# CS244

## Advanced Topics in Networking


## Lecture 7: Programmable Forwarding

Nick McKeown

"Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN"
[Pat Bosshart et al. 2013]

Spring 2020

# Context

Pat Bosshart
At the time: TI (Texas Instruments)
Architect of first LISP CPU and 1GHz DSP

+ Others from TI

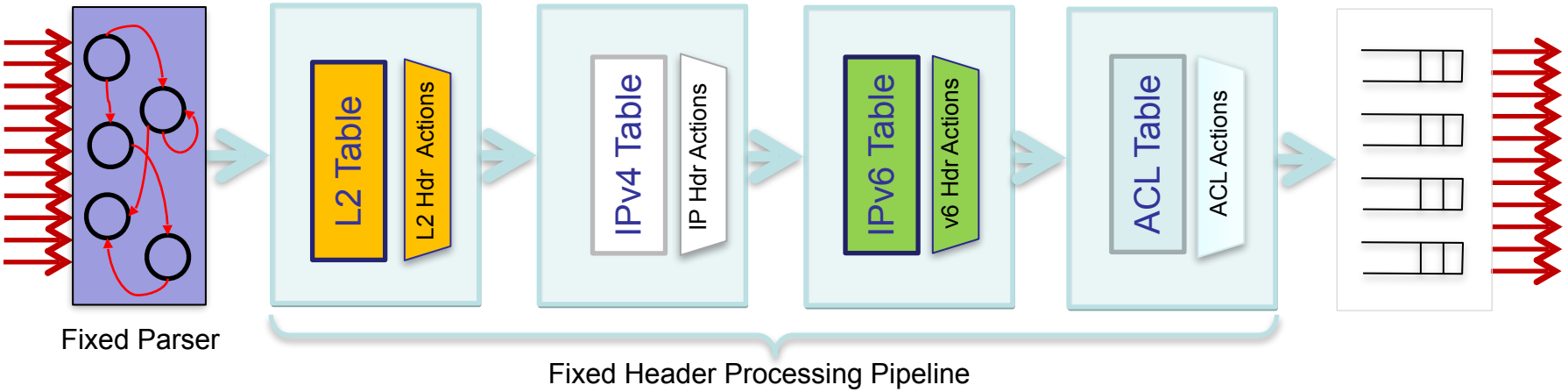
George Varghese
At the time: MSR
Today: Professor at UCLA

+ Others from Stanford

At the time the paper was written (2012)…

- Fastest switch ASICs were fixed function, around 1Tb/s
- Lots of interest in “disaggregated” switches for large data-centers

# Switch with fixed function pipeline



# You said

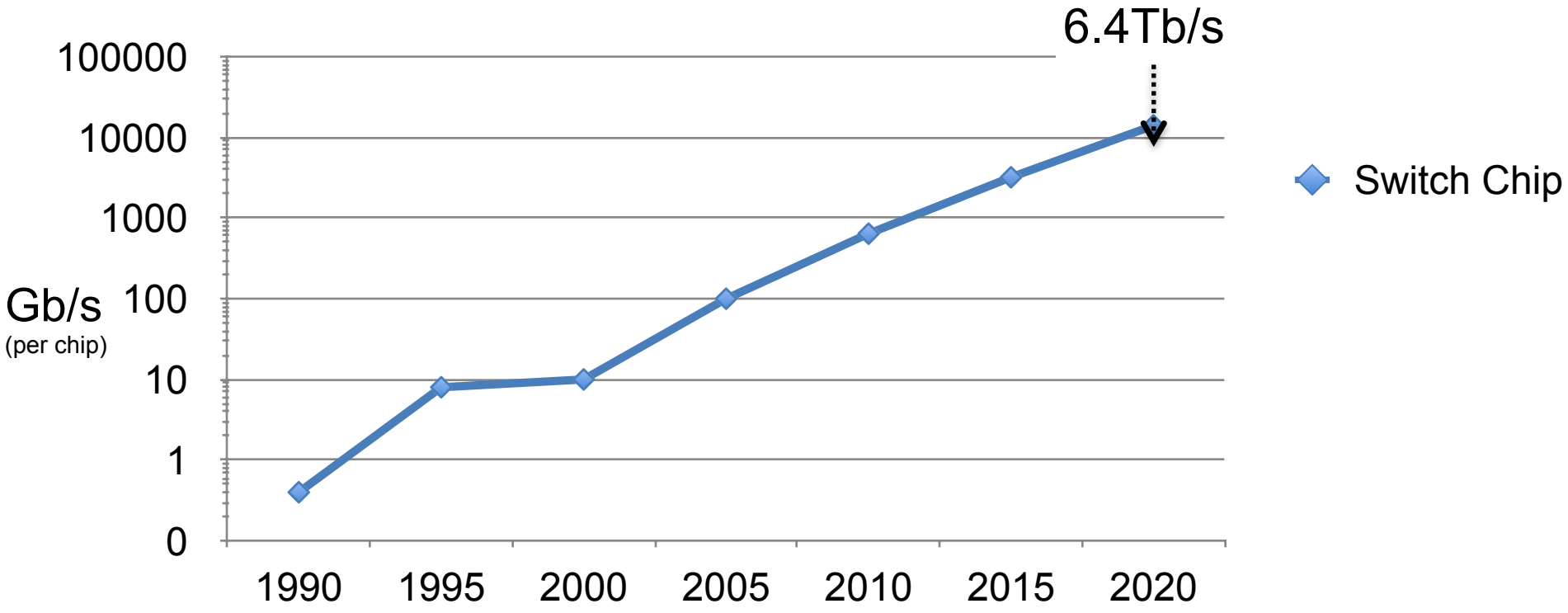
Amalee Wilson

There’s a key phrase in the abstract, “contrary to concerns within the community,” and I’m curious about what those concerns are.

