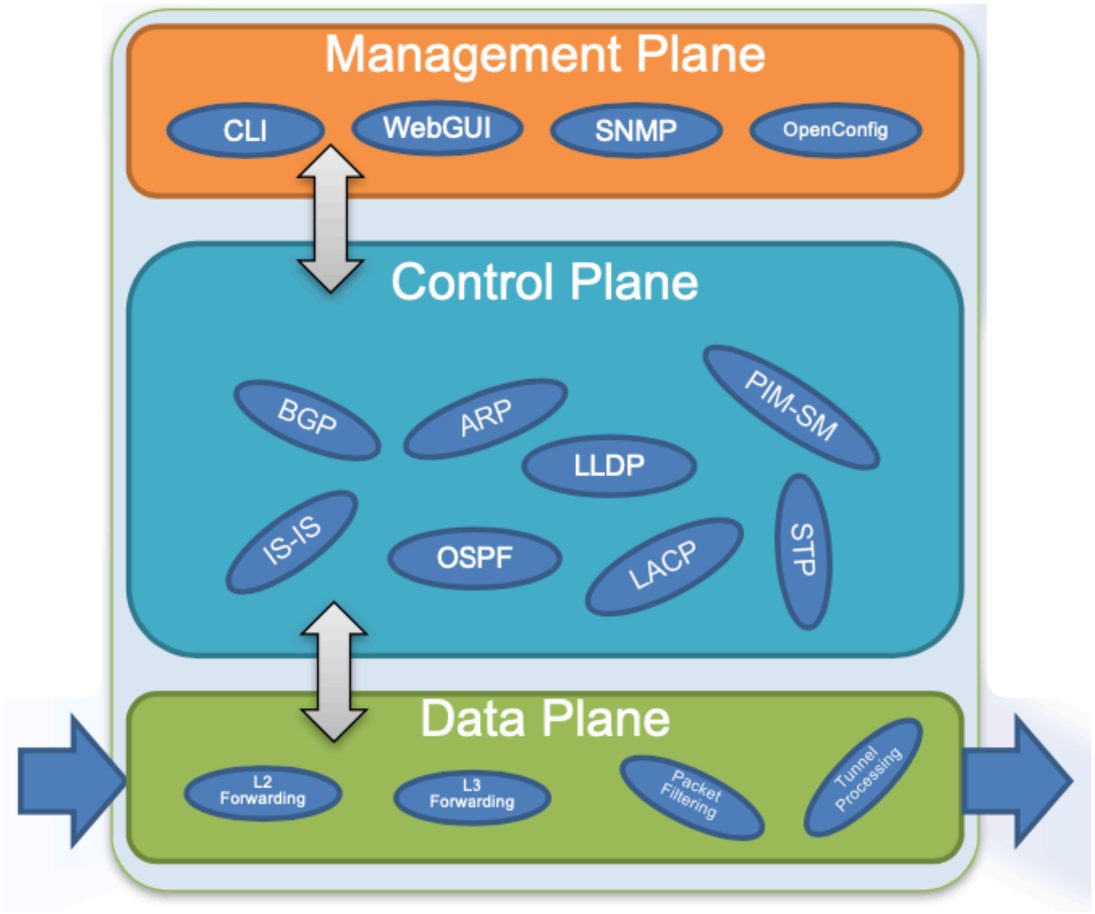
- Last lecture
  - Software-Defined Network
- Today
  - Programmable Switch
- Announcements
  - Lab2 due 11/05/2025 11:59 PM
  - Midterm report due 11/04/2025 11:59 PM

Some Terms

- Management Plane: device configuration
  - Manage the control plane (and the system)
  - E.g., CLI, GUI,…
- Control Plane: routing, discovery, …
  - Control the data plane
  - Run the protocol logics
- Data Plane: packet forwarding
  - Determine how packets traverse the switching pipeline

Some Terms

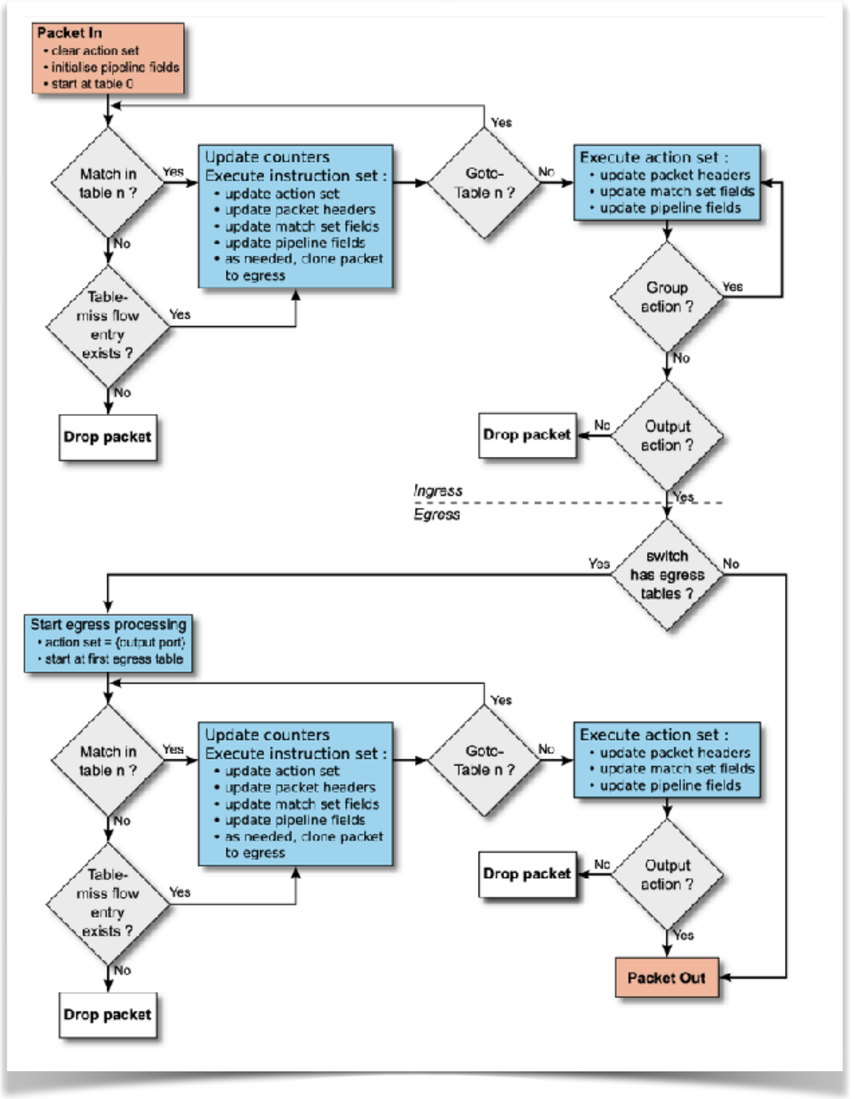
- Management
  - Manage the
  - E.g., CLI, G
- Control Plan
  - Control the
  - Run the prot
- Data Plane:
  - Determine h



SDN and Limitations

- Programmable control plane
- High-bandwidth data plane
  - Limited flexibility

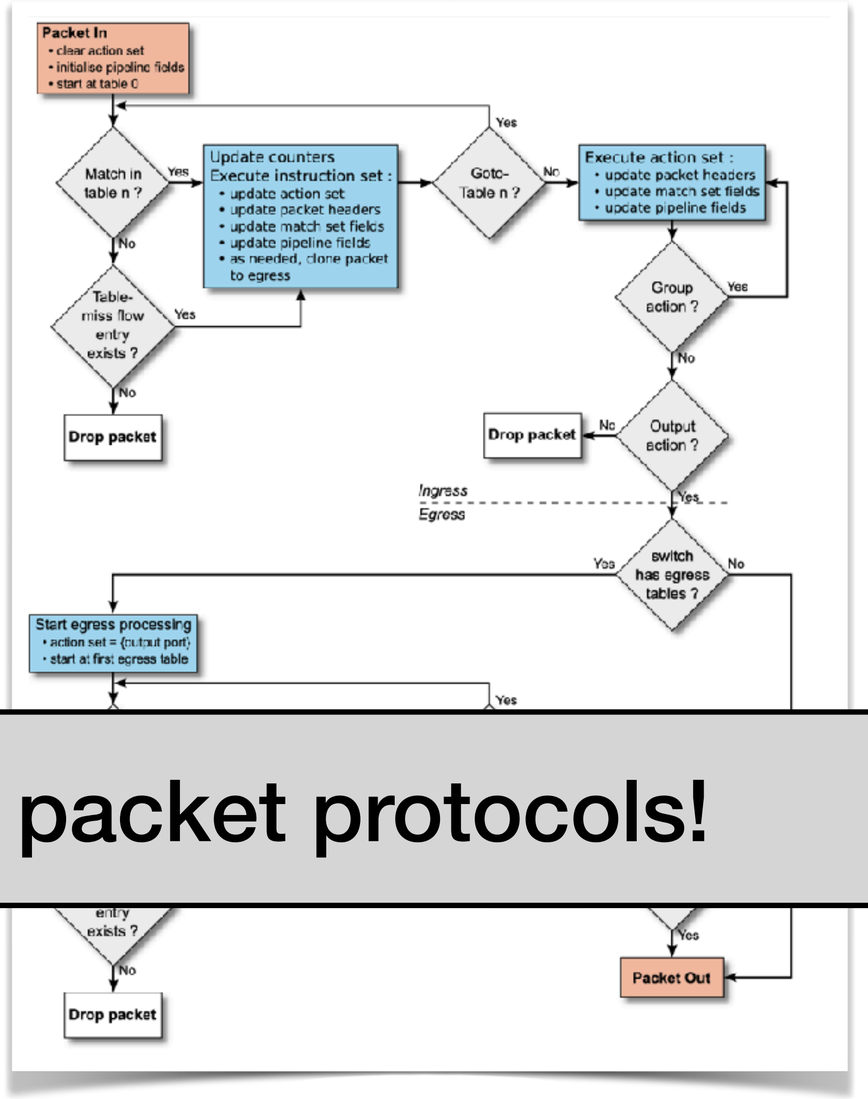
Version	Date	Header Fields
OF 1.0	Dec 2009	12 fields (Ethernet, TCP/IPv4)
OF 1.1	Feb 2011	15 fields (MPLS, inter-table metadata)
OF 1.2	Dec 2011	36 fields (ARP, ICMP, IPv6, etc.)
OF 1.3	Jun 2012	40 fields
OF 1.4	Oct 2013	41 fields



https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

SDN and Limitations

- Programmable control plane
- High-bandwidth data plane
  - Limited flexibility



Restricted to conventional packet protocols!

https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

Let’s make the data plane programmable.

