X_405082

Advanced Computer Networks

Programmable Data Plane

Lin Wang (lin.wang@vu.nl)

Period 2, Fall 2020

Part of the content is adapted from Laurent Vanbever

Course outline

Warm-up

- Fundamentals
- Forwarding and routing
- Network transport

Data centers

- Data center networking
- Data center transport

Programmability

- Software defined networking
- **Programmable data plane** 👈

Video

- Video streaming
- Video stream analytics

Networking and ML

- Networking for ML
- ML for networking

Mobile computing

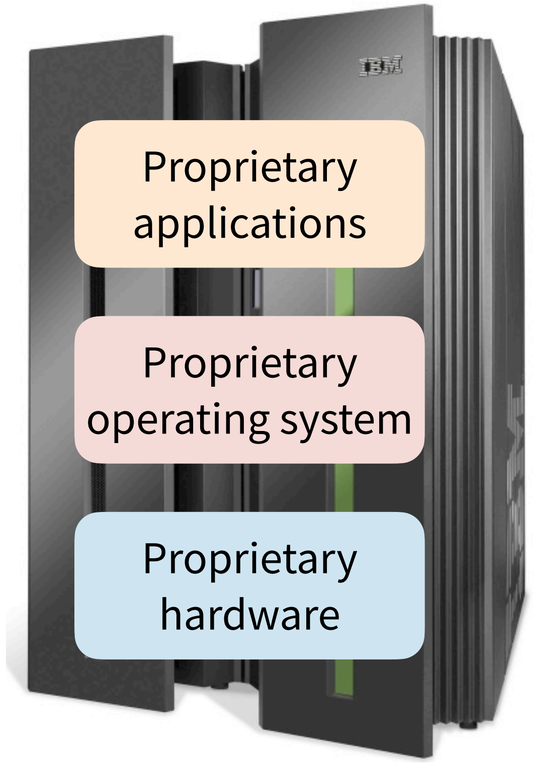
- Wireless and mobile

Learning objectives

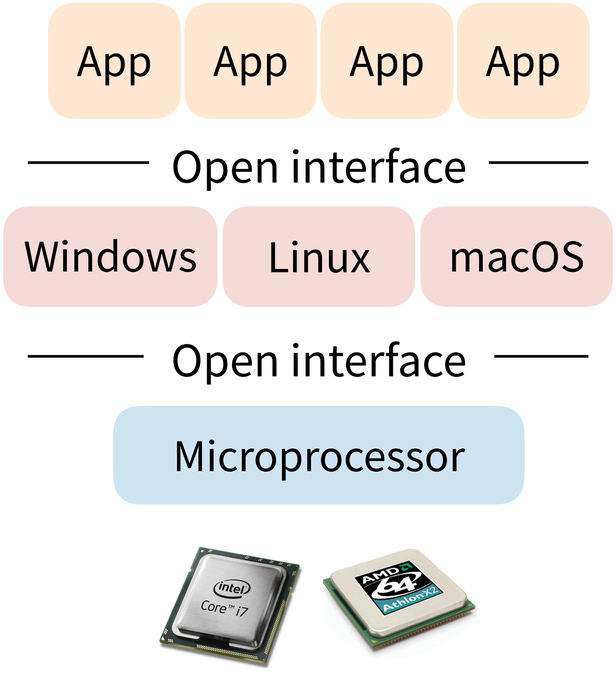
Why do we need data plane programmability? **How** to achieve it?

What **new ideas** can we explore with data plane programmability?

Evolution of the computer industry

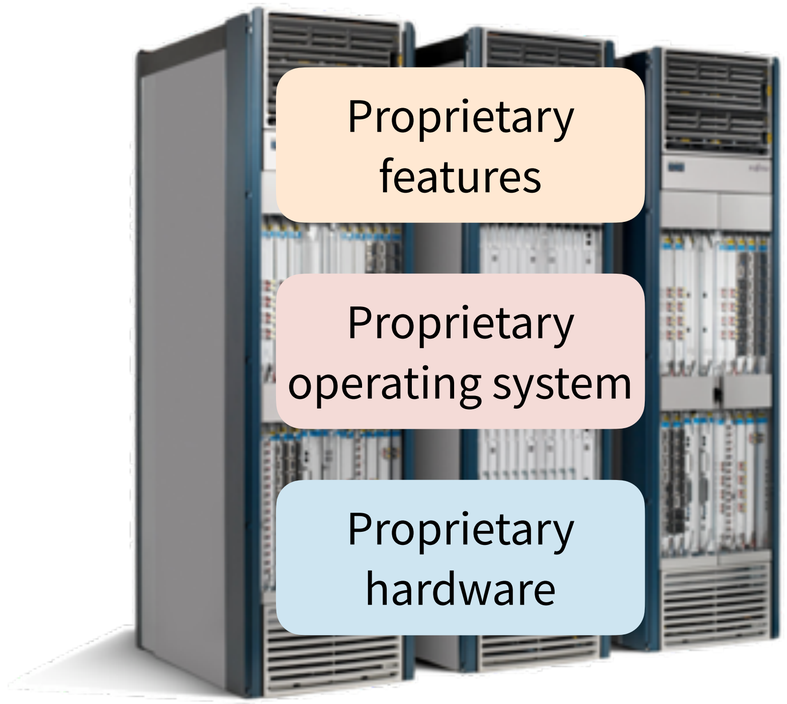


Disaggregation



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

Evolution of networking industry



Disaggregation

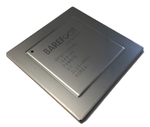
App | App | App | App

Open interface

Floodlight | Ryu | ONOS

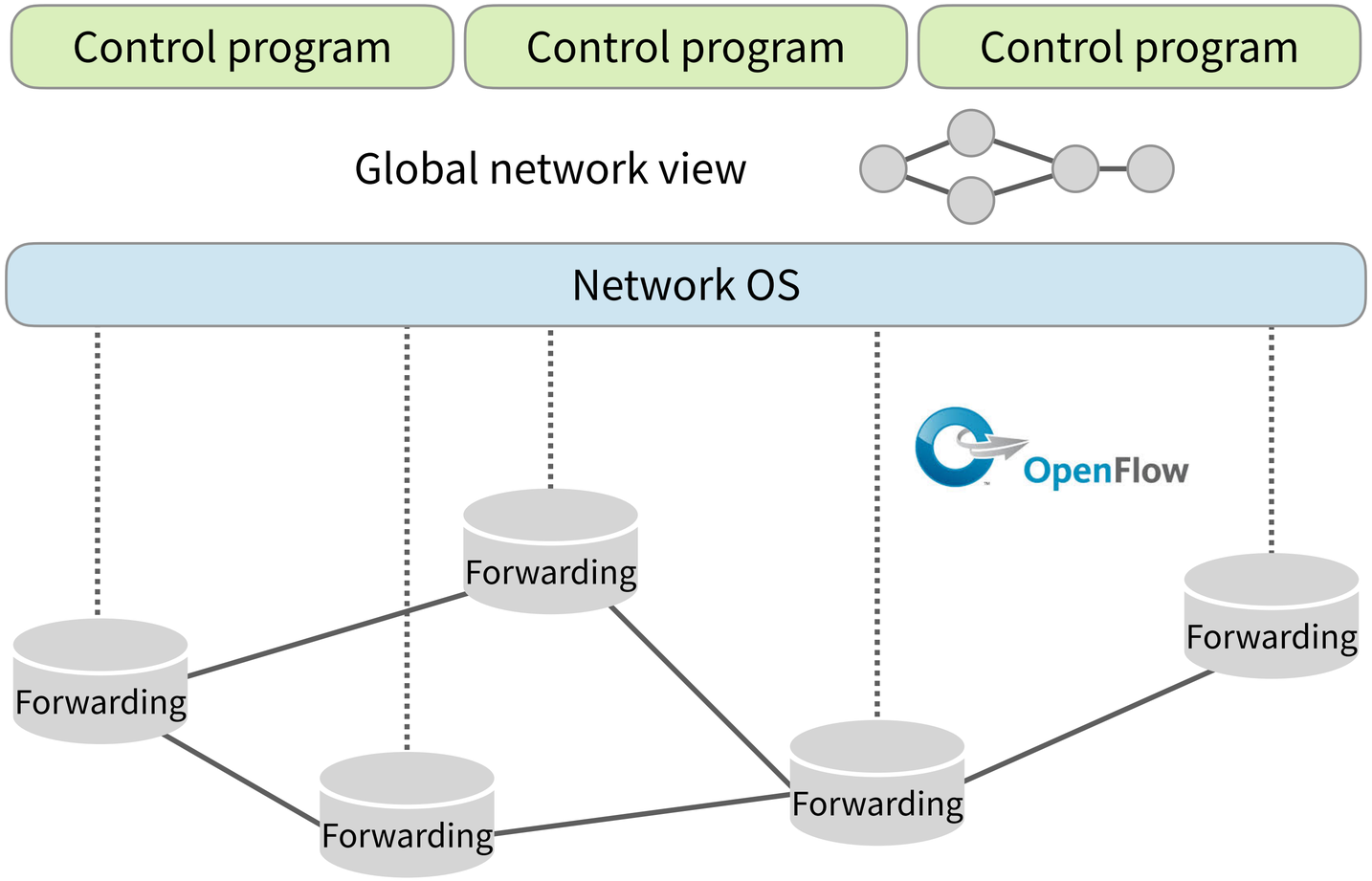
Open interface

Merchant switch chips

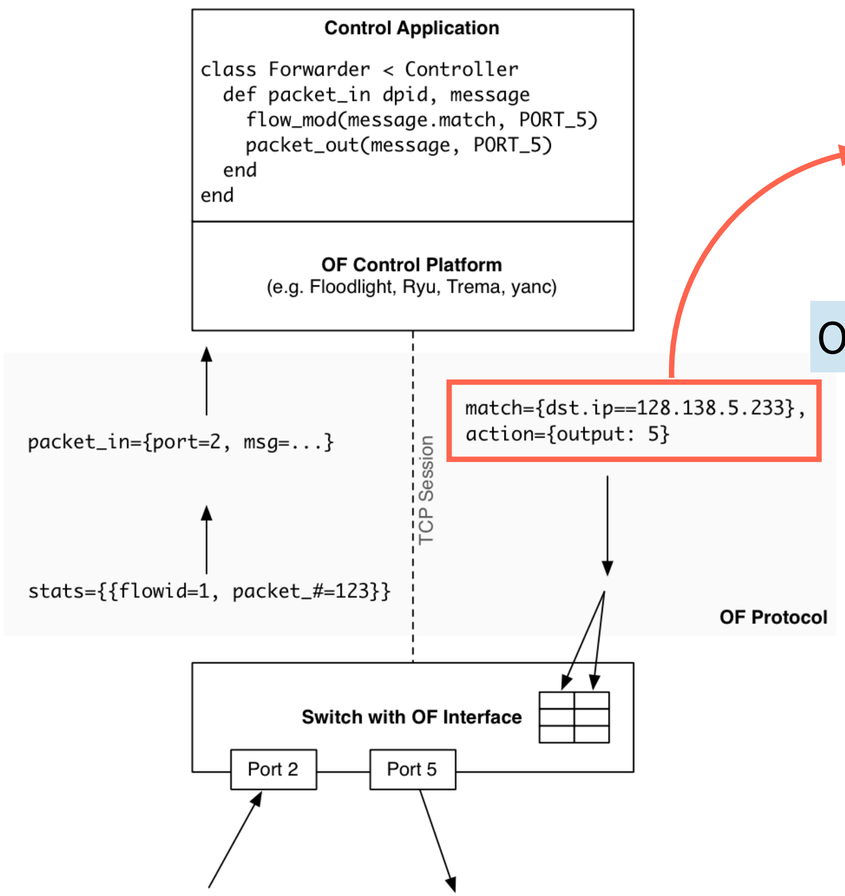


The networking industry has also been evolving from proprietary hardware/software towards more **general-purpose** hardware/software with **open standards/interfaces**.