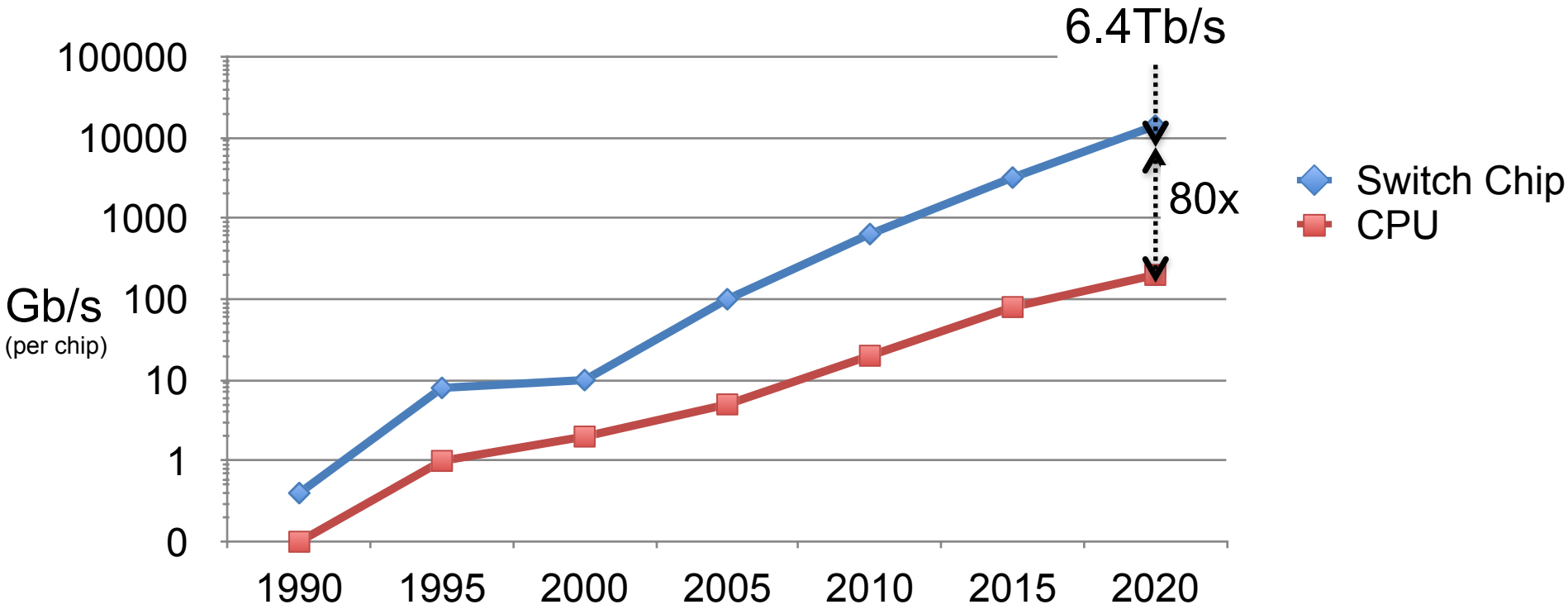
“Programmable switches run 10x slower, consume more power and cost more.”

Conventional wisdom in 2010

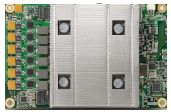
# Packet Forwarding Speeds



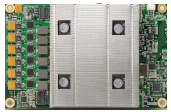
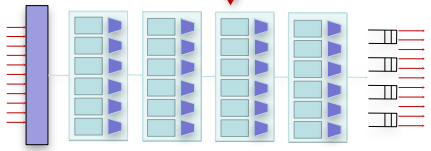
# Packet Forwarding Speeds



# Domain Specific Processors

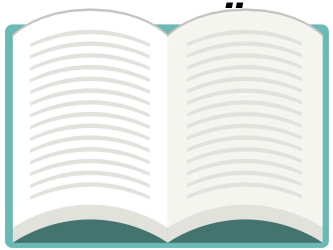
| Computers | Graphics | Signal Processing | Machine Learning | | Networking |
|---|---|---|---|---|---|
| Java | OpenCL | Matlab | TensorFlow | >> | Language |
| Compiler | Compiler | Compiler | Compiler | | Compiler |
| ↓ | ↓ | ↓ | ↓ | | ↓ |
|  |  |  |  | | ? |
| CPU | GPU | DSP | TPU | | |

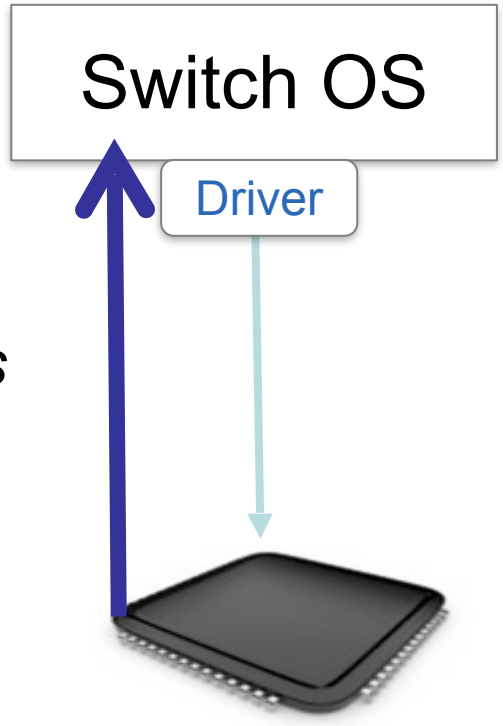
# Domain Specific Processors

| Computers | Graphics | Signal Processing | Machine Learning | | Networking |
|---|---|---|---|---|---|
| Java | OpenCL | Matlab | TensorFlow | >> | P4 |
| Compiler | Compiler | Compiler | Compiler | | Compiler |
|  |  |  |  | |  |
| CPU | GPU | DSP | TPU | | PISA *aka* "RMT" |

# Network systems tend to be designed “bottom-up”

*“This is how I process packets*





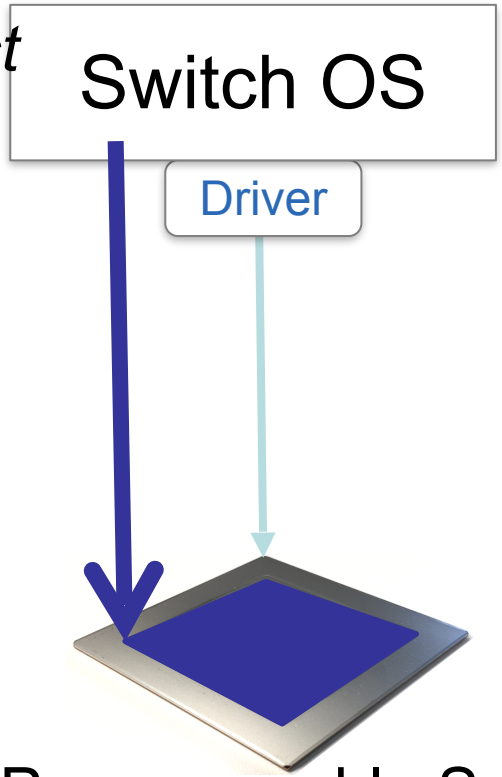

Fixed-function switch

# What if they could be programmed “top-down”?

*“This is precisely how you must process packets”*

```
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);
}
```



# You said

Wantong Jiang:

At the end of the paper, the authors mention FPGA and claim that they are too expensive. This paper was published in 2013 and I wonder if it's still the case nowadays.

Firas Abuzaid:

The paper mentions that FPGAs are too expensive to be considered. Now that FPGAs have become more widely available, could they be used instead of RMTs?