Let’s make the data plane programmable.

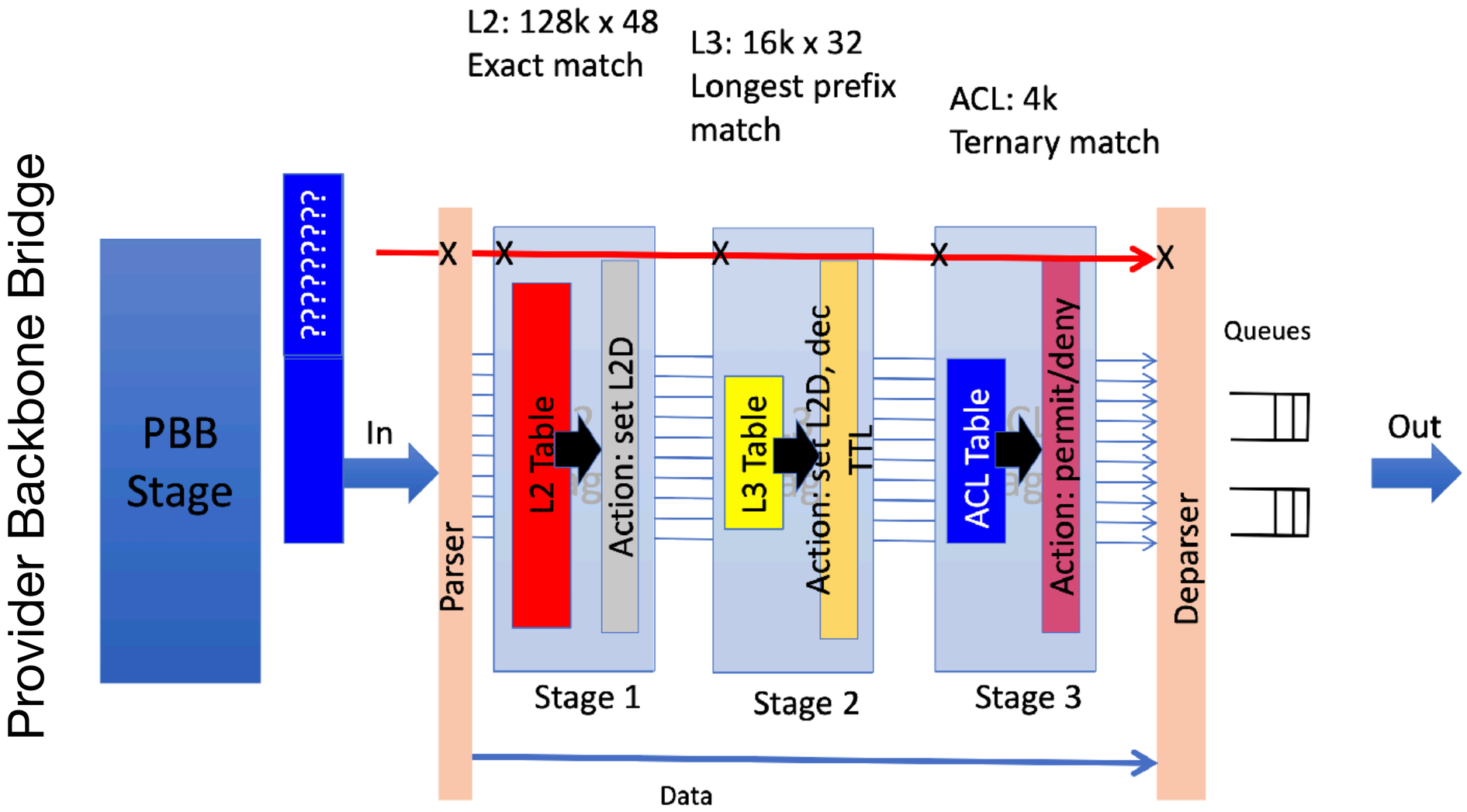
But, how to do it?

Let’s make the data plane programmable.

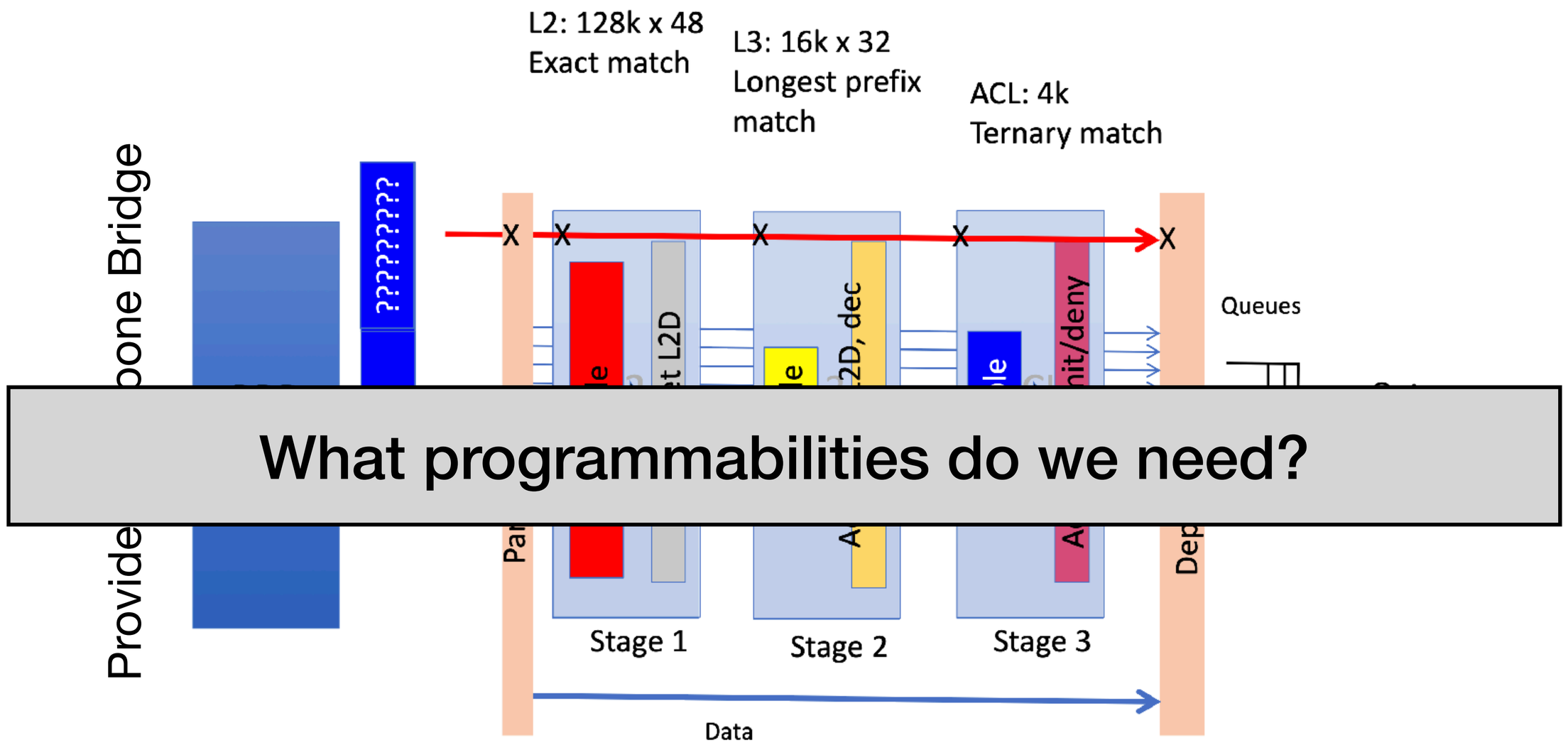
But, how to do it?

More importantly, how to make the data plane programmable without losing BW?

Fixed Function Switch



Fixed Function Switch



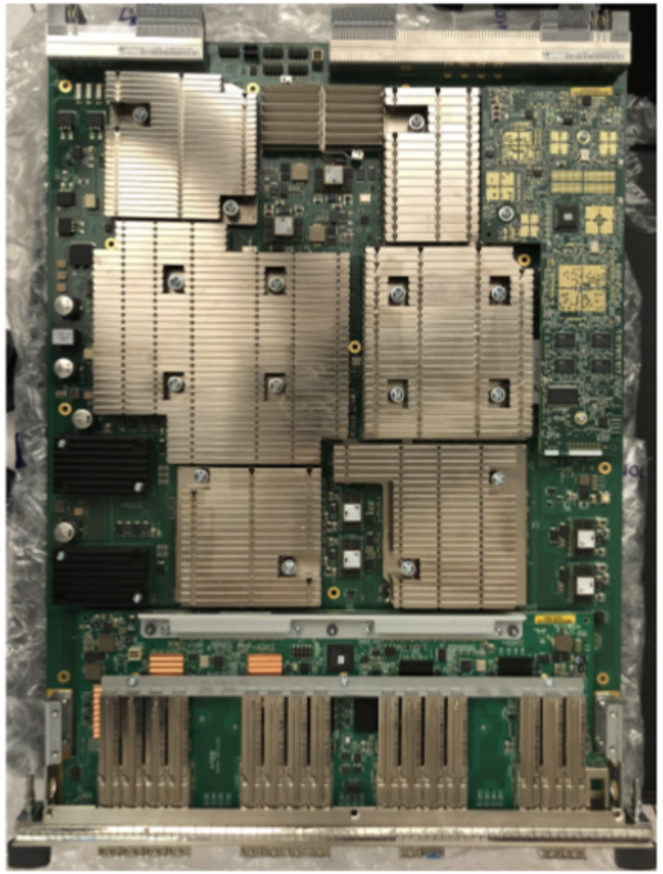
What programmabilities do we need?