Recap: software define networking



A deep dive into OpenFlow



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

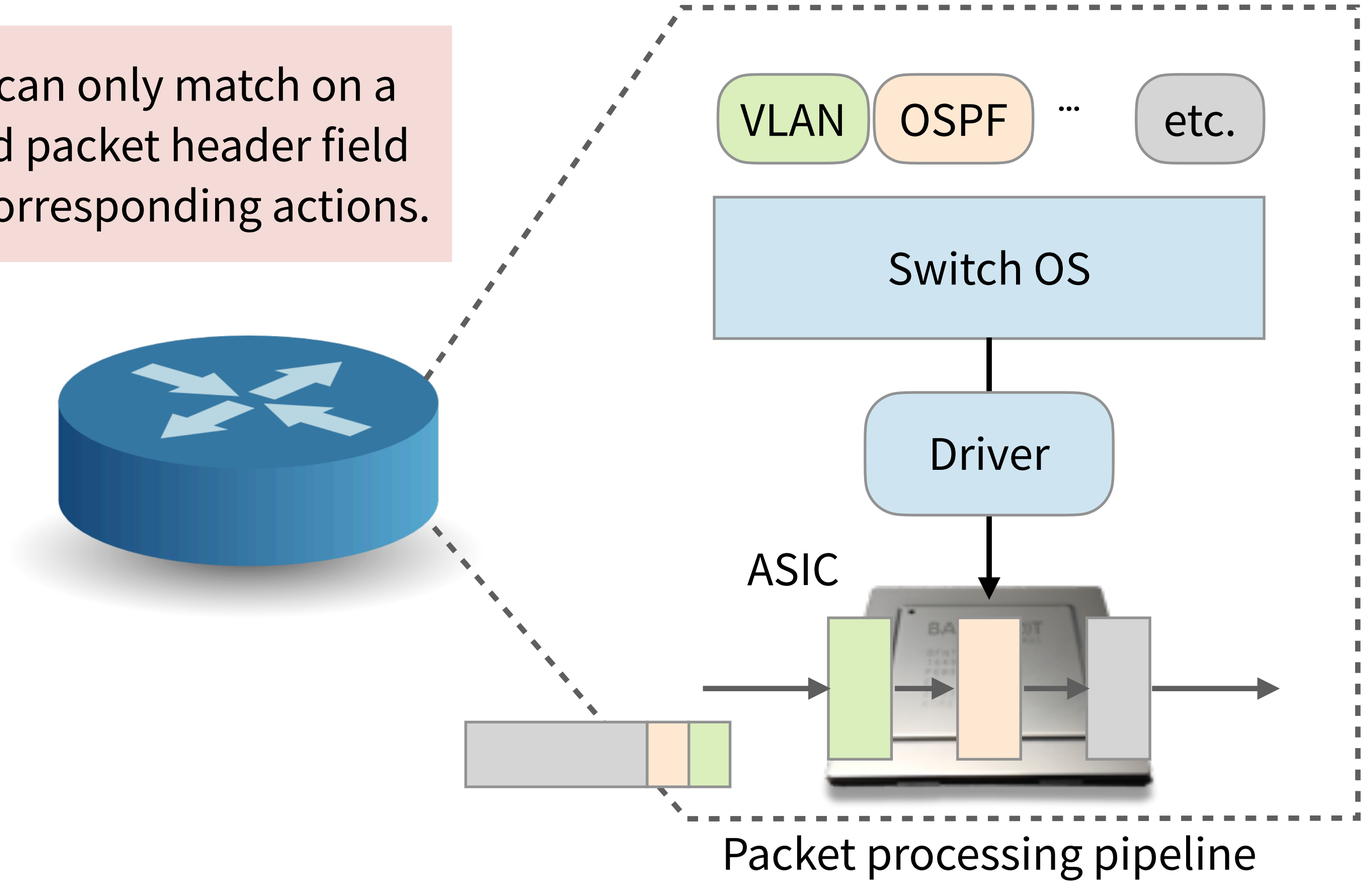
```
enum oxm_ofb_match_fields {
  OFPXMT_OFB_IN_PORT,
  OFPXMT_OFB_IN_PHY_PORT,
  OFPXMT_OFB_METADATA,
  OFPXMT_OFB_ETH_DST ,
  OFPXMT_OFB_ETH_SRC ,
  OFPXMT_OFB_ETH_TYPE ,
  OFPXMT_OFB_VLAN_VID ,
  OFPXMT_OFB_VLAN_PCP ,
  OFPXMT_OFB_IP_DSCP ,
  OFPXMT_OFB_IP_ECN ,
  OFPXMT_OFB_IP_PROTO ,
  OFPXMT_OFB_IPV4_SRC ,
  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
};
```