# The RMT design [2013]

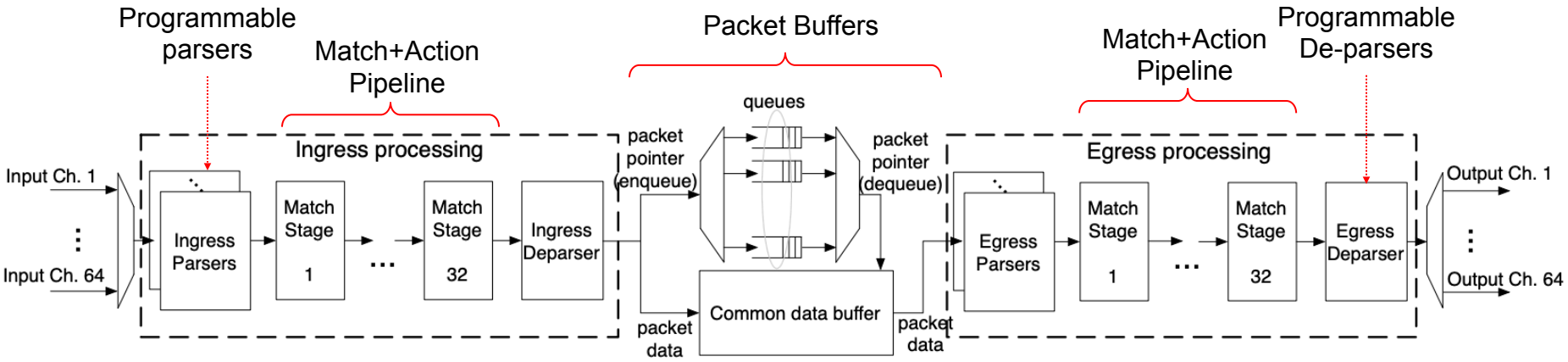
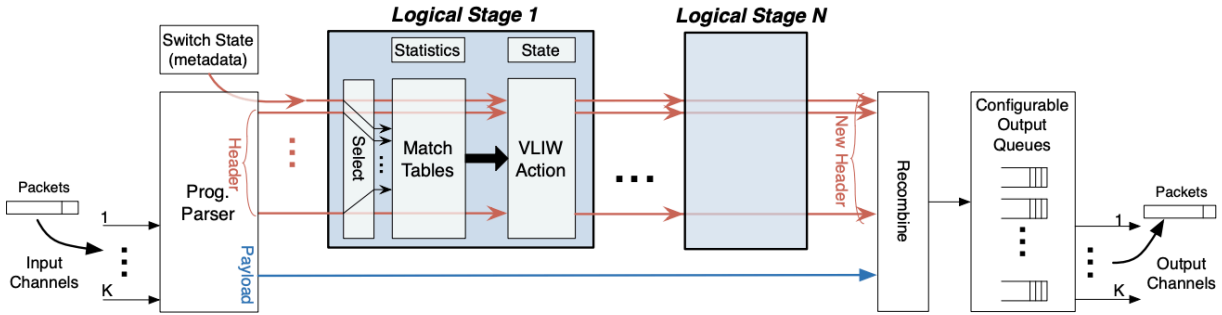
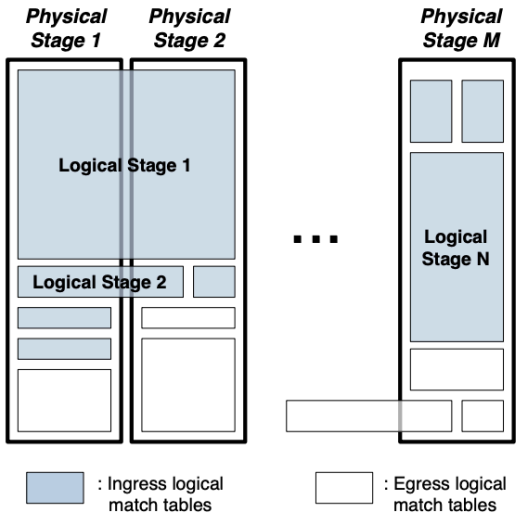


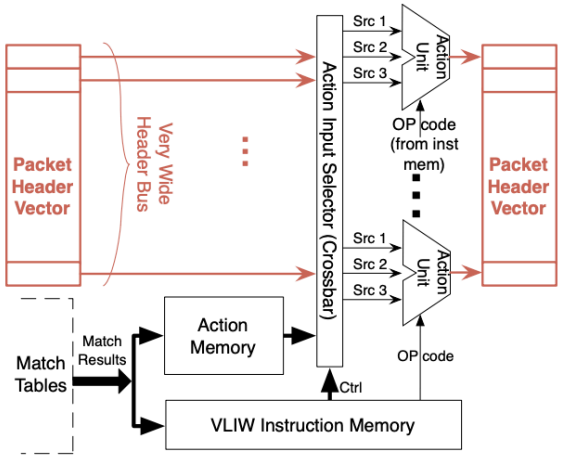

Figure 3: Switch chip architecture.




(a) RMT model as a sequence of logical Match-Action stages.




(b) Flexible match table configuration.




(c) VLIW action architecture.

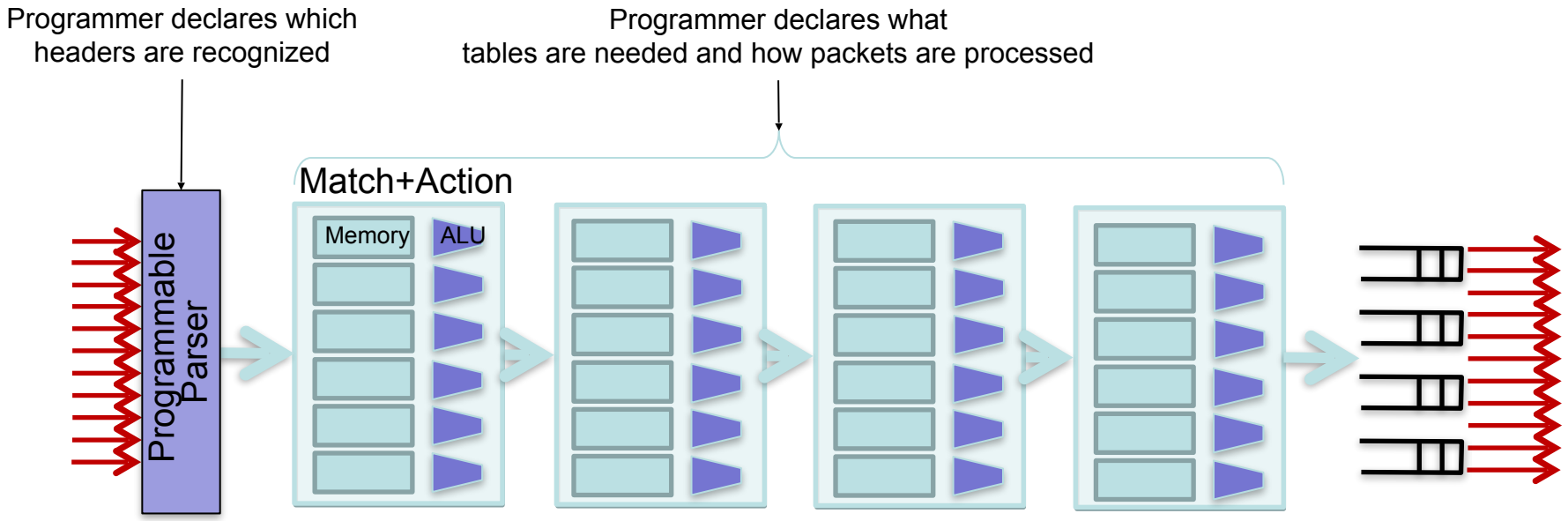
Figure 1: RMT model architecture.