Goal: add flexibility to packet forwarding

- Add a different header field
- Add a new table
- Add a different action
- Dynamic memory allocation
- Programmable packet scheduling
- ……

Designing a Flexible Switch is Hard!

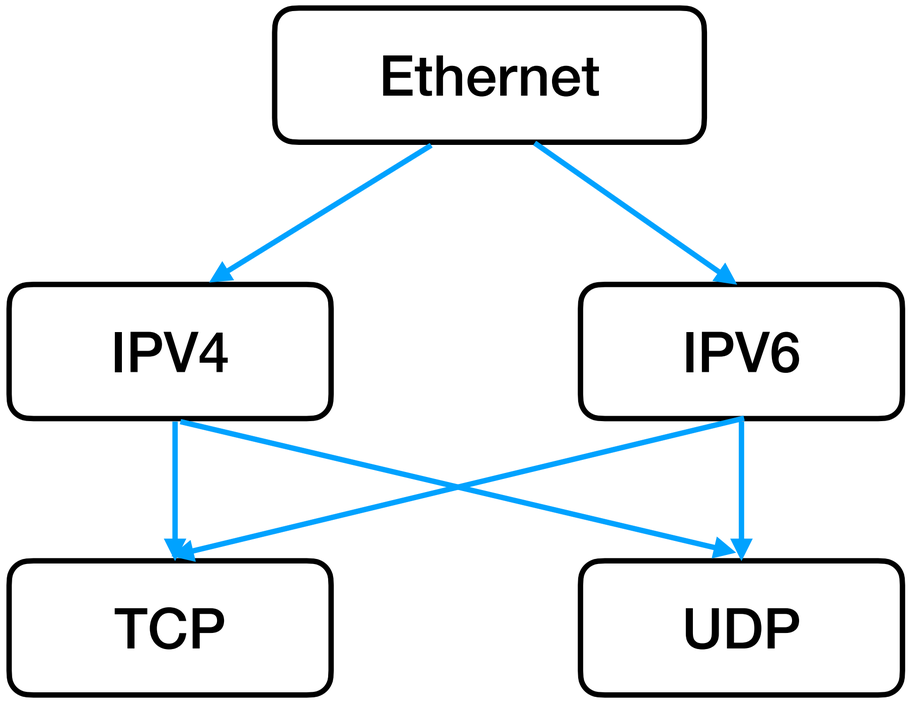
- Big chip
- High frequency
- Massive bandwidth
- Wiring intensive
- Many crossbars
- Lots of TCAM
- SerDes
- ……



RMT=Reconfigurable Match-action Table

Technique #1: Parser Graph

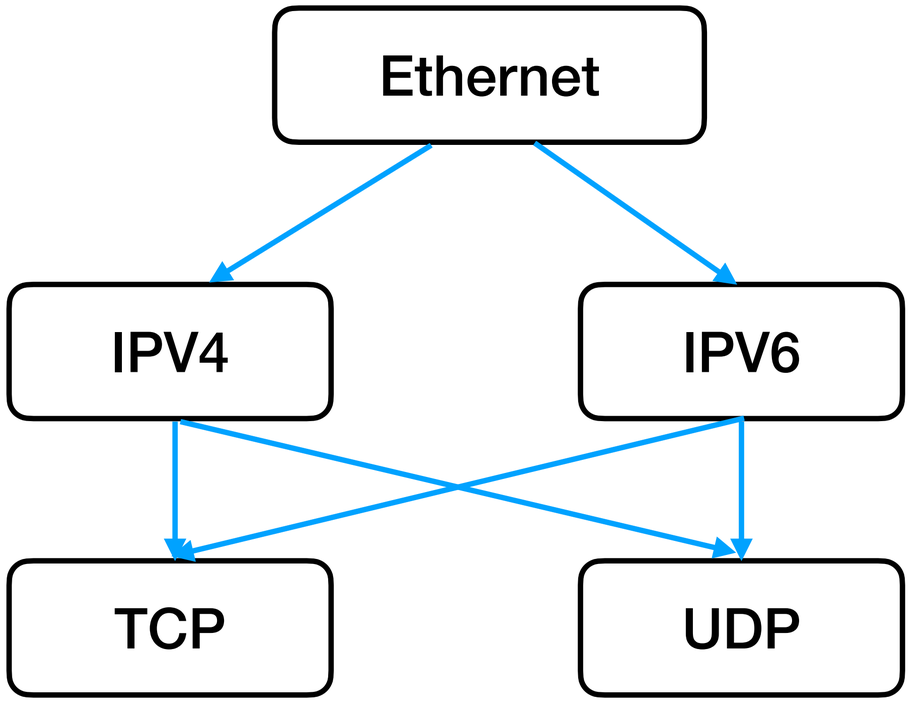
- Programmable parser
  - Arbitrary fields