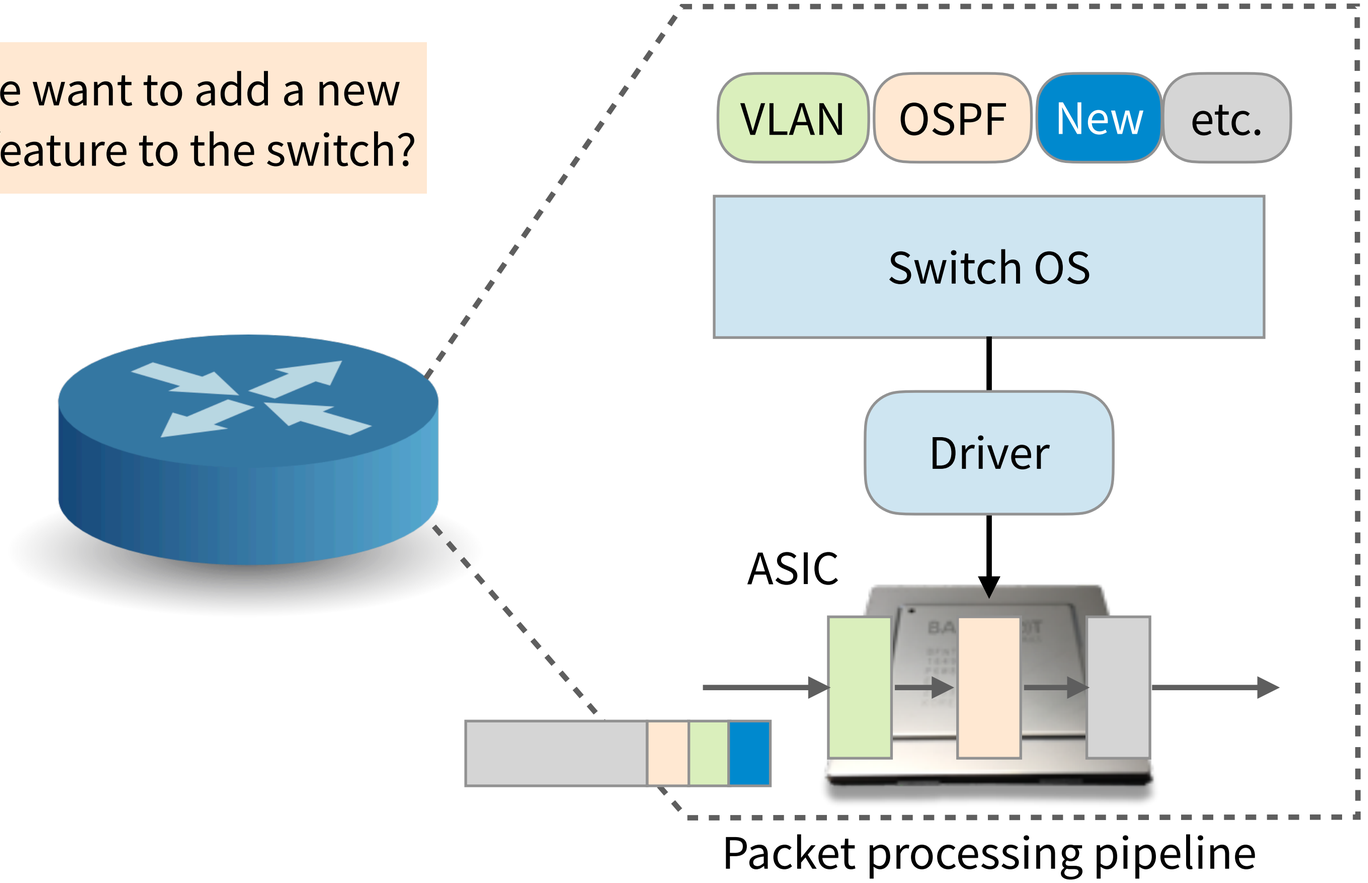
Switch architecture

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



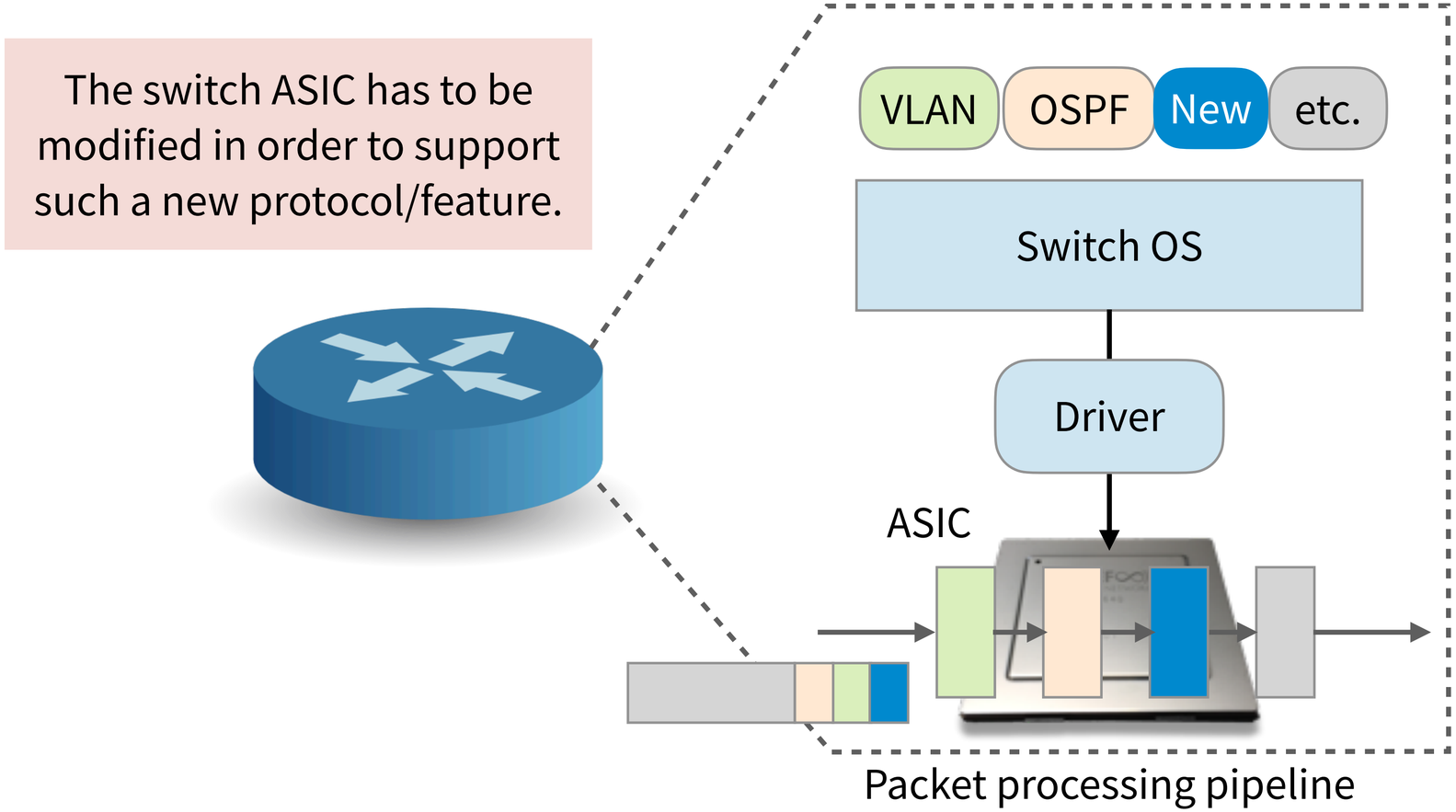
Switch architecture: adding a new protocol

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

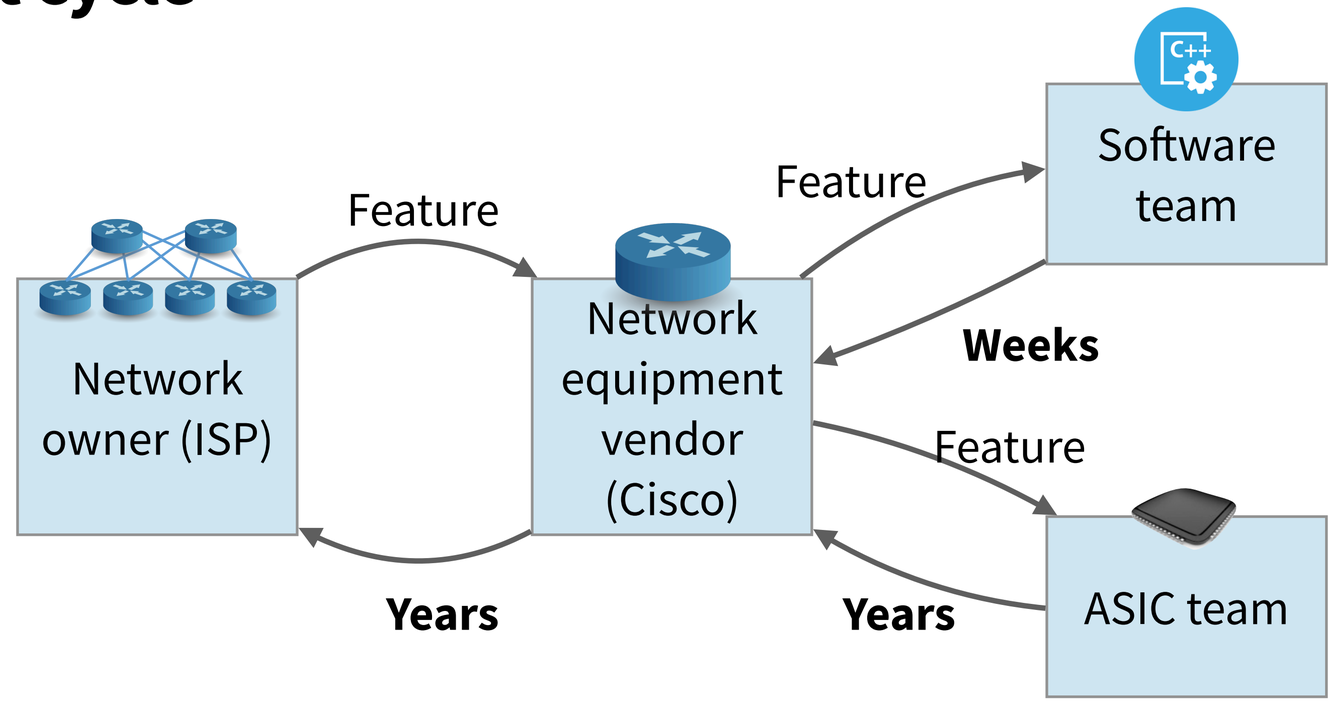


Switch architecture: required modifications

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



Development cycle



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

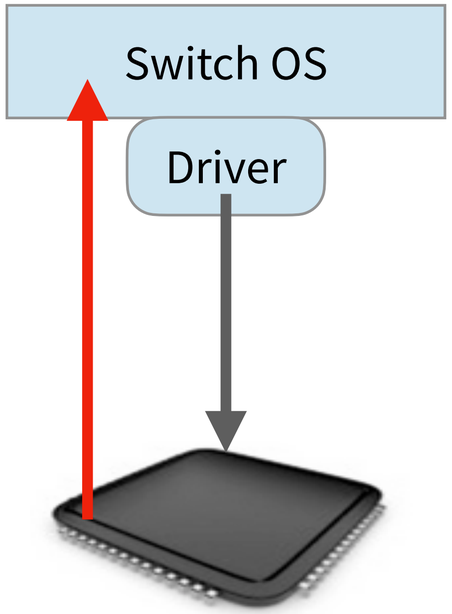
- It either **no longer solves your problem**
- You need **a fork-lift upgrade** at huge expenses

What is the root cause of all this?

The "bottom-up" mentality

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

"This is how I process packet..."



Fixed function switch

Any other ideas? If so, why you think it is better?