# You said

Will Brand

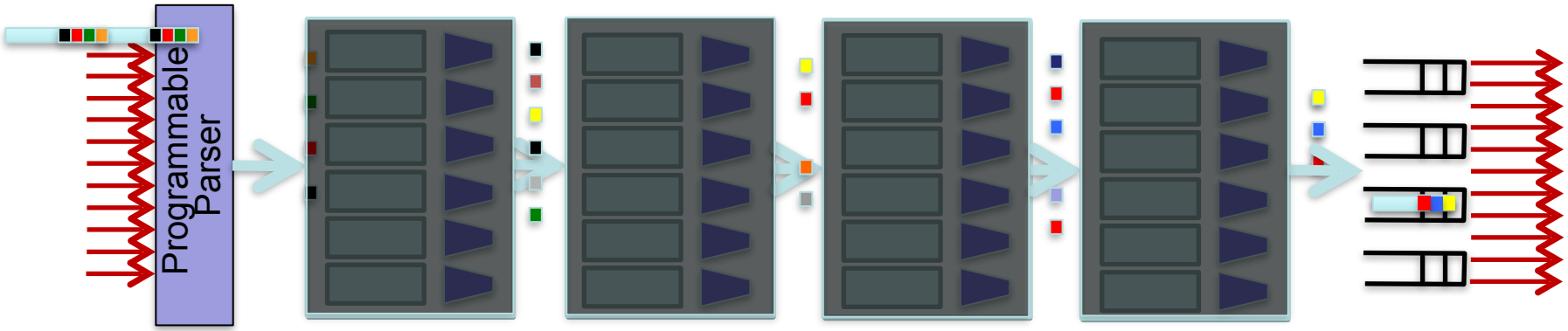
[W]hat goes into designing the vocabulary of a RISC instruction set? Since I can't just try to prove the instructions are Turing-complete, and the instruction set doesn't have the kind of specification I might expect from a general-purpose language, I find it difficult to "trust" that Table 1 encapsulates a reasonable portion of the actions we might want to make possible…

# PISA: Protocol Independent Switch Architecture

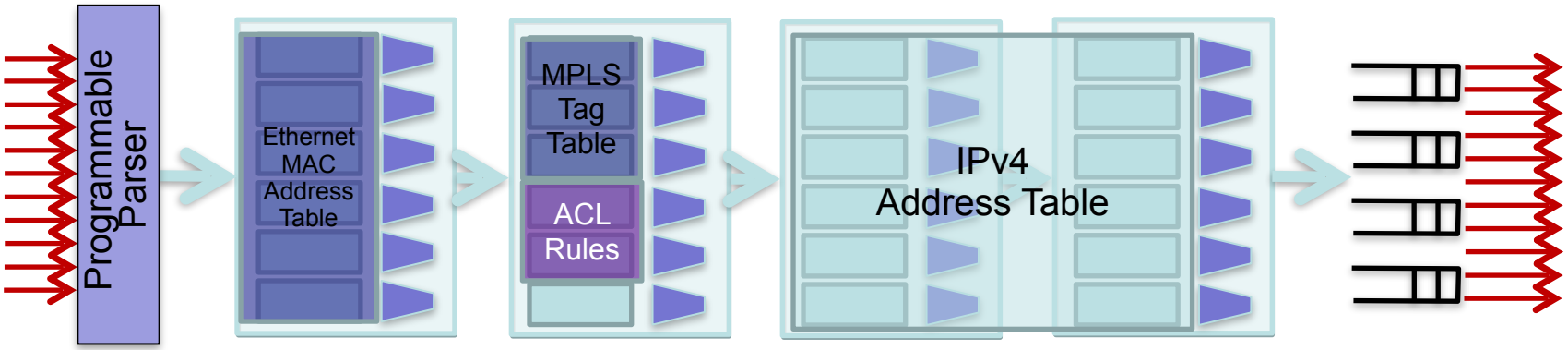



All stages are identical. A “compiler target”.

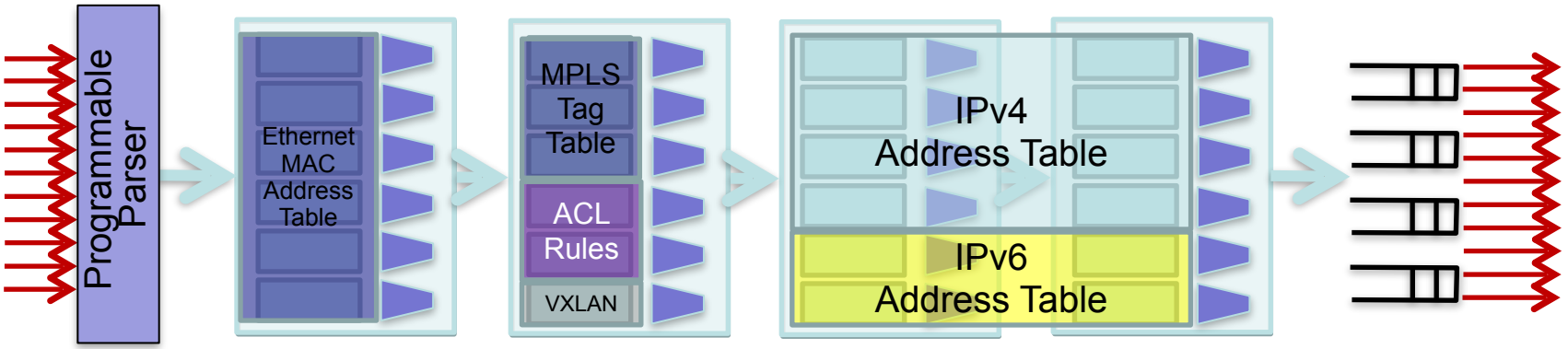
# PISA: Protocol Independent Switch Architecture



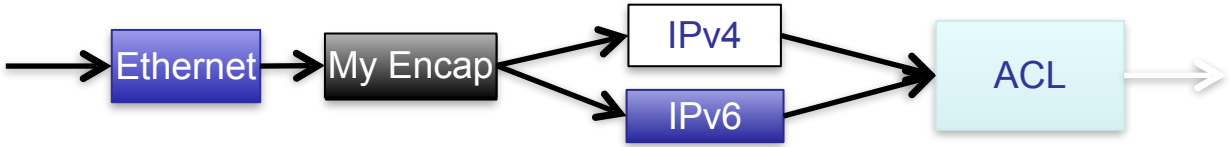
# PISA: Protocol Independent Switch Architecture