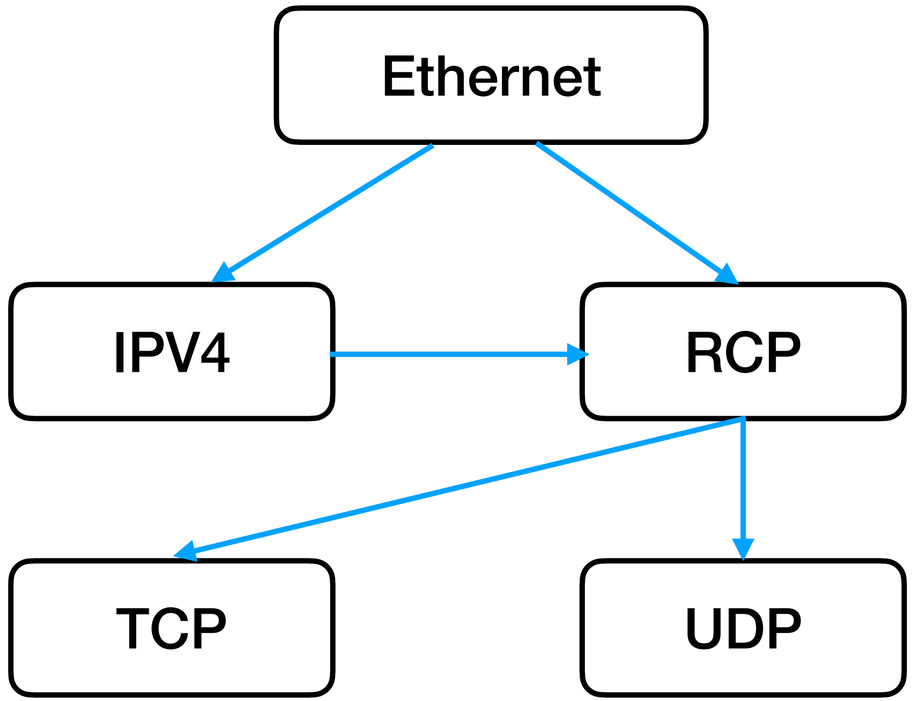
Default

Technique #1: Parser Graph

- Programmable parser
  - Arbitrary fields

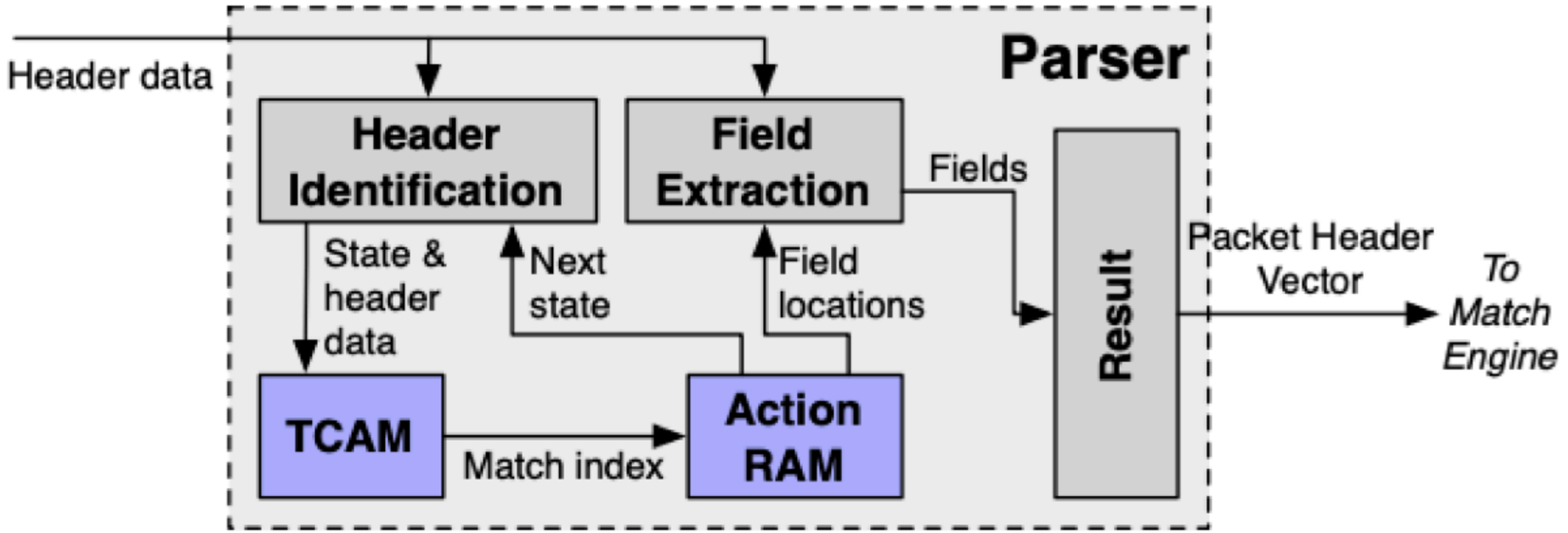


Default



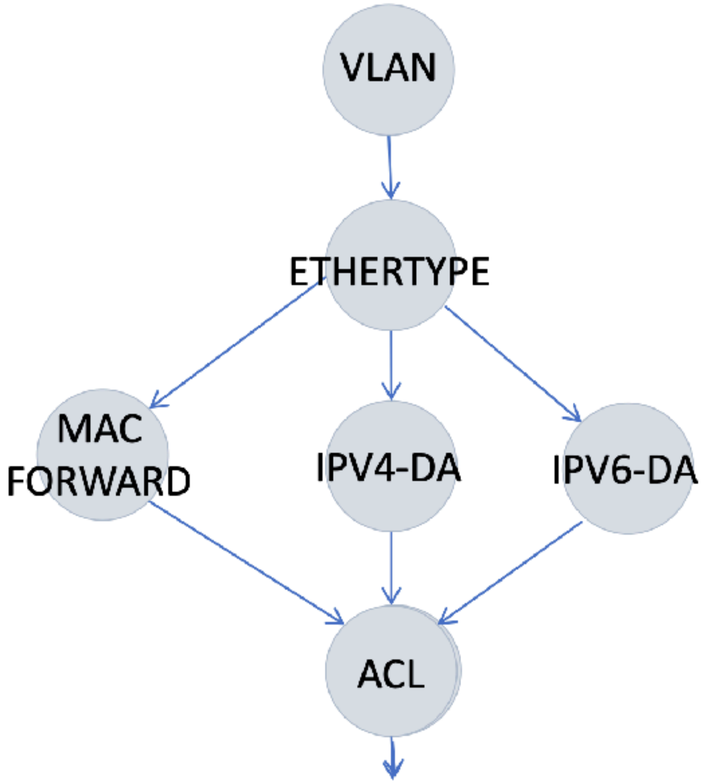
New

Programmable Parser Hardware Architecture



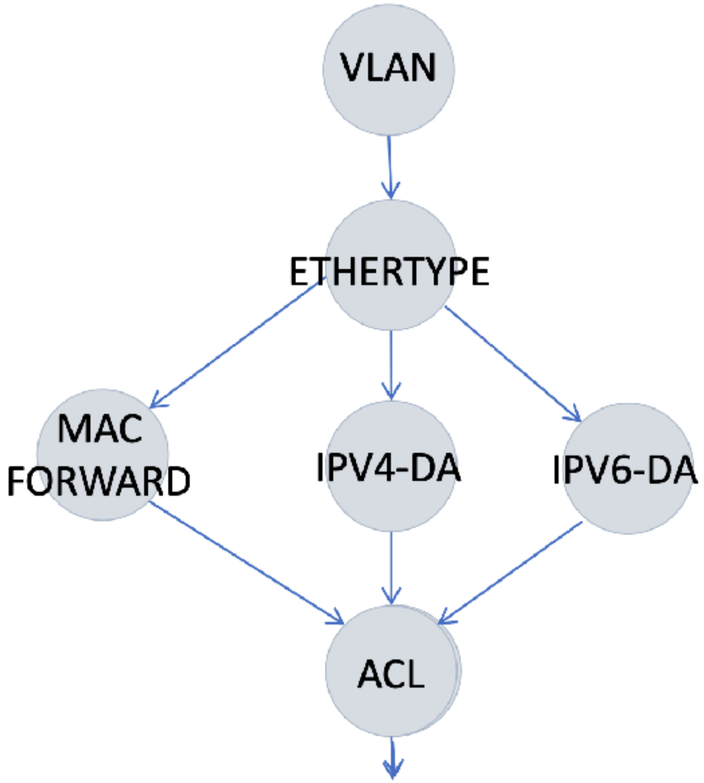
Technique #2: Table Graph

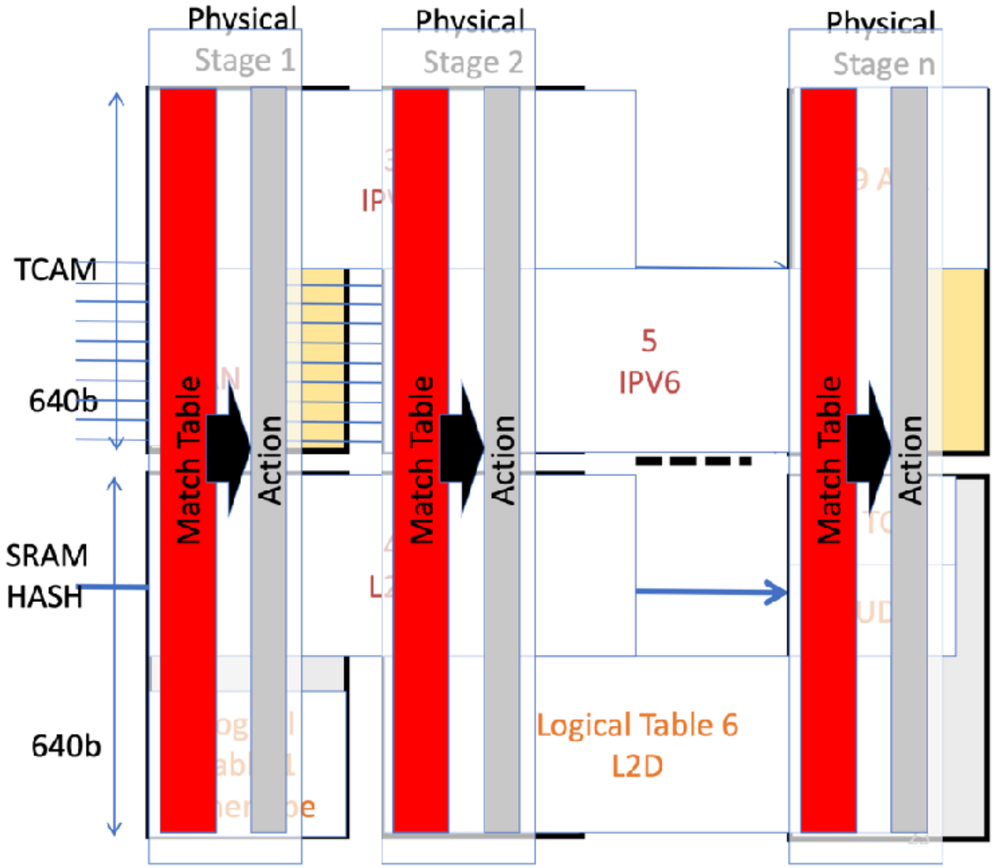
- Dynamic table memory provisioning
  - No static allocation



Technique #2: Table Graph

- Dynamic table memory provisioning
  - No static allocation





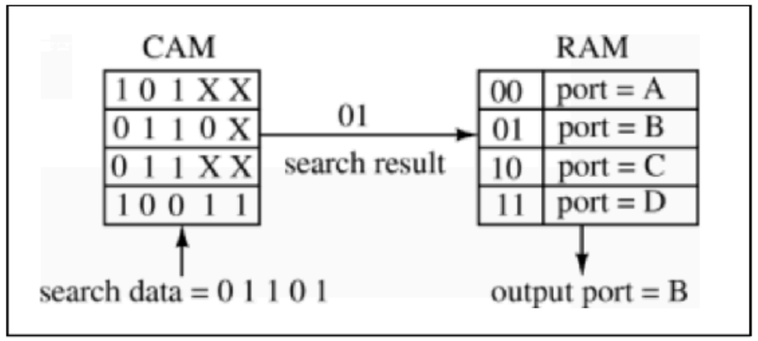
Recap: CAMs and RAMs

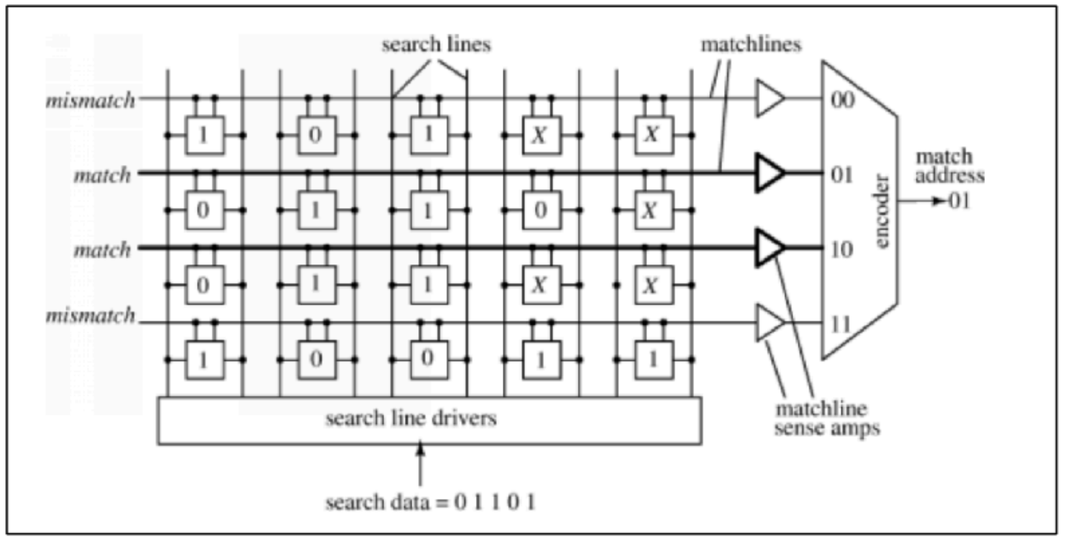
- RAM:
  - Look up the value associated with a memory address

- CAM:
  - Look up the memory address of a given value
  - Binary CAM: exact match (matches on 0 or 1)
  - Ternary CAM (TCAM): allow wildcard (matched on 0, 1, or X)