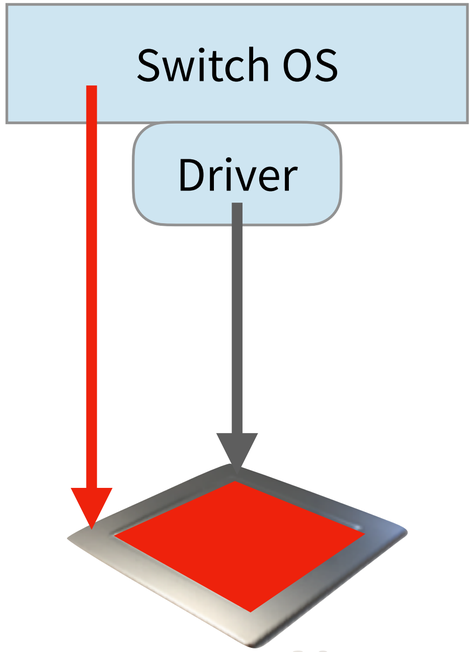
The "top-down" approach

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);
}
```

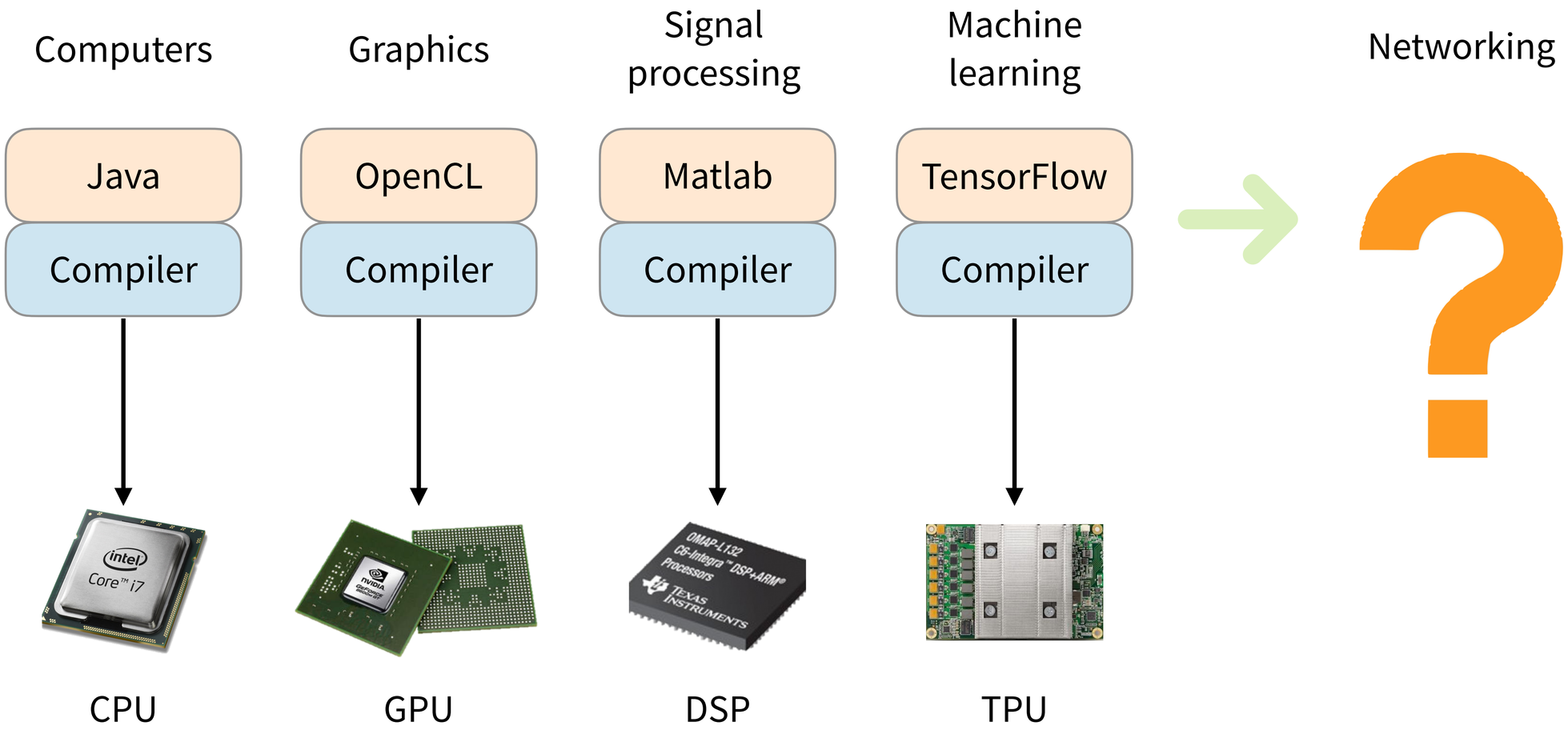
"This is precisely how you must process packets..."



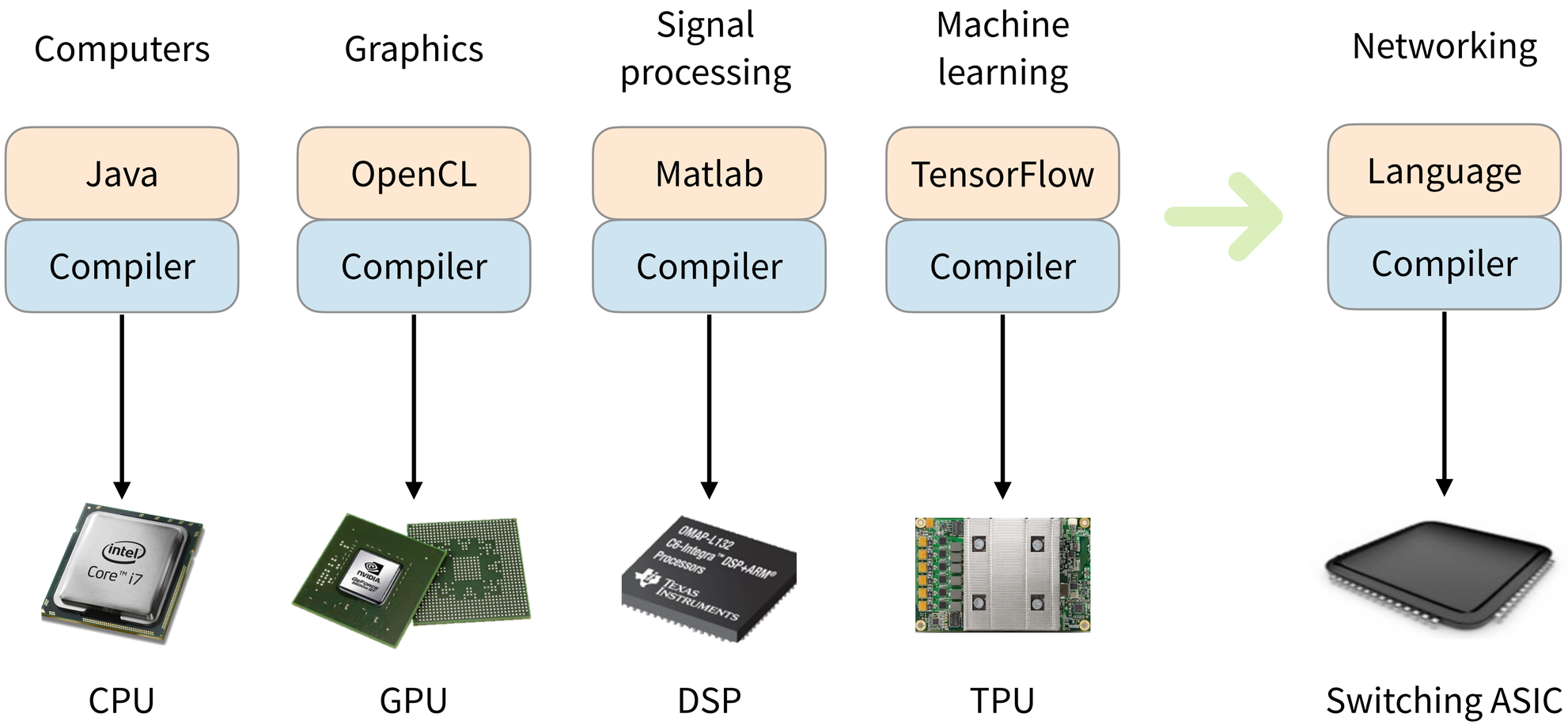
Customizable switching ASIC

How to support programmability?

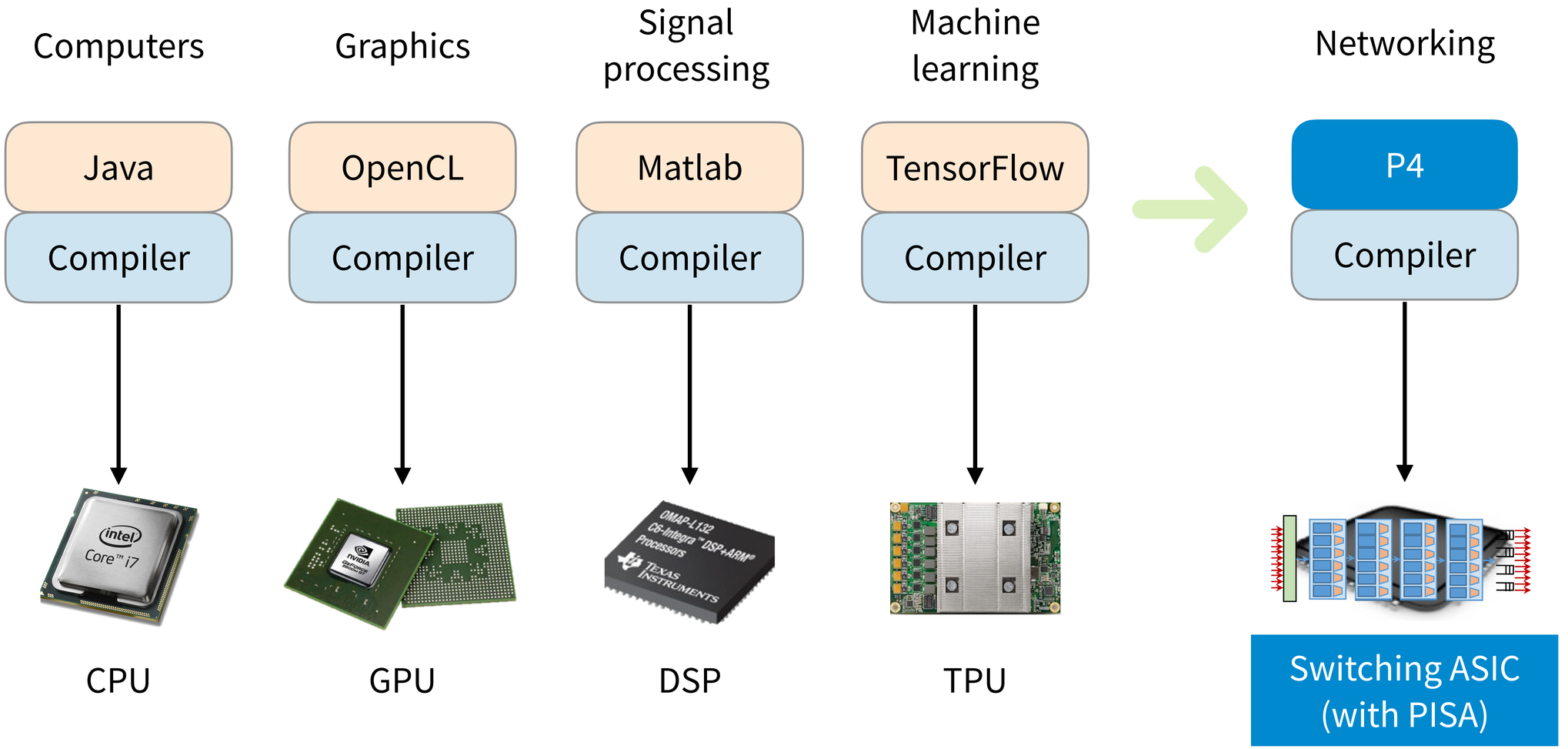
Domain-specific processors



Domain-specific processors



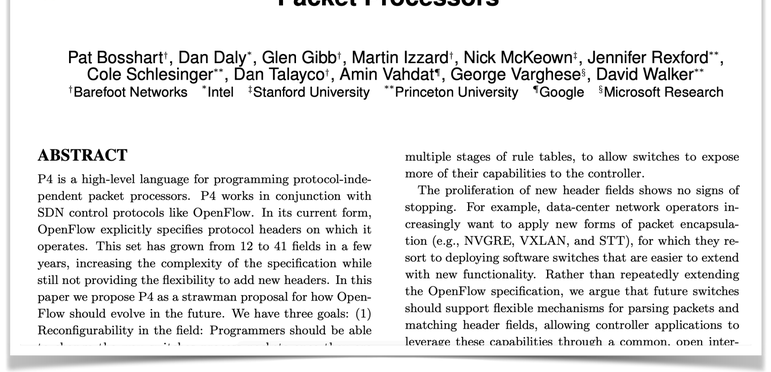
Programmable switch ASIC abstraction: PISA



PISA and P4

PISA: **p**rotocol **i**ndependent **s**witch **a**rchitecture

P4: **p**rogramming **p**rotocol-independent **p**acket **p**rocessors – a high-level 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

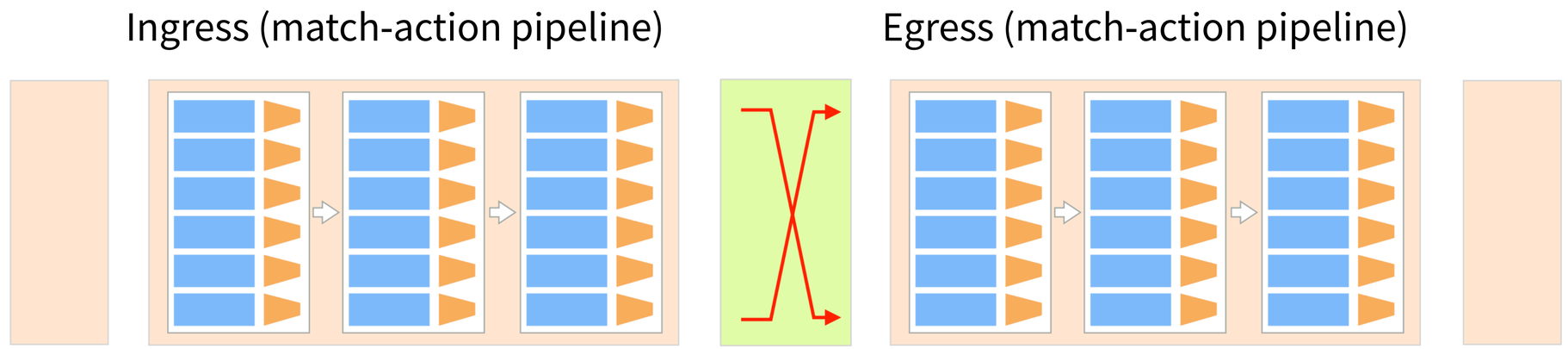