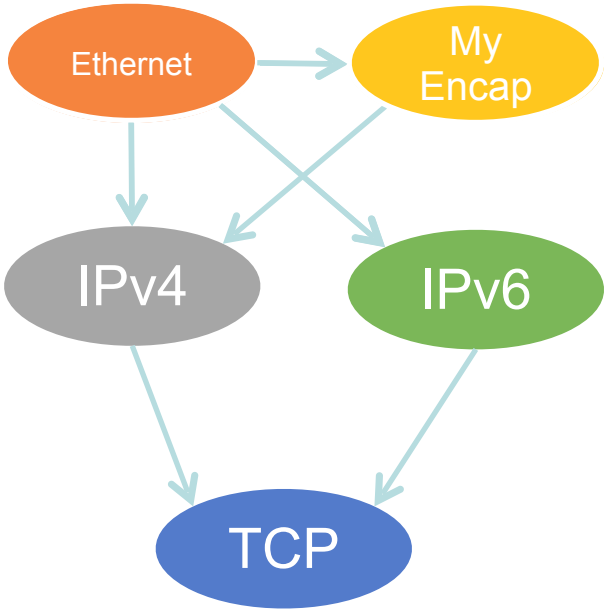


# PISA: Protocol Independent Switch Architecture



# P4 program example: Parsing Headers



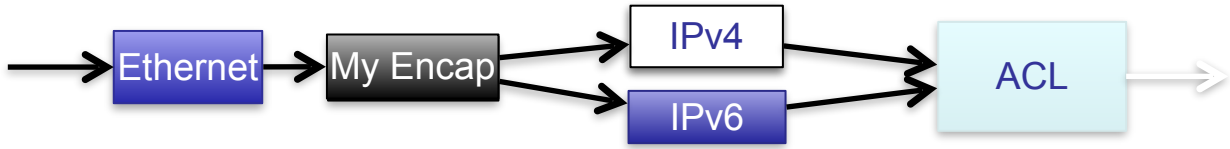




```
header_type ethernet_t {
        fields {
                dstAddr : 48;
                srcAddr : 48;
                etherType : 16;
        }
}

parser parse_ethernet {
  extract(ethernet);
  return select(latest.etherType) {
    0x8100 : parse_my_encap;
    0x800  : parse_ipv4;
    0x86DD : parse_ipv6;
  }
}

                bar : 3;
                baz : 4;
                qux : 4;
                next_protocol : 4;
        }
}
```

# P4 program example




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

```
control ingress
{
        apply(l2);
        apply(my_encap);
        if (valid(ipv4) {
                apply(ipv4_lpm);
        } else {
                apply(ipv6_lpm);
        }
        apply(acl);
}
```

```
action set_next_hop(nhop_ipv
{
        modify_field(metadata.nhop_ipv4_addr, nhop_ipv4_addr);
        modify_field(standard_metadata.egress_port, port);
        add_to_field(ipv4.ttl, -1);
}
```

# How programmability is used

1. Reducing complexity

# Reducing complexity



**IPv4 and IPv6 routing**
- Unicast Routing
  - Routed Ports & SVI
  - VRF
- Unicast RPF
  - Strict and Loose
- Multicast
  - PIM-SM/DM & PIM-Bidir

**Ethernet switching**
- VLAN Flooding
- MAC Learning & Aging
- STP state
- VLAN Translation

**Load balancing**
- LAG
- ECMP & WCMP
- Resilient Hashing
- Flowlet Switching

**Fast Failover**
– LAG & ECMP

**Tunneling**
- IPv4 and IPv6 Routing & Switching
  - IP-in-IP (6in4, 4in4)
  - VXLAN, NVGRE, GENEVE & GRE
  - Segment Routing, ILA

**MPLS**
- LER and LSR
- IPv4/v6 routing (L3VPN)
- L2 switching (EoMPLS, VPLS)
- MPLS over UDP/GRE

**ACL**
- MAC ACL, IPv4/v6 ACL, RACL
- QoS ACL, System ACL, PBR
- Port Range lookups in ACLs

**QOS**
- QoS Classification & marking
- Drop profiles/WRED
- RoCE v2 & FCoE
- CoPP (Control plane policing)

**NAT and L4 Load Balancing**

**Security Features**
- Storm Control, IP Source Guard

**Monitoring & Telemetry**
- Ingress Mirroring and Egress Mirroring
- Negative Mirroring
- Sflow
- INT

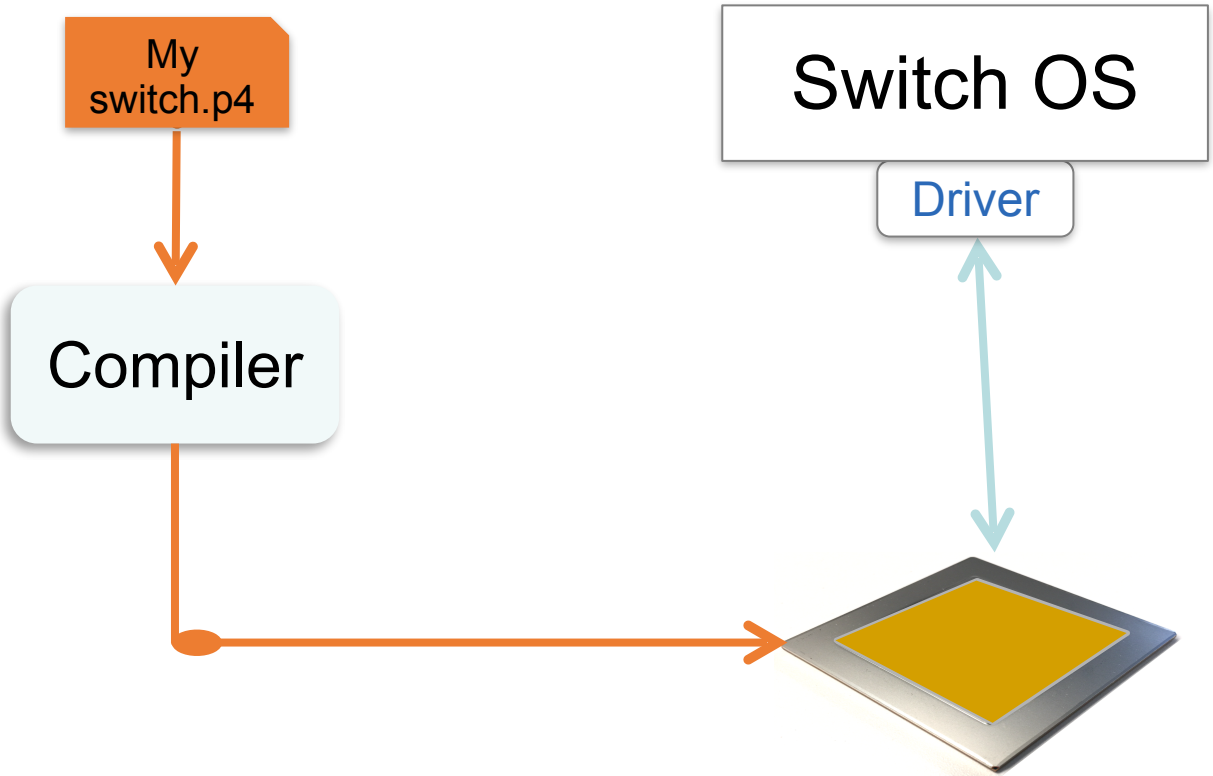
**Counters**
- Route Table Entry Counters
- VLAN/Bridge Domain Counters
- Port/Interface Counters