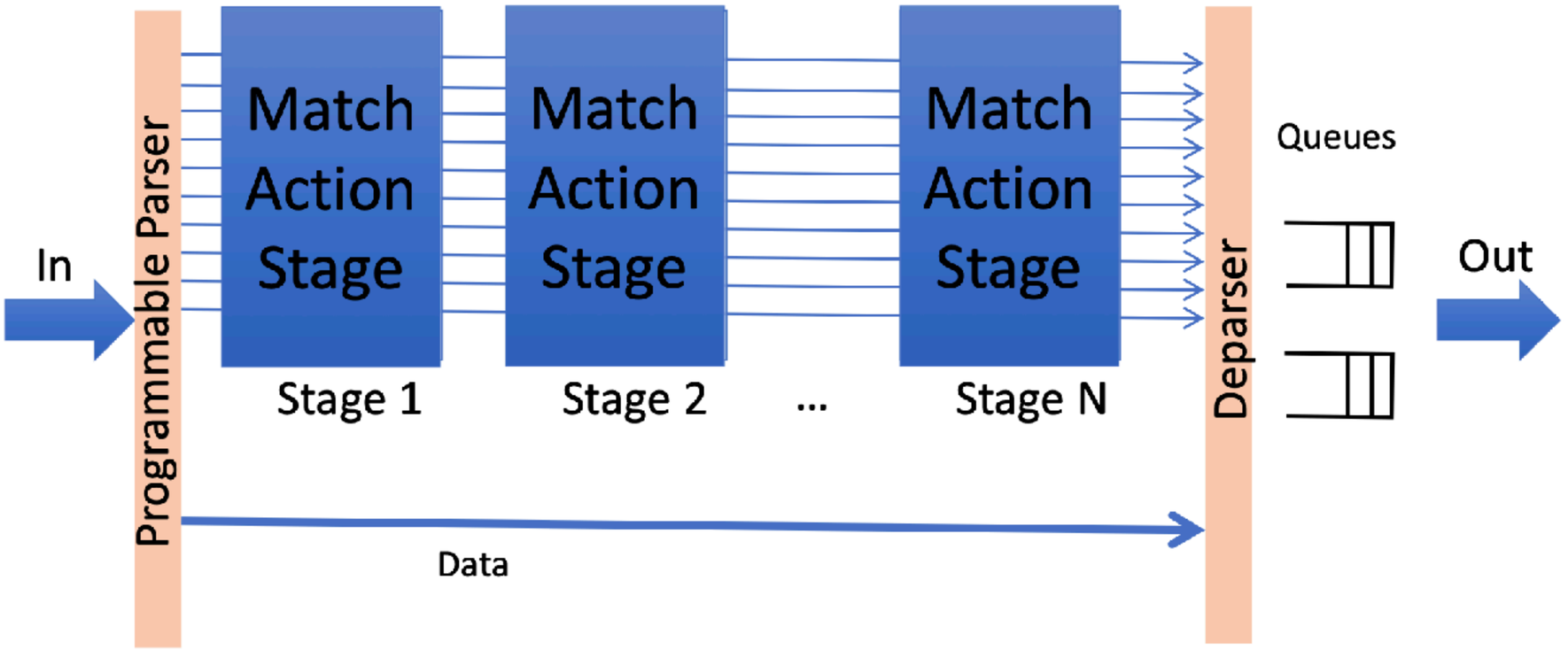
A CAM Example

Line No.	Address (Binary)	Output Port
1	101XX	A
2	0110X	B
3	011XX	C
4	10011	D

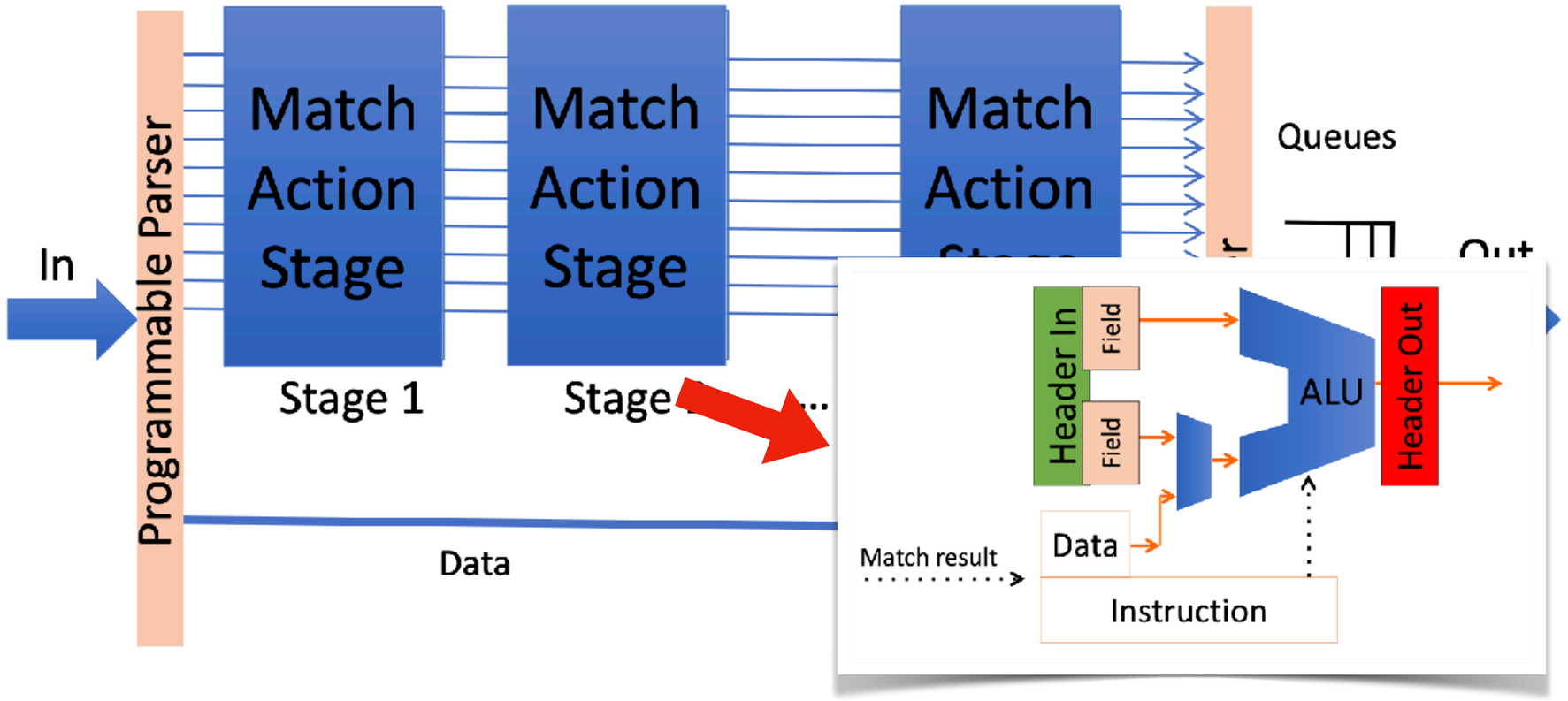




Technique #3: Match/Action Forwarding Model

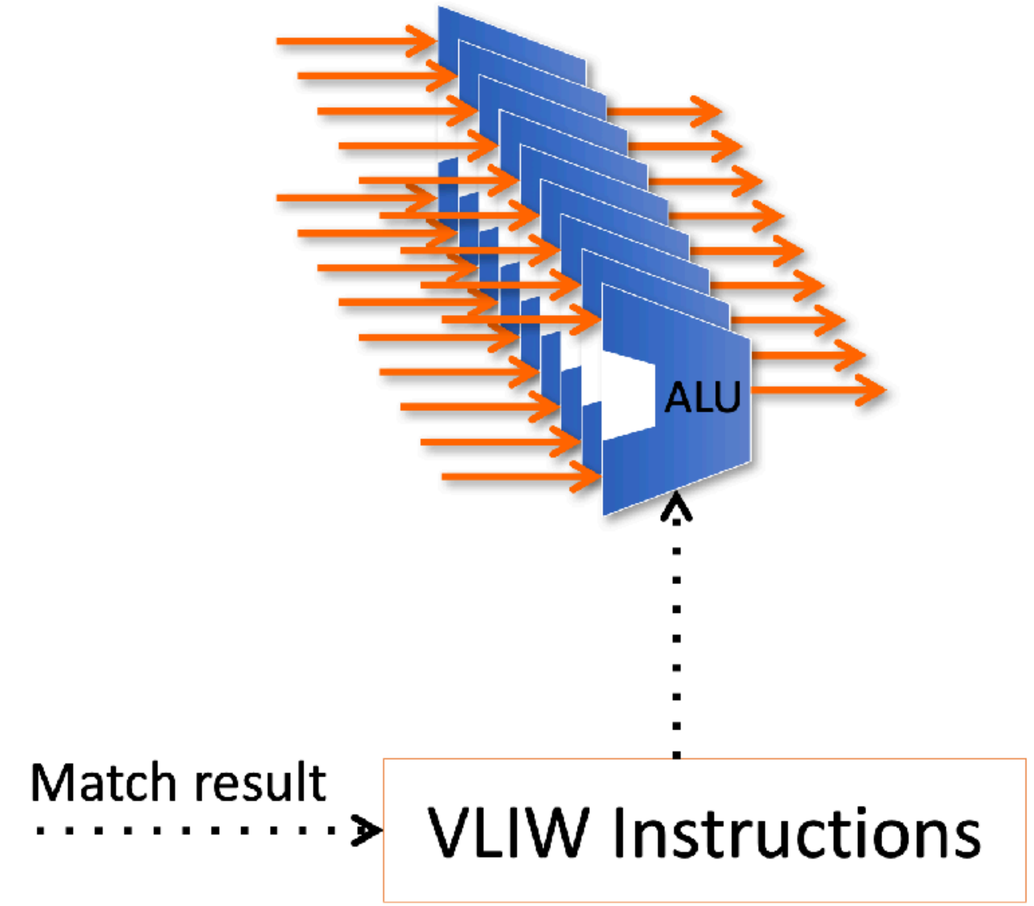


Technique #3: Match/Action Forwarding Model



Modeled as Multiple VLIW CPUs per Stage

- VLIW = Very Long Instruction Word



It Works

Area

Section	Area % of chip	Extra Cost
IO, buffer, queue, CPU, etc	37%	0.0%
Match memory & logic	54.3%	8.0%
VLIW action engine	7.4%	5.5%
Parser + deparser	1.3%	0.7%
Total extra area cost		**14.2%**

Power

Section	Power % of chip	Extra Cost
I/O	26.0%	0.0%
Memory leakage	43.7%	4.0%
Logic leakage	7.3%	2.5%
RAM active	2.7%	0.4%
TCAM active	3.5%	0.0%
Logic active	16.8%	5.5%
Total extra power cost		**12.4%**

How can we use RMT switches to build applications?