ABSTRACT

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

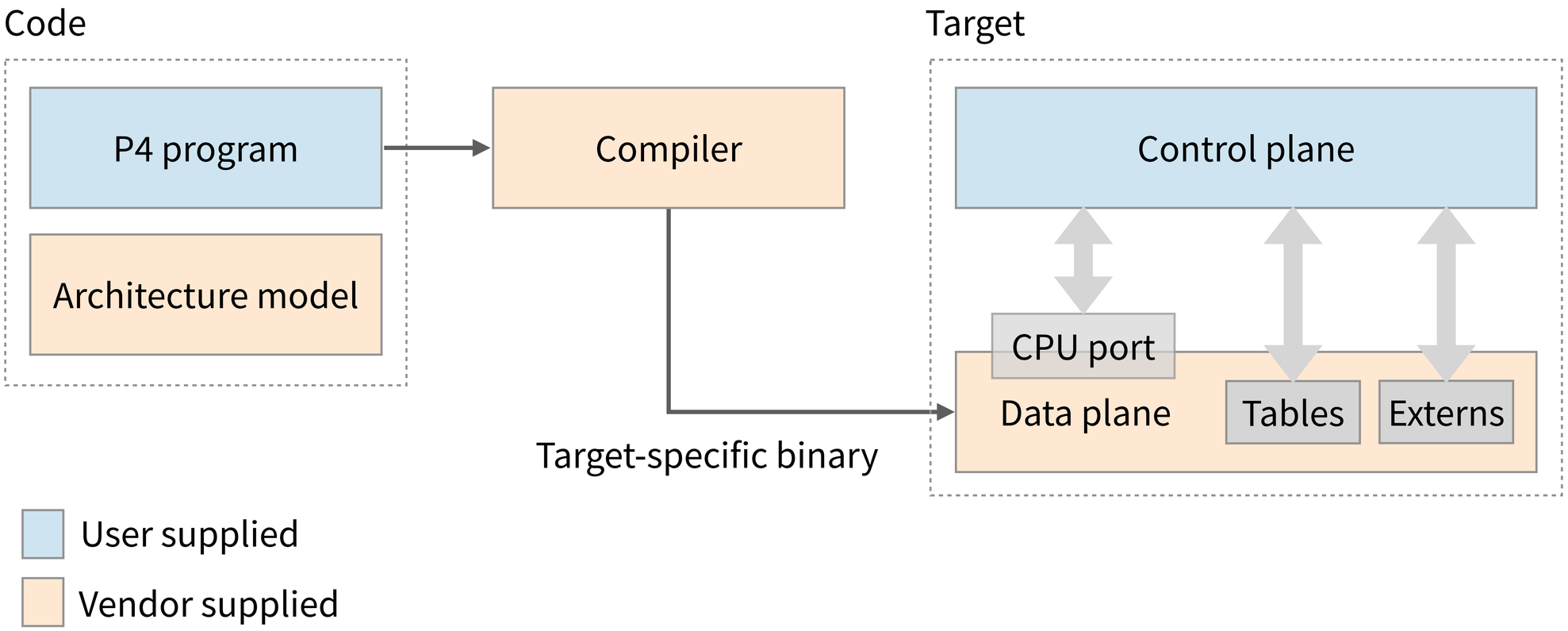
multiple stages of rule tables, to allow switches to expose more of their capabilities to the controller.

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-

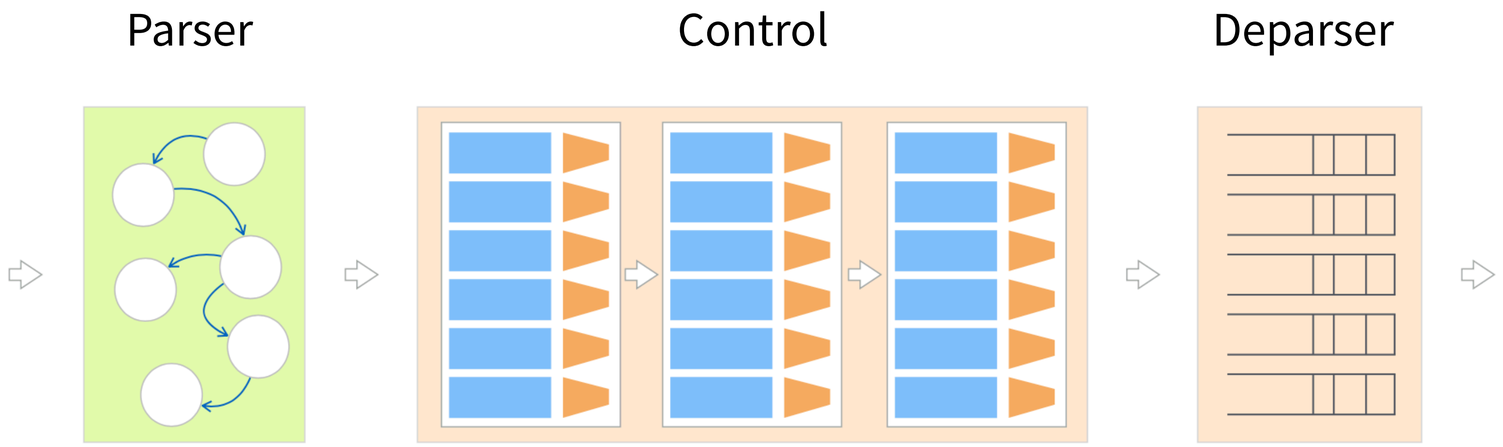
ACM SIGCOMM CCR 2014



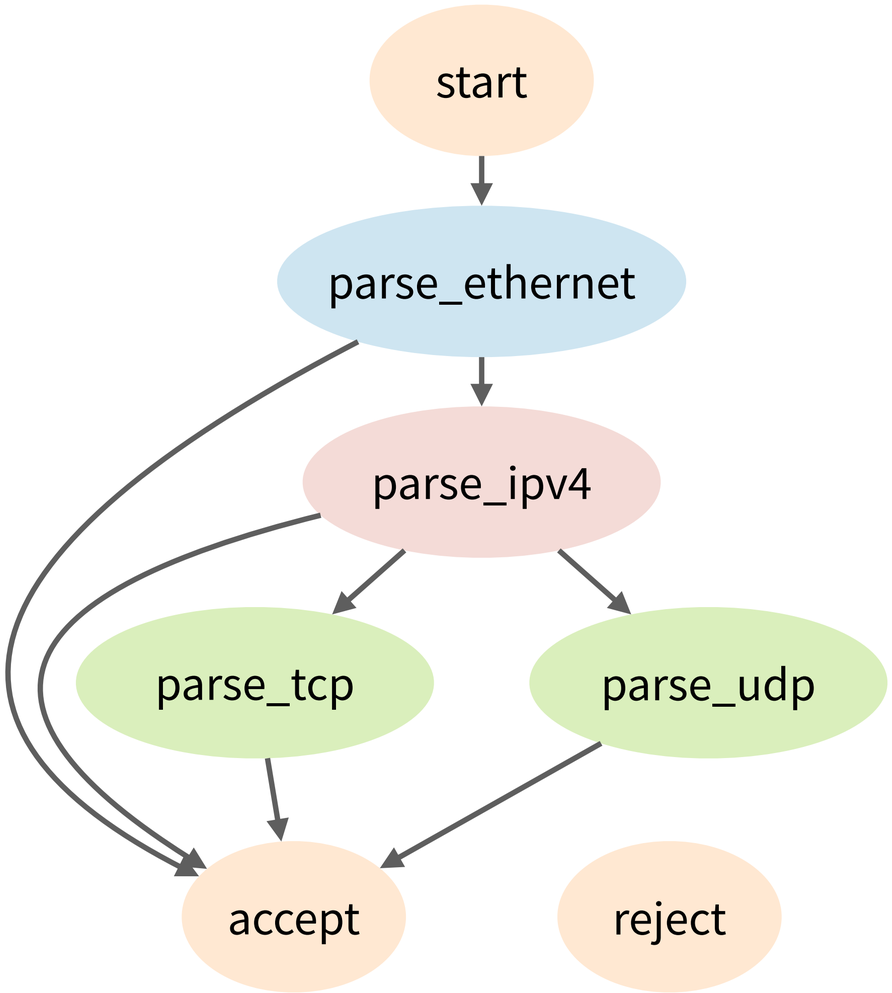
Programming a P4 target



P4 processing overview



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 {
      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

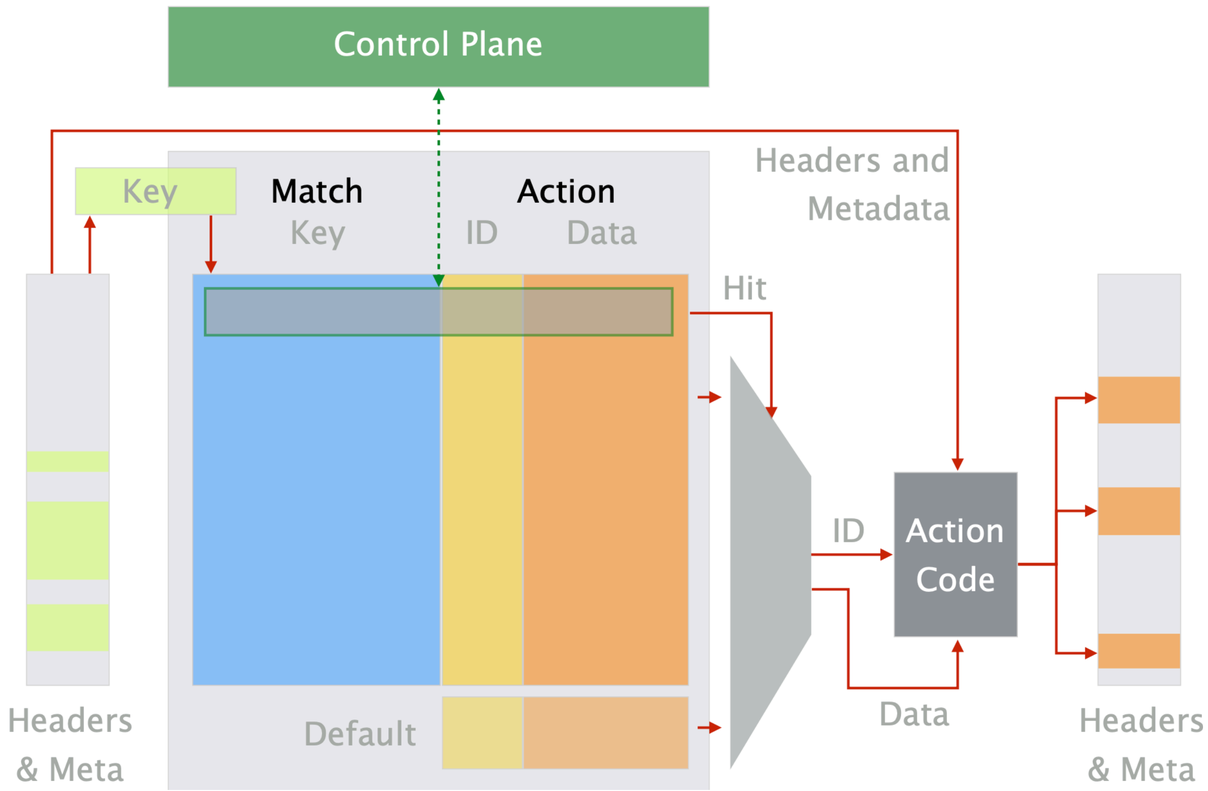
P4 control: table, action

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

   size = 1024;
   default_action = drop();
}

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);
```