**Protocol Offload**
- BFD, OAM

**Multi-chip Fabric Support**
- Forwarding, QOS

h

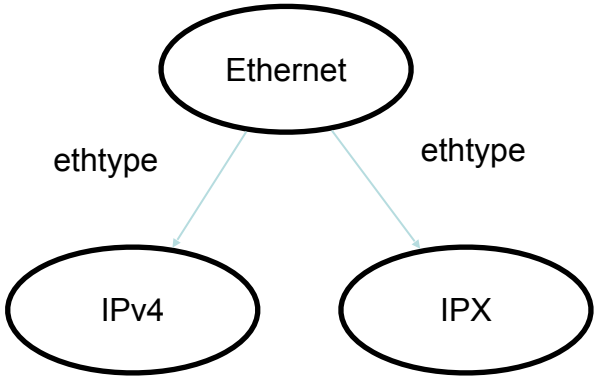
# Reducing complexity



# How programmability is used

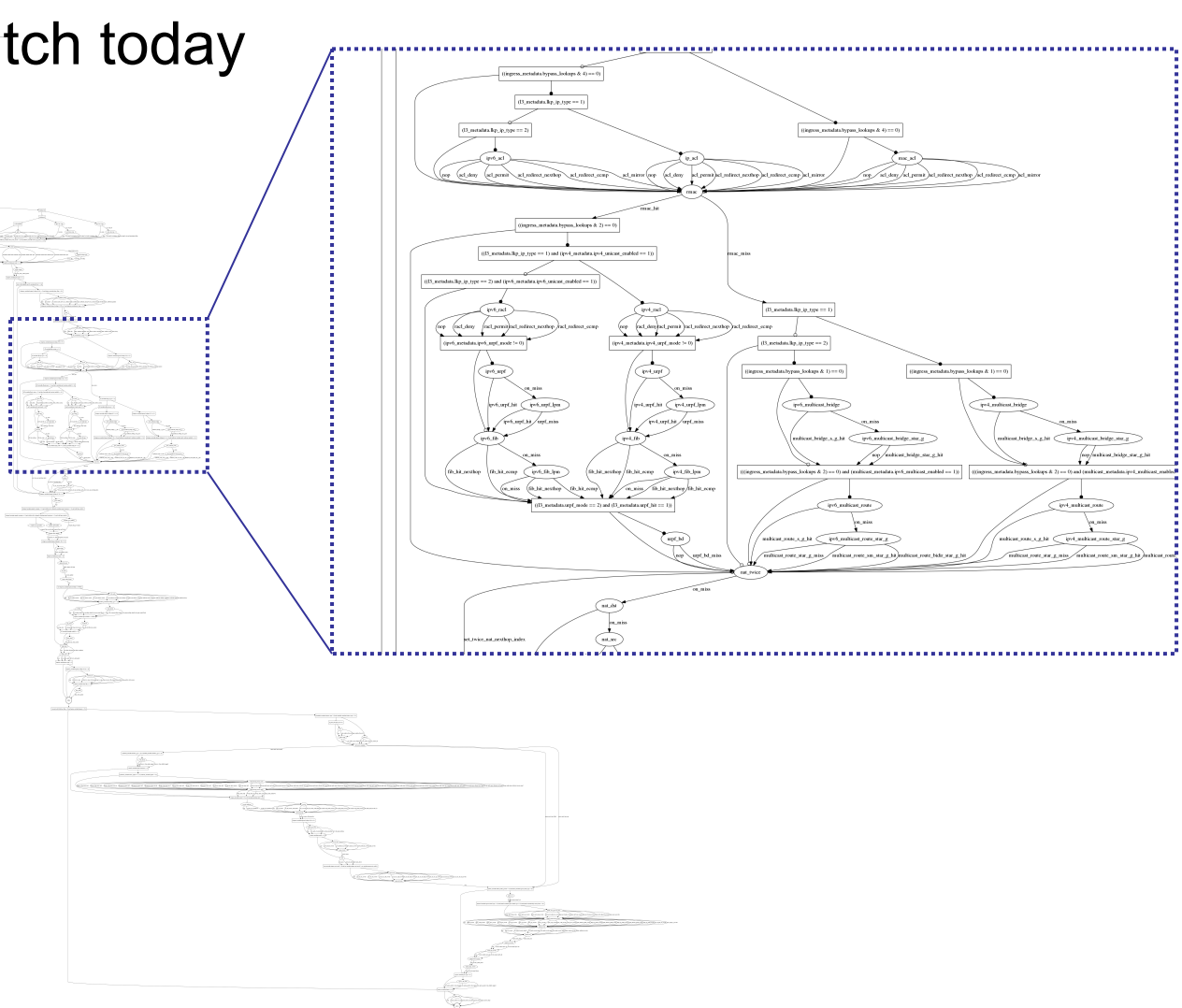
2 Adding new features

# Protocol complexity 20 years ago



# Datacenter switch today

switch.p4



# Example new features

1. New encapsulations and tunnels
2. New ways to tag packets for special treatment
3. New approaches to routing: e.g. source routing in DCs
4. New approaches to congestion control
5. New ways to process packets: e.g. ticker-symbols