Case Study: Fair Queuing

- Enforce fair allocation and isolation at switches:
  - Provide an illustration that every flow has its own queue
  - Proven to have perfect isolation and fairness

- Benefits:
  - Simplify congestion control at the end-host
  - Protect against misbehaving traffic
  - Enable bounded delay guarantees

Fair Queueing without Per-flow Queues

Analysis and Simulation of a Fair Queueing Algorithm

Alan Demers
Srinivasan Keshav†
Scott Shenker
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

Abstract

We discuss gateway queueing algorithms and their role in controlling congestion in datagram networks. A fair queueing algorithm, based on an earlier suggestion by Nagle, is proposed. Analysis and simulations are used to compare this algorithm to other congestion control schemes. We find that fair queueing provides several important advantages over the usual first-come-first-serve queueing algorithm: fair allocation of bandwidth, lower delay for sources using less than their full share of bandwidth, and protection from ill-behaved sources.

often ignored, makes queueing algorithms a crucial component in effective congestion control.

Queueing algorithms can be thought of as allocating three nearly independent quantities: bandwidth (*which* packets get *transmitted*), promptness (*when* do those packets get *transmitted*), and buffer space (*which* packets are *discarded* by the gateway). Currently, the most common queueing algorithm is first-come-first-serve (FCFS). FCFS queueing essentially relegates all congestion control to the sources, since the order of arrival completely determines the bandwidth, promptness, and buffer space allocations. Thus, FCFS inextricably intertwines these three allocation issues. There may indeed be flow

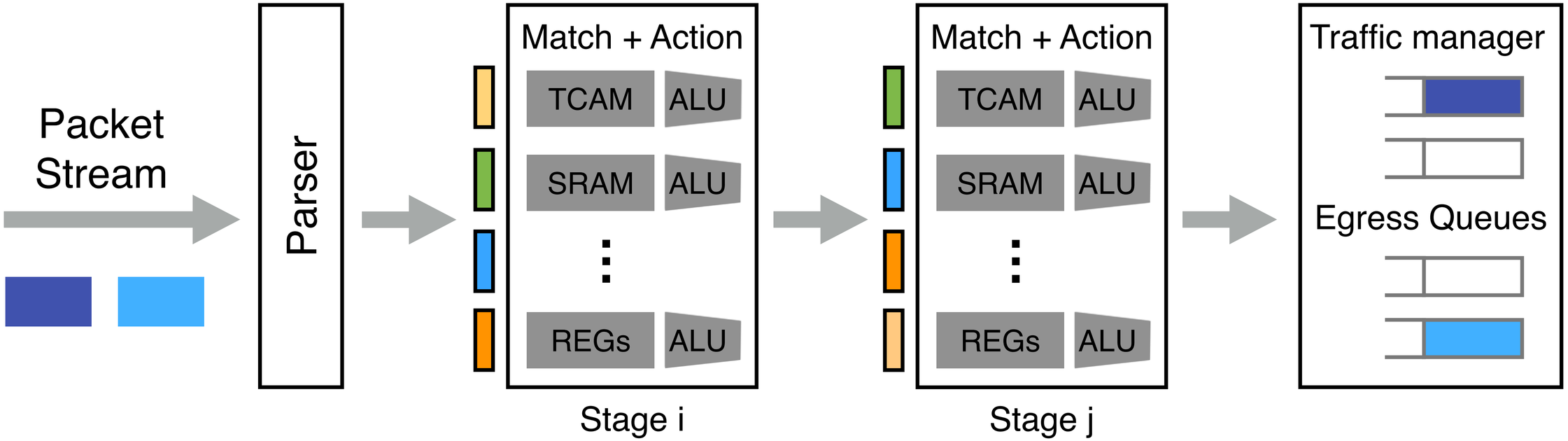
Fair Queueing without Per-flow Queues

- Key idea:
  - Simulate an ideal **round-robin scheme** where each active flow transmits **a single bit of data** every round

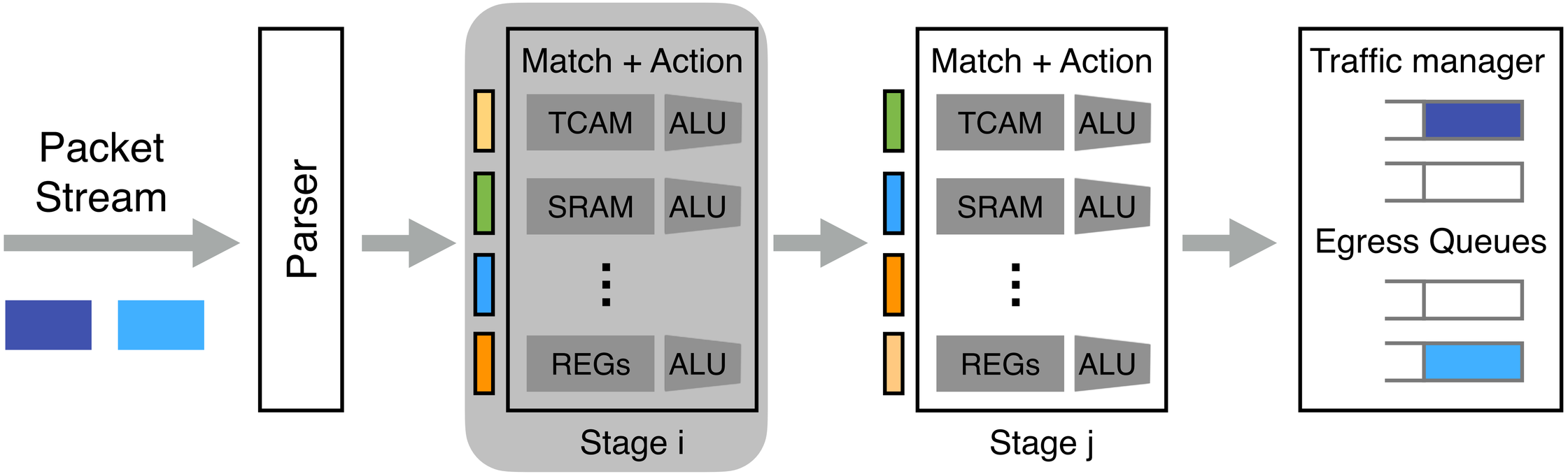
- Challenges:
  - Track the global round number of each active flow
  - Maintain a sorted packet buffer
  - Store and update per-flow counters

How can we use RMT switches to implement fair queueing?

RMT Switch


Packet
Stream
Parser
Match + Action
TCAM
ALU
SRAM
ALU
REGs
ALU
Stage i
Match + Action
TCAM
ALU
SRAM
ALU
REGs
ALU
Stage j
Traffic manager
Egress Queues

RMT Switch



- TCAM/SRAM for matches
- Mutable registers for storing flow states
- ALUs for modifying headers and payloads

- port = lookup(eth.dst_mac)
- counter[ipv4.dst_port]++
- ipv4.ttl = ipv4.ttl - 1

Challenges of Realizing Fair Queueing

- #1: Maintain a sorted packet buffer
  - Requirement: O(logN) insert complexity
  - Constraint: Limited operations per packet

Challenges of Realizing Fair Queueing

- #1: Maintain a sorted packet buffer
  - Requirement: O(logN) insert complexity
  - Constraint: Limited operations per packet

- #2: Store per-flow counters
  - Requirement: Per-flow mutable state
  - Constraint: Limited switch memory

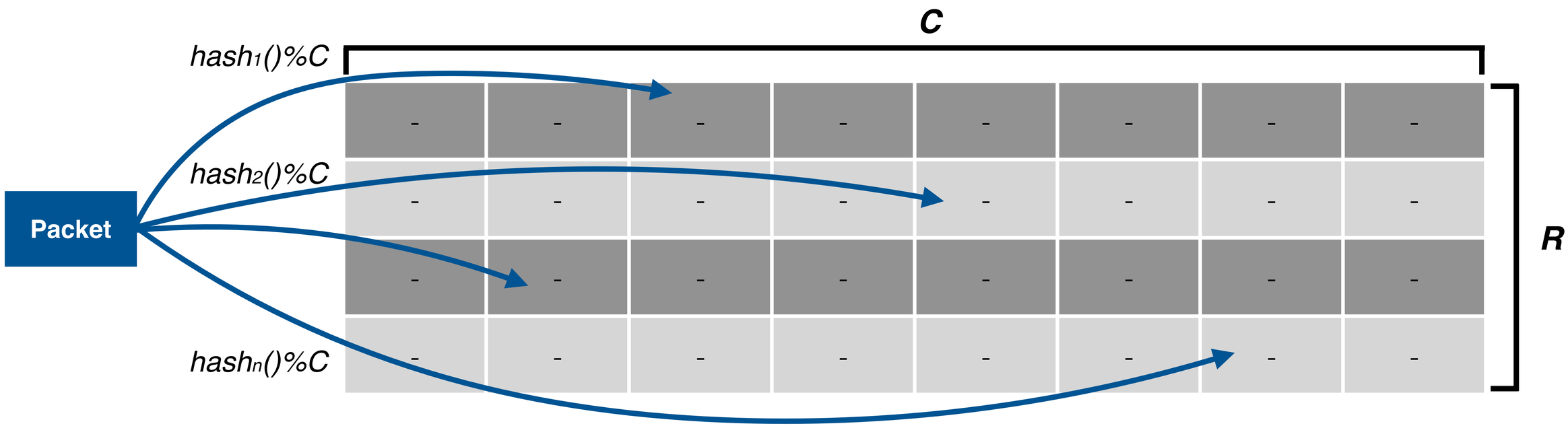
Challenges of Realizing Fair Queueing

- #1: Maintain a sorted packet buffer
  - Requirement: O(logN) insert complexity
  - Constraint: Limited operations per packet
- #2: Store per-flow counters
  - Requirement: Per-flow mutable state
  - Constraint: Limited switch memory
- #3: Access and modify the current round number
  - Requirement: Synchronize state across switch modules
  - Constraint: Limited cross-module communication