P4 deparser

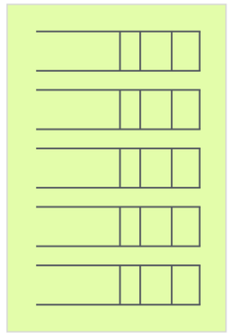
Packet headers

ethernet {srcAddr: a:b:c:d, …}

ipv4 {srcAddr: 1.2.3.4, …}

tcp {srcPort: 1234, …}

Deparser



Packet

a:b:c:d → 1:2:3:4

1.2.3.4 → 5.6.7.8

1234 → 56789

Payload

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

More on p4.org.

Don't forget to watch the video on the page!

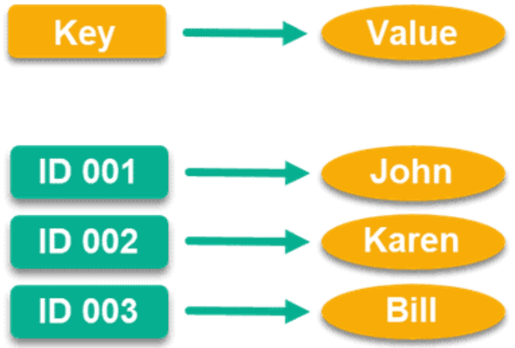
Key-value storage

Store, retrieve, manage key-value objects

- Critical building block for large-scale cloud services
- Need to meet aggressive latency and throughput objectives efficiently

Target workloads

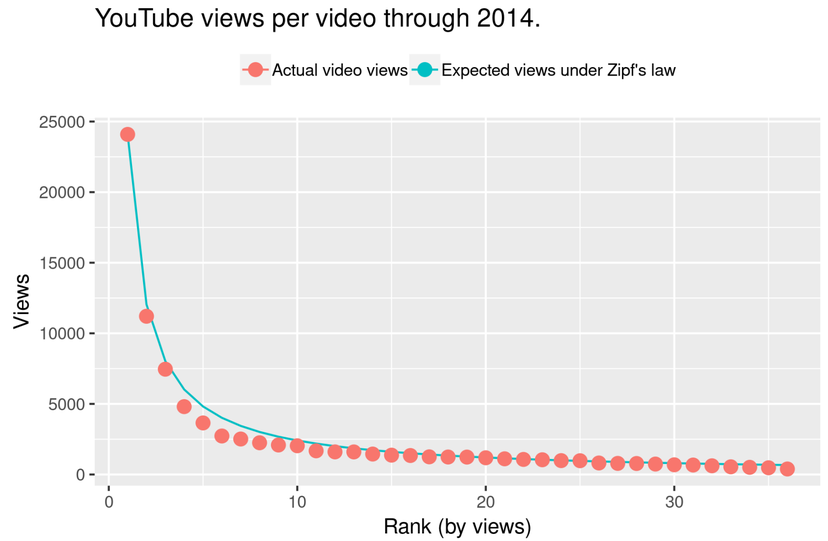
- Small objects
- Read intensive
- **Highly skewed and dynamic key popularity**

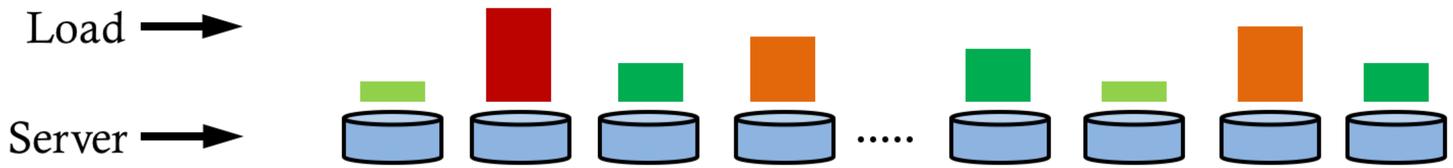


Key-value store

Challenges

- Highly skewed and rapidly changing workloads
- **Low throughput** and **high tail latency**
- How to provide effective dynamic load balancing?





If we randomly distribute the items, some machines will get more visits than others.

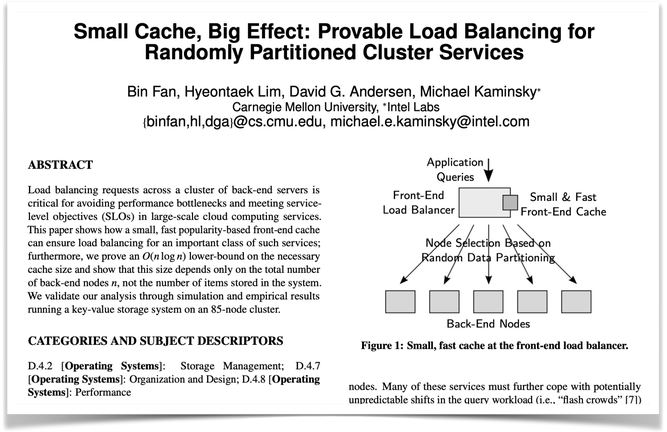
Key-value store

Fast, small cache can ensure load balancing

Strong theoretical results: cache O(NlogN) hottest items can already ensure load balancing

- N: number of servers
- E.g., 100 backend servers with 100 billion items

Requirement: cache throughput ≥ backend aggregate throughput



Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services

Bin Fan, Hyeontaek Lim, David G. Andersen, Michael Kaminsky*
Carnegie Mellon University, *Intel Labs
{binfan,hl,dga}@cs.cmu.edu, michael.e.kaminsky@intel.com

ABSTRACT

Load balancing requests across a cluster of back-end servers is critical for avoiding performance bottlenecks and meeting service-level objectives (SLOs) in large-scale cloud computing services. This paper shows how a small, fast popularity-based front-end cache can ensure load balancing for an important class of such services; furthermore, we prove an $O(n \log n)$ lower-bound on the necessary cache size and show that this size depends only on the total number of back-end nodes n, not the number of items stored in the system. We validate our analysis through simulation and empirical results running a key-value storage system on an 85-node cluster.

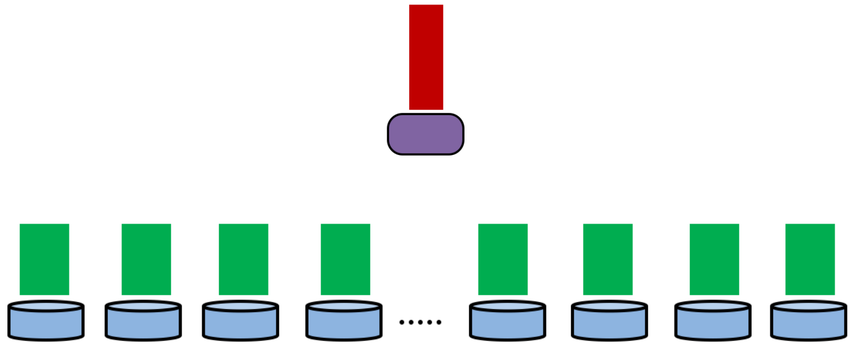
CATEGORIES AND SUBJECT DESCRIPTORS

D.4.2 [**Operating Systems**]: Storage Management; D.4.7 [**Operating Systems**]: Organization and Design; D.4.8 [**Operating Systems**]: Performance

Figure 1: Small, fast cache at the front-end load balancer.