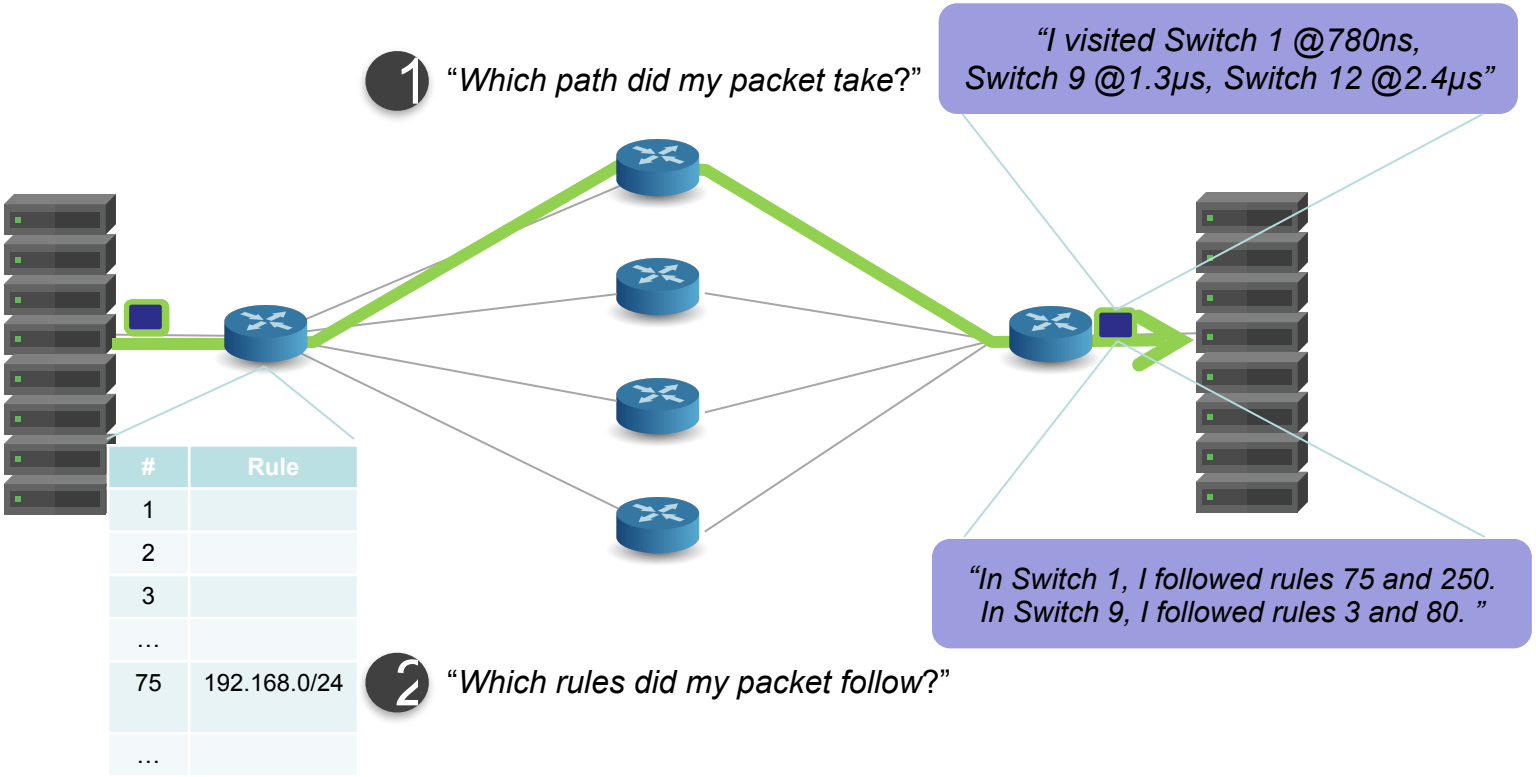
# Example new features

1. Layer-4 Load Balancer[1]
   - Replace 100 servers or 10 dedicated boxes with one programmable switch
   - Track and maintain mapping for 5-10 million http flows
2. Fast stateless firewall
   - Add/delete and track 100s of thousands of new connections per second
3. Cache for Key-value store[2]
   - Memcache in-network cache for 100 servers
   - 1-2 billion operations per second

[1] “SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs.” Rui Miao et al. Sigcomm 2017.
[2] “NetCache: Balancing Key-Value Stores with Fast In-Network Caching”, Xin Jin et al. SOSP 2017

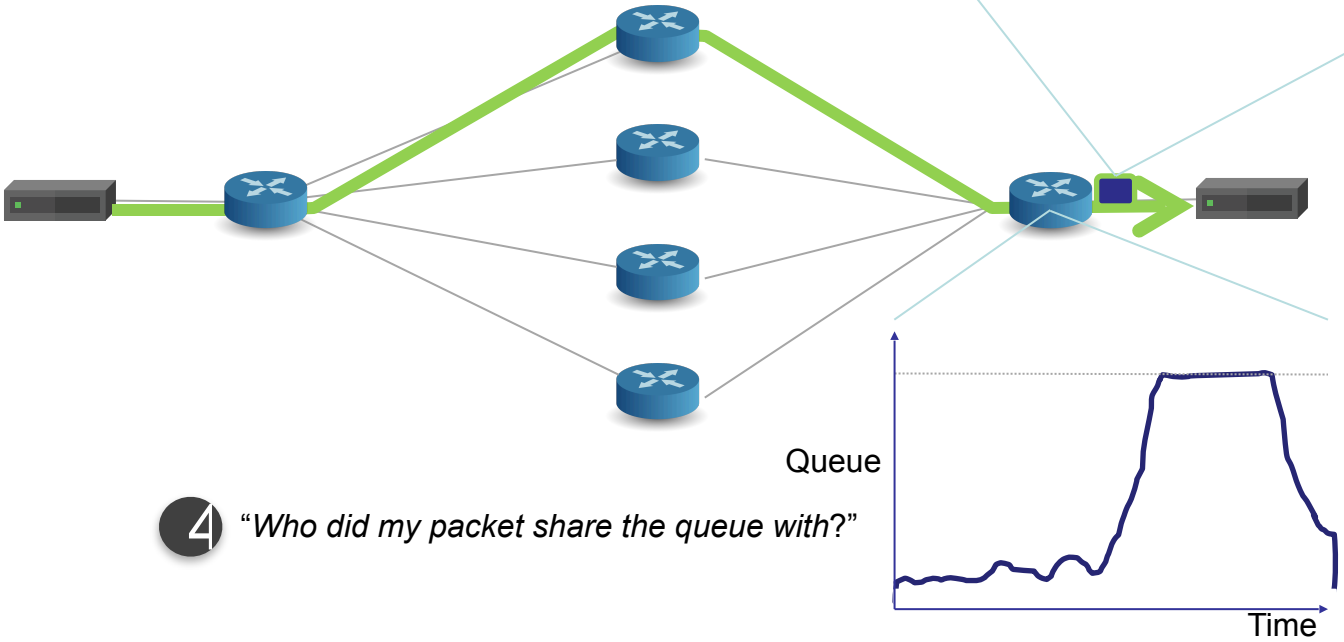
# How programmability is used

3 Network telemetry


1
"Which path did my packet take?"
"I visited Switch 1 @780ns,
Switch 9 @1.3µs, Switch 12 @2.4µs"
# Rule
1
2
3
…
75 192.168.0/24
…
2
"Which rules did my packet follow?"
"In Switch 1, I followed rules 75 and 250.
In Switch 9, I followed rules 3 and 80. "

3 “*How long did my packet queue at each switch?*”

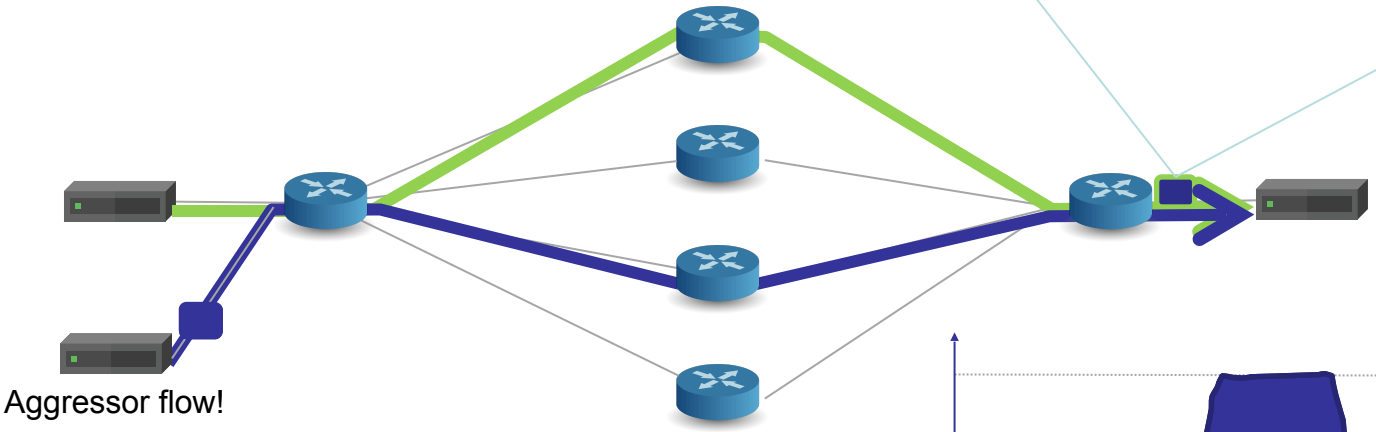
“*Delay: 100ns, 200ns, 19740ns*”