Key idea: approximate fair queueing (AFQ)

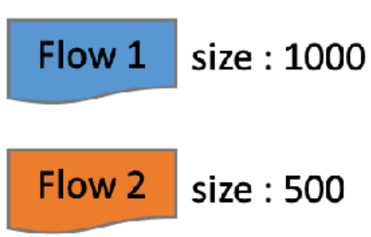
AFQ Technique #1: Store Approx. Flow Counters

- Variation of a count-min sketch to track flow finish round number
  - Update increments all cells; read returns the minimum
  - Never under-estimates, has provable space-accuracy trade-off



An Example

- Use switch read-write registers



0	0	1000	0	0	500	0	0
0	0	0	0	1000	0	0	0
0	1000	0	500	0	0	0	0
0	0	0	0	0	500	1000	0

`min (0, 1000, 0, 0) = 0 + 500 = 500`

Read Counter

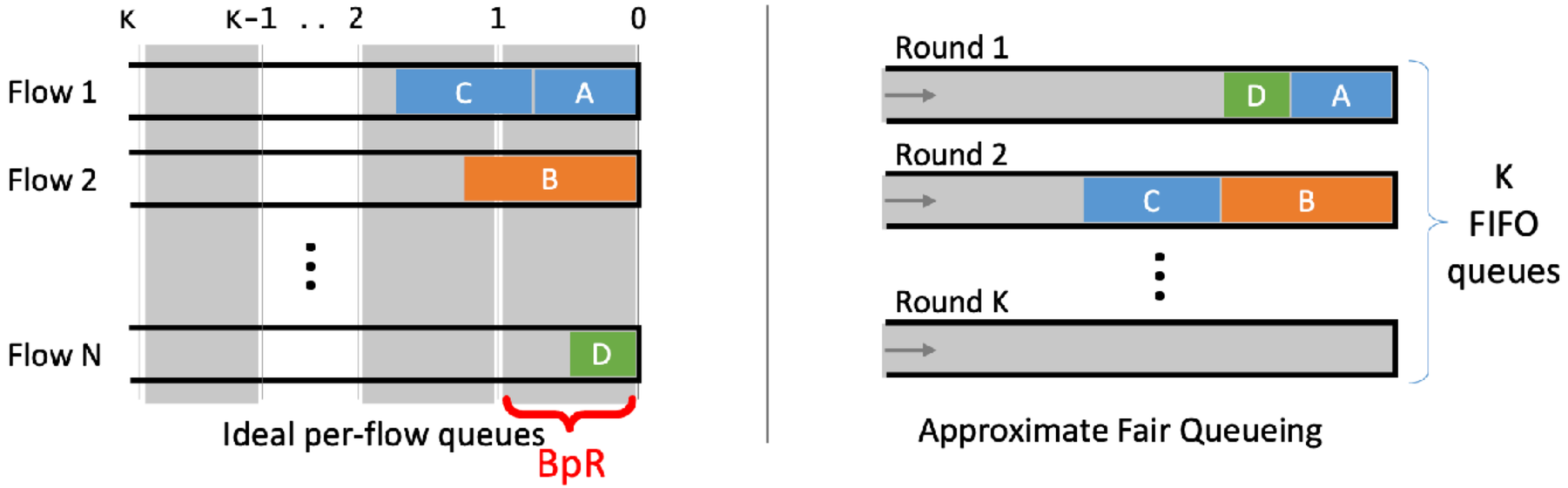
- Find the minimum of all cells
- Bytes sent = minimum + pkt.size

Update Counter

- Increment all cells upto new value
- $cell^{x,y}$ = max ($cell^{x,y}$, new value)

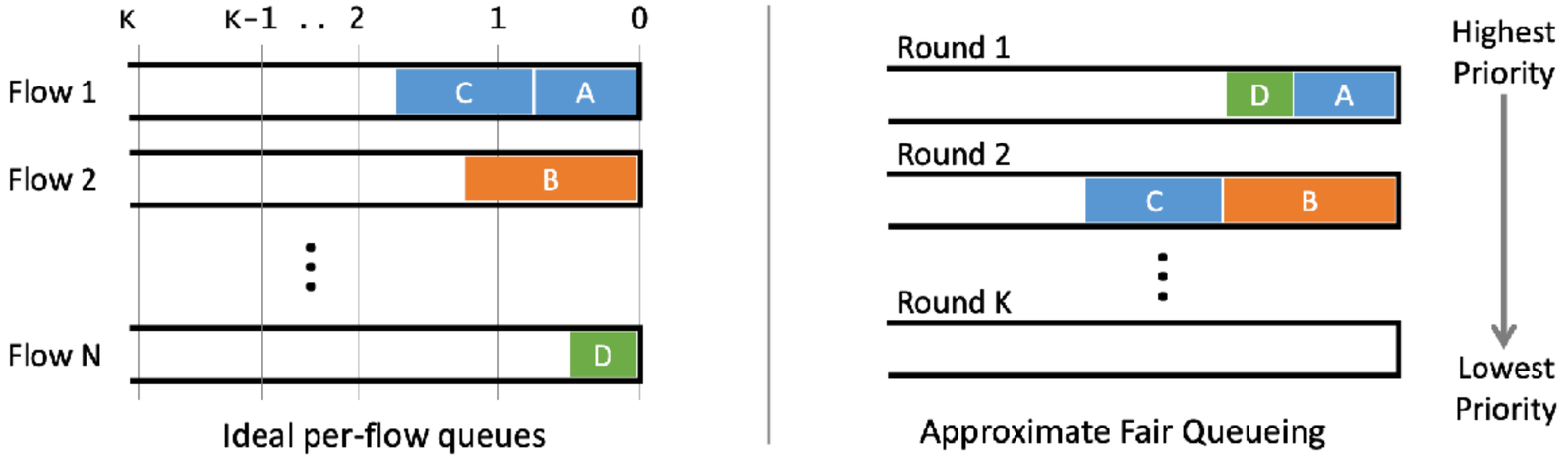
AFQ Technique #2: Buffer Packets in Approx. Sorted Order

- Coarse rounds:
  - Flows transmit a quantum of bytes per round (BpR)
  - For each packet, outgoing round number = byte sent / BpR



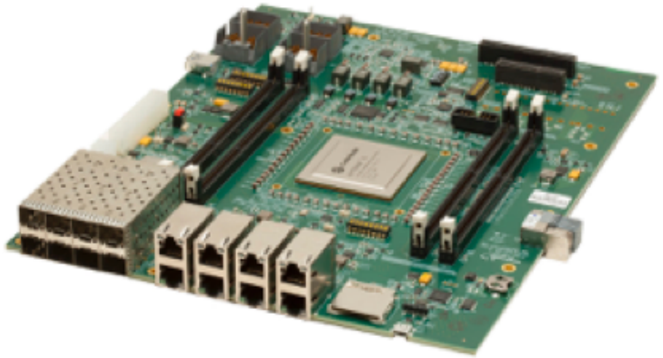
AFQ Technique #3: Rotating Strict Priority Scheduler

- Approximate sorted buffer
  - Drain queue with the lowest round number till it is empty
  - Push queue to lowest priority; increment round number by 1



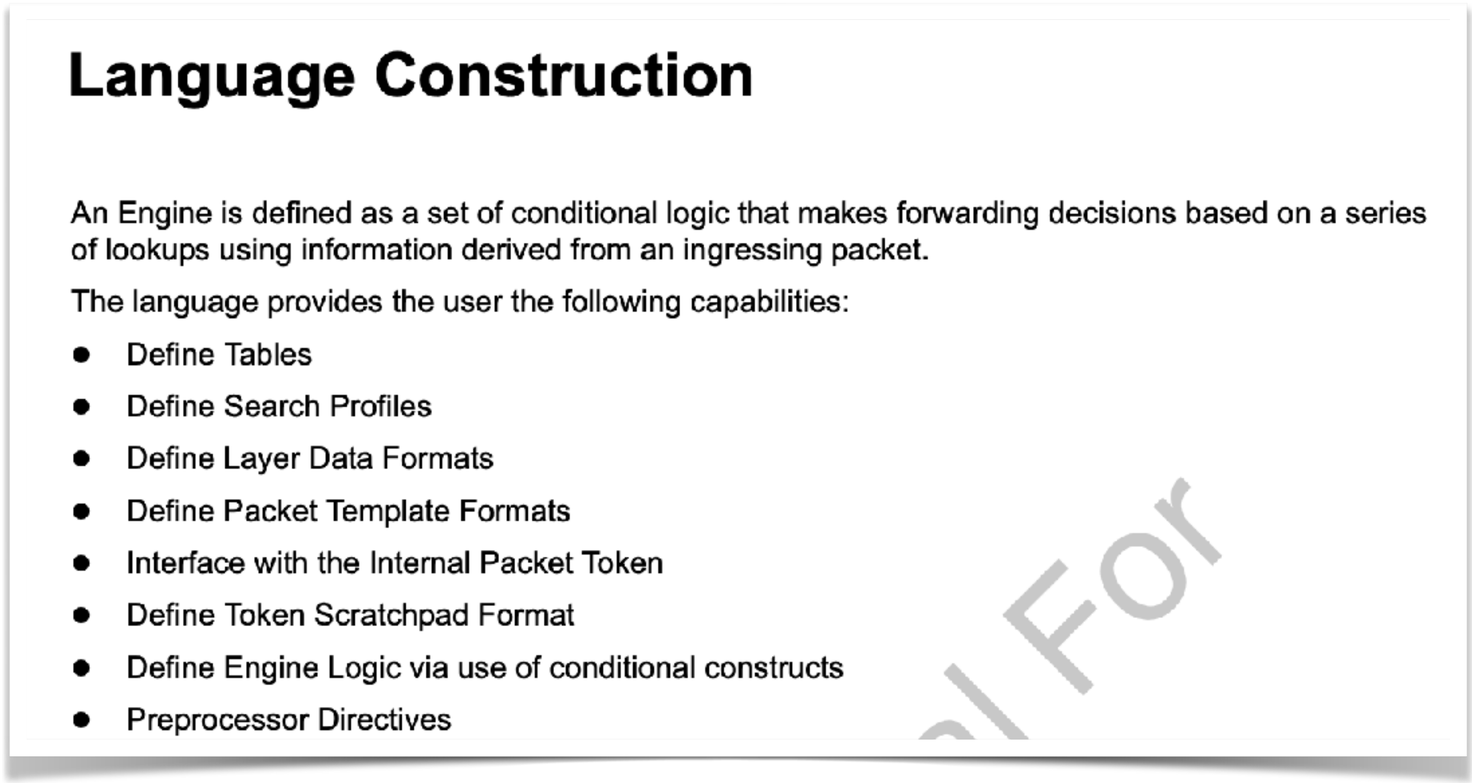
AFQ Implememention

- Cavium XPliant switch and networking processors
  - Xpliant A0 and B0 engineering samples in 2016
  - Extensively collaborated with Cavium engineers (Kishore Atreya)