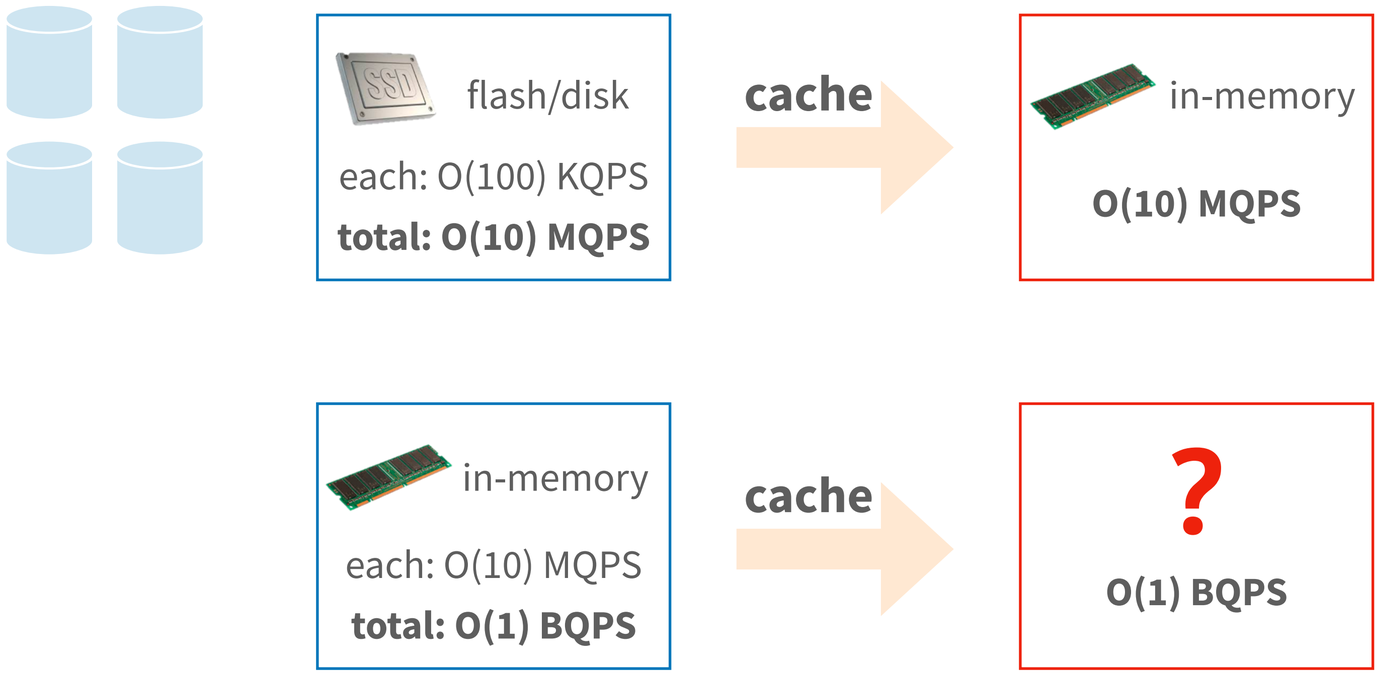
nodes. Many of these services must further cope with potentially unpredictable shifts in the query workload (i.e., "flash crowds" [7])

ACM SOCC 2011



How to build the cache?

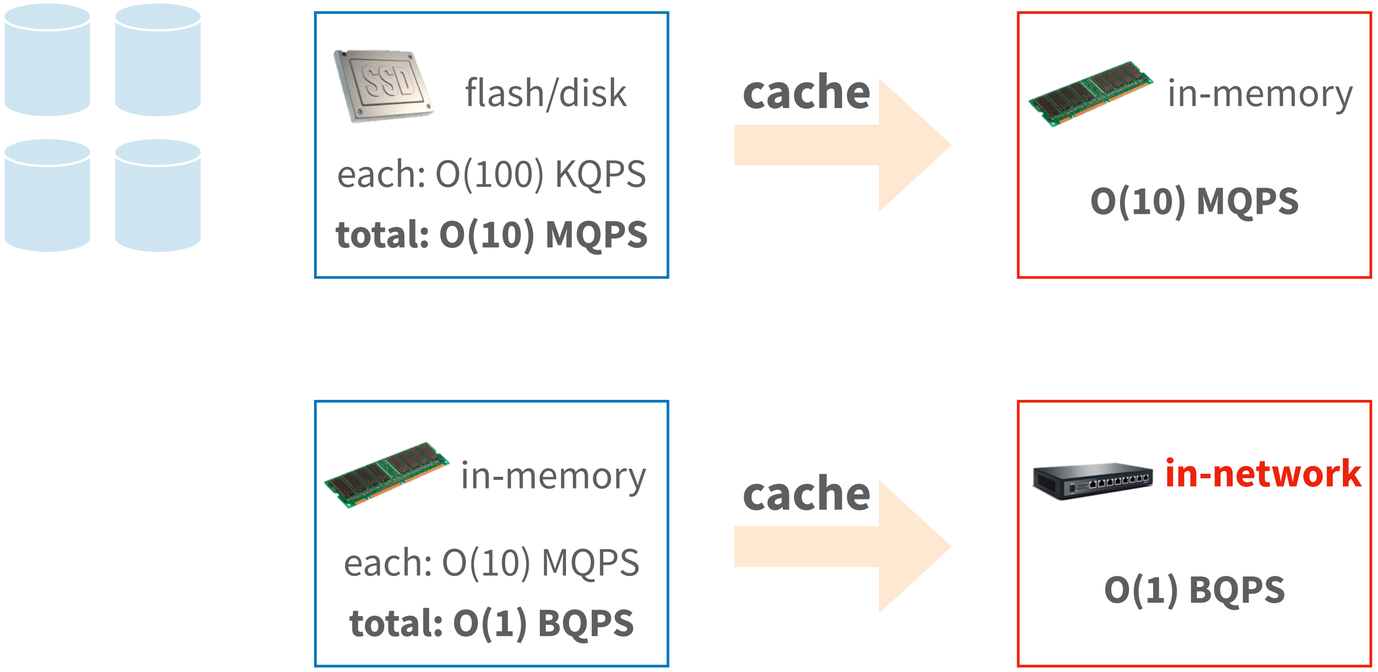
Cache needs to provide the **aggregate** throughput of the storage layer



Nowadays key-value stores are already in memory. How can we achieve even higher throughput?

How to build the cache?

Cache needs to provide the **aggregate** throughput of the storage layer



Limited on-chip memory?
But we only need to cache O(NlogN) small items!

In-network caching

Key-value caching in network ASIC at line rate?! How?

We need to answer the following questions:

How to identify application-level packet fields?

How to store and serve variable length data?

How to efficiently keep the cache up-to-date?

Recall PISA

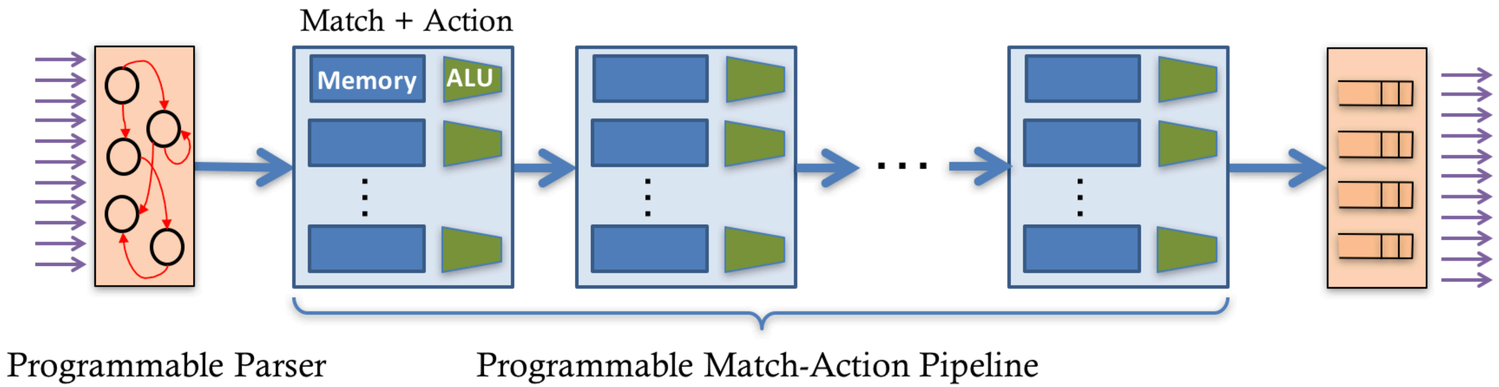
PISA: Protocol independent switch architecture

Programmable parser

- Converts packet data into metadata

Programmable match-action pipeline

- Operate on metadata and update memory states



Key-value store with PISA

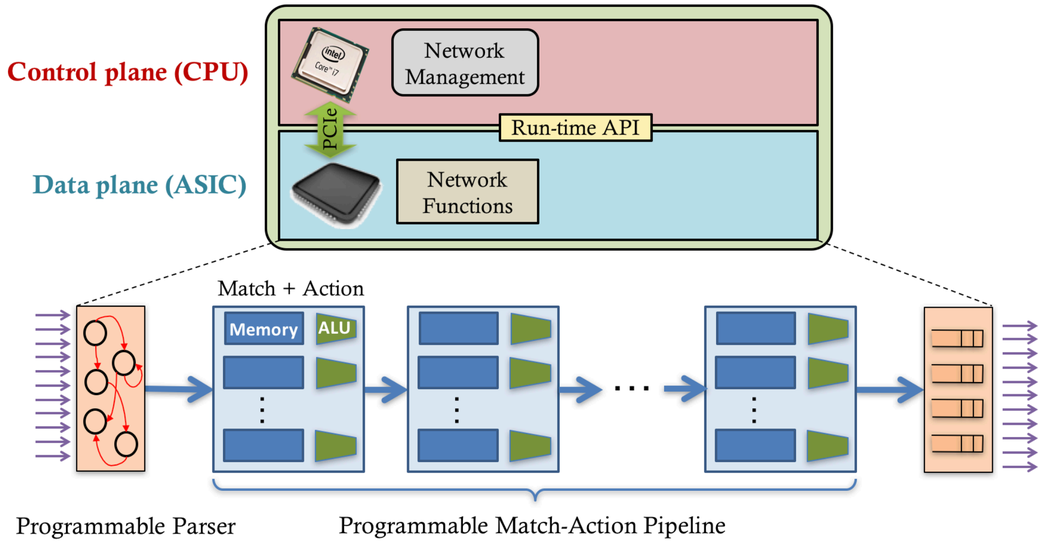
PISA: Protocol independent switch architecture

Programmable parser

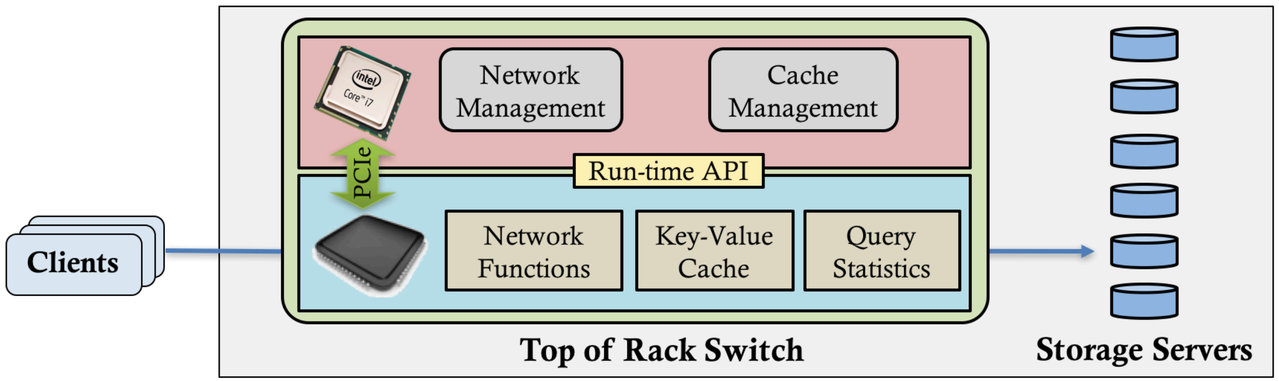
- Parse custom key-value fields in the packet