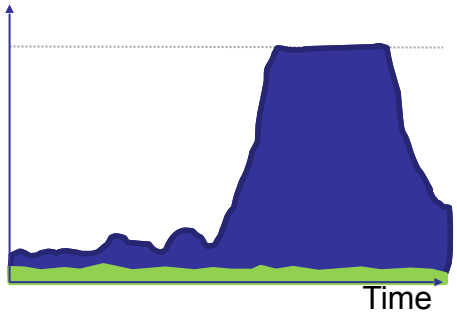



4 “*Who did my packet share the queue with?*”

3 “*How long did my packet queue at each switch?*”

“*Delay: 100ns, 200ns, 19740ns*”



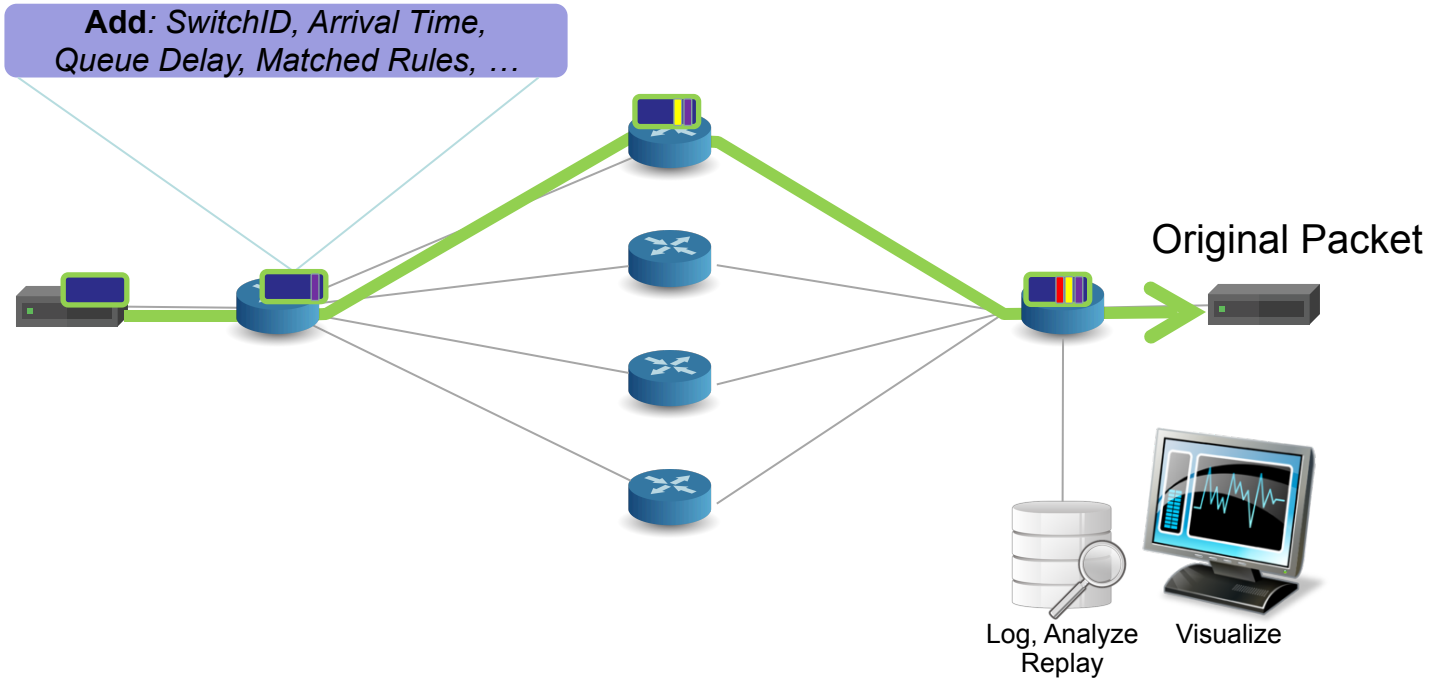

4 “*Who did my packet share the queue with?*”

# These seem like pretty important questions

1. "*Which path did my packet take*?"
2. "*Which rules did my packet follow*?"
3. "*How long did it queue at each switch*?"
4. "*Who did it share the queues with*?"

A programmable device can potentially answer all four questions.
At line rate.

# INT: In-band Network Telemetry

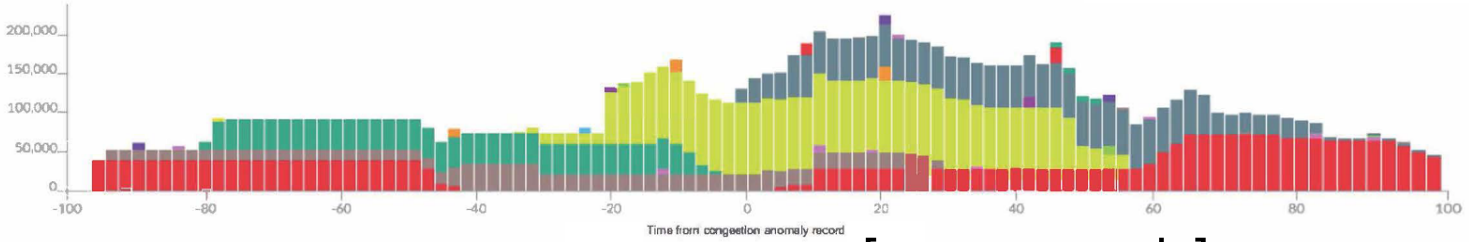


# Example using INT

Anomaly Records

| Timestamp | Switch Id |
|---|---|
| July 25, 2017 - 18:17:51.513 UTC | |

**Queue Occupancy Over Time (bytes)**




[nanoseconds]

**17 Affected Flows**

| Flow | kB in Queue | % of Queue Buildup | Packet Drops |
|---|---|---|---|
| 10.32.2.2:46380 -> 10.36.1.2:5101 TCP | 3282 | 29 | 0 |
| 10.32.2.2:46374 -> 10.36.1.2:5101 TCP | 3073.5 | 27 | 25 |
| 10.32.2.2:46386 -> 10.36.1.2:5101 TCP | 2092.5 | 18 | 27 |
| 10.32.2.2:46388 -> 10.36.1.2:5101 TCP | 1456.5 | 13 | 0 |
| 10.32.2.2:46390 -> 10.36.1.2:5101 TCP | 1227 | 11 | 36 |
| 10.32.2.2:46372 -> 10.36.1.2:5101 TCP | 45 | 0 | 0 |
| 10.32.2.2:46392 -> 10.36.1.2:5101 TCP | 37.5 | 0 | 39 |
| 10.35.1.2:34256 -> 10.36.1.2:5102 TCP | 34.5 | 0 | 0 |

End.