Early RMT Switch Development

- Cavium XPliant provides a DSL called XPC



Language Construction

An Engine is defined as a set of conditional logic that makes forwarding decisions based on a series of lookups using information derived from an ingressing packet.

The language provides the user the following capabilities:

- Define Tables
- Define Search Profiles
- Define Layer Data Formats
- Define Packet Template Formats
- Interface with the Internal Packet Token
- Define Token Scratchpad Format
- Define Engine Logic via use of conditional constructs
- Preprocessor Directives

Early RMT Switch Development

Parser Graph

Layer: Decision Point Offset (Bytes)	Pre-defined Metadata Bits	Field Size (bits) For Canonical Format	Canonical Format	DecPoint Value, DecPoint Mask, Metadata (Comma Separated)				
				field	field: X	field: Y	field: 0	field: 1
ETHERNET: 12, 16, 20, 24, 28, NA	HW: 0-11							
	LayerId: 21|22	48	MAC DA					
		48	MAC SA					
		64	E-Tag	0x000000000000893F, 0x0000FFFF00000000, 12				
		32	S-Tag	0x000088A8, 0x00000000, 13				
		32	C-Tag	0x00008100, 0x00000000, 14				
		16	EtherType	NEXT_LAYER_INDICATOR				
		Next Possible Layer	Available Formats	NEXT_LAYER_INDICATOR: EtherType				
		ARP	{}, {C-Tag: NA}, {S-TAG: NA}, {S-TAG: NA, C-Tag: NA}, {E-TAG: NA}, {E-TAG: NA, C-TAG: NA},	0x0806, 0x0000, 15				
		IPv4	{}, {C-Tag: NA}, {S-TAG: NA}, {S-TAG: NA, C-Tag: NA}, {E-TAG: NA}, {E-TAG: NA, C-TAG: NA},	0x0800, 0x0000, 16				
		IPv6	{}, {C-Tag: NA}, {S-TAG: NA}, {S-TAG: NA, C-Tag: NA}, {E-TAG: NA}, {E-TAG: NA, C-TAG: NA},	0x86DD, 0x0000, 17				
		MPLS_UC	{}, {C-Tag: NA}, {S-TAG: NA}, {S-TAG: NA, C-Tag: NA}, {E-TAG: NA}, {E-TAG: NA, C-TAG: NA},	0x8847, 0x0000, 18				
		MPLS_MC	{}, {C-Tag: NA}, {S-TAG: NA}, {S-TAG: NA, C-Tag: NA}, {E-TAG: NA}, {E-TAG: NA, C-TAG: NA},	0x8848, 0x0000, 19				
		FCoE	{}, {C-Tag: NA}, {S-TAG: NA}, {S-TAG: NA, C-Tag: NA}, {E-TAG: NA}, {E-TAG: NA, C-TAG: NA},	0x8906, 0x0000, NA				
		CNM	{}, {C-Tag: NA}, {S-TAG: NA}, {S-TAG: NA, C-Tag: NA}, {E-TAG: NA}, {E-TAG: NA, C-TAG: NA},	0x22e7, 0x0000, NA				
		NSH	{}, {C-Tag: NA}, {S-TAG: NA}, {S-TAG: NA, C-Tag: NA}, {E-TAG: NA}, {E-TAG: NA, C-TAG: NA},	0x894F, 0x0000, 24				
		PTP	{}, {C-Tag: NA}, {S-TAG: NA}, {S-TAG: NA, C-Tag: NA}, {E-TAG: NA}, {E-TAG: NA, C-TAG: NA},	0x88F7, 0x0000, 20				
		LAYER_NA	{}, {C-Tag: NA}, {S-TAG: NA}, {S-TAG: NA, C-Tag: NA}, {E-TAG: NA}, {E-TAG: NA, C-TAG: NA},	0x0000, 0xFFFF, NA				

Table Graph

Direct Access, TCAM, or LPM tables

```
TABLE <name> (ID = <table_id>, depth = <table_depth>, type = DIRECT_ACCESS_TABLE)
{
    FIELD one:  1;
    FIELD two:  3;
    FIELD three:10;
    FIELD four: 10;
    STRUCT str;
};
TABLE <name> (ID = <table_id>, depth = <table_depth>, type = TCAM_TABLE)
{
    FIELD one:  1;
    FIELD two:  3;
    FIELD three:10;
    FIELD four: 10;
    STRUCT str;
};
TABLE <name> (ID = <table_id>, depth = <table_depth>, type = LPM_TABLE)
{
    FIELD one:  1;
    FIELD two:  3;
    FIELD three:10;
    FIELD four: 10;
    STRUCT str;
};
```

Action Logic

```
OFIT
{
    if( (bdMacSAMode != XP_BRIDGE_SA_LOOKUP_DISABLED) && (iVifStpState !=
STATE_BLOCKING) &&
            (fdbTableSa.sorry == false))
    {
        if(fdbTableSa.hit == false)
        {
            mirrorSession = XP_MIRROR_SESSION_NEW_ADDR;
            scOut.fdbNewAddr = true;
            scOut.saHitData  = XP_BRIDGE_MAC_SA_NEW;
        }
        else
        if((fdbTableSa.hit == true)
```

P4 Language

- Programming Protocol-Independent Packet Processors
  - Originally described in a 2014 SIGCOMM CCR paper

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 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.

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

Table 1: Fields recognized by the OpenFlow standard

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 past five years, the specification has grown increasingly more complicated (see Table 1), with many more header fields and 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.

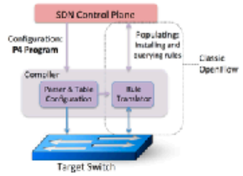


Figure 1: P4 is a language to configure switches.

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 Programming Protocol-independent Packet Processors (P4). Figure 1 shows the relationship between P4—used to configure a switch, telling it how packets are to be processed—and existing APIs (such as OpenFlow) that are designed to populate the forwarding tables in fixed function switches. P4 raises the level of abstraction for programming the network, and can serve as a

```
1  #include <core.p4>
2  #include <v1model.p4>
3
4  struct headers { ethernet_t eth;
5                   ipv4_t  ipv4;
6  }
7
8  header ethernet_t {
9      bit<48> dstAddr;
10     bit<48> srcAddr;
11     bit<16> etherType;
12 }
13
14 header ipv4_t { bit<8> ttl;
15                 bit<32> dstAddr;
16                 [...]
17 }
18
19 V1Switch(p = ParserImpl(),
20           ig = ingress(),
21           dep = DeparserImpl()) main;
22
23 parser ParserImpl [...] { [...] }
24 control ingress(
25  inout headers h,
26  out standard_metadata_t sm) {
27
28 action on_miss() { }
29
30 action nexthop(bit<9> port) {
31  sm.egress_port = port;
32  h.ipv4.ttl     = h.ipv4.ttl - 8w1;
33 }
34
35 table fib {
36  actions = { on_miss; nexthop; }
37  key = { h.ipv4.dstAddr : exact; }
38  size = 131072;
39 }
40
41 table fib_lpm {
42  actions = { on_miss; nexthop; }
43  key = { h.ipv4.dstAddr : lpm; }
44  size = 16384;
45 }
46
47 apply {
48  if (h.ipv4.isValid()) {
49     switch (fib.apply().action_run) {
50      on_miss: { fib_lpm.apply(); }
51     }
52   }
53  }
54 }
55 // end of control ingress
56
57
58 control DeparserImpl(
59   packet_out packet,
60   in headers hdrs) {
61
62  apply {
63    packet.emit(hdrs.eth);
64    packet.emit(hdrs.ipv4);
65  }
66 }
67
68
69
```

In-Network Computing

- Transport layer: bandwidth allocation, telemetry, etc.
- Application layer: caching, scheduling, etc.

In-Network Computing

- Transport layer: bandwidth allocation, telemetry, etc.
- Application layer: caching, scheduling, etc.

But the killer use case (application) is unclear!

RMT Switch Today

- Seems dead,
  - Cavium was acquired by Marvell in 2018, but Xpliant Switch products are ceased in 2018
  - Barefoot was acquired by Intel in 2019, but Tofino Switch products are ceased in 2023

RMT Switch Today

- Seems dead,
  - Cavium was acquired by Marvell in 2018, but Xpliant Switch products are ceased in 2018
  - Barefoot was acquired by Intel in 2019, but Tofino Switch products are ceased in 2023

- But not,
  - NVIDIA/Mellanox Spectrum switches: microcode programming
  - Juniper Trio-based switches: microcode programming
  - Cisco dRMT-like (sigcomm’17) model becomes more friendly

Summary

- Today
  - Programmable Switch

- Next
  - SDN and programmable networks (III)