Programmable match-action pipeline

- Read and update key-value data
- Provide query statistics for cache updates



NetCache rack-scale architecture



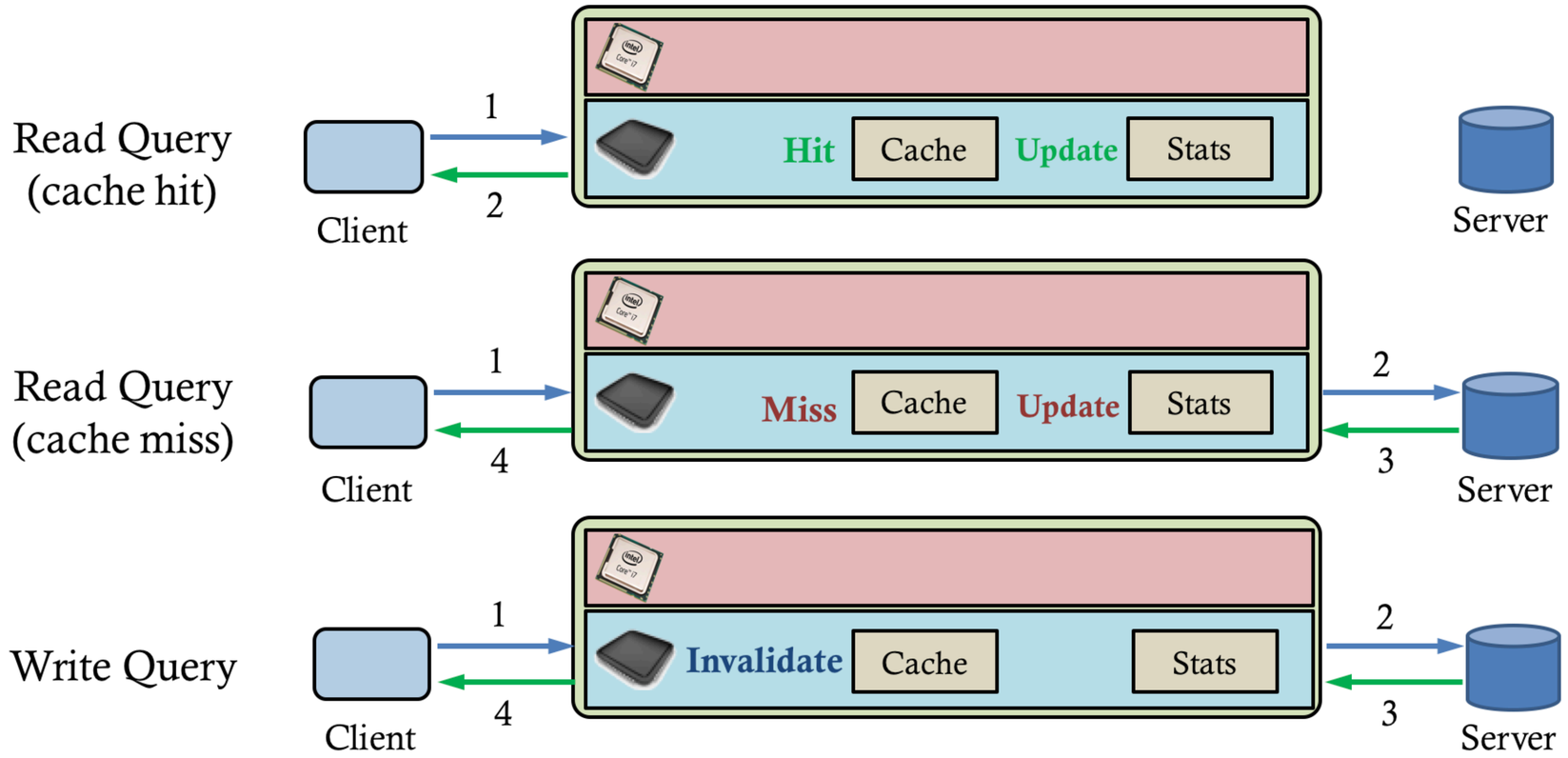
Switch data plane

- **Key-value** store to serve queries for cached keys
- **Query statistics** to enable efficient cache updates (e.g., identifying hot items)

Switch control plane

- Insert hot items into the cache and evict less popular items
- Manage memory allocation for on-chip key-value store

Data plane query handling



In-network caching

Key-value caching in network ASIC at line rate?! How?

We need to answer the following questions:

How to identify application-level packet fields?

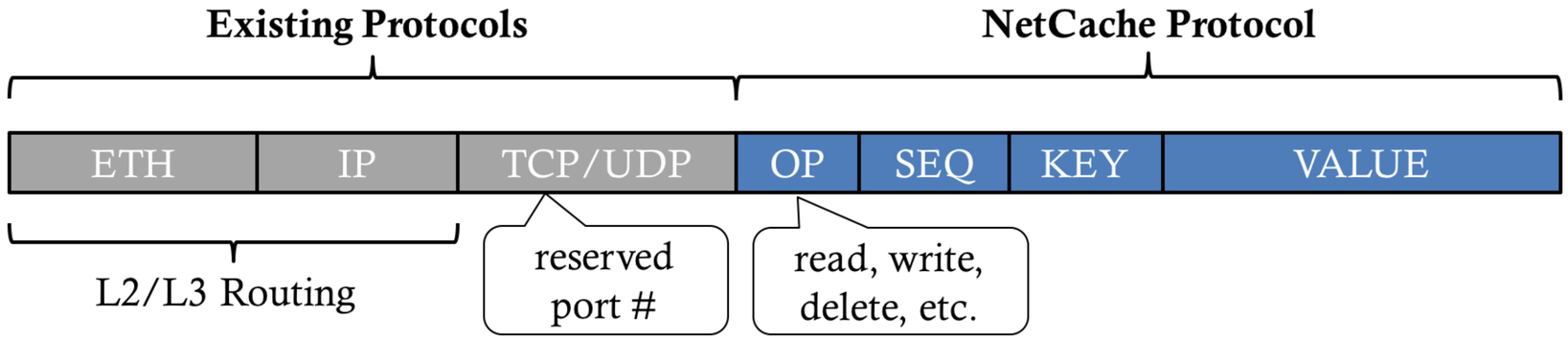
How to store and serve variable length data?

How to efficiently keep the cache up-to-date?

NetCache packet format

Application-layer protocol: compatible with existing L2-L4 layers

Only the top-of-rack switch needs to parse NetCache fields



P4 parser is used to parse the existing protocols headers plus the NetCache protocol fields

In-network caching

Key-value caching in network ASIC at line rate?! How?

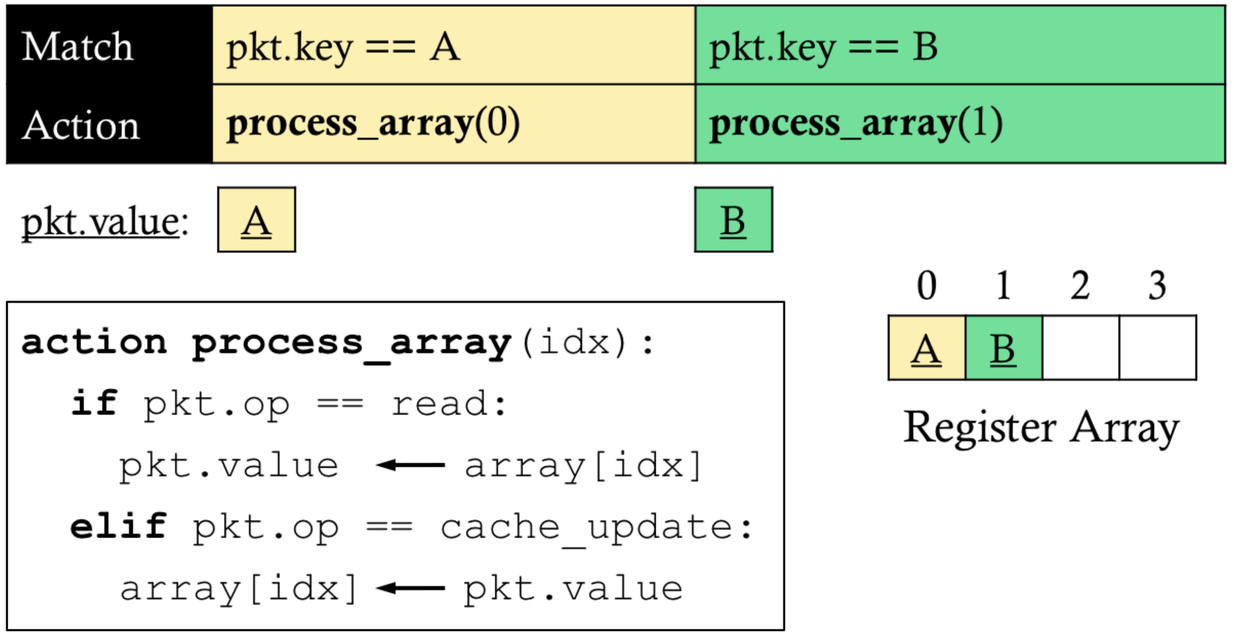
We need to answer the following questions:

How to identify application-level packet fields?

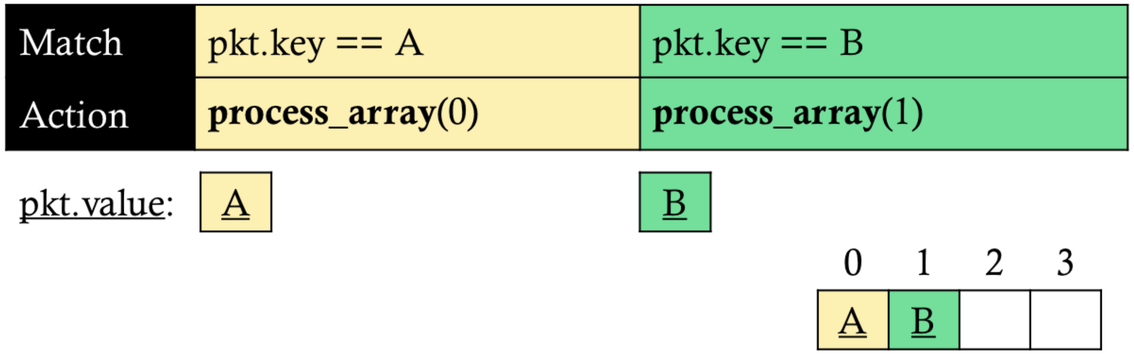
How to store and serve variable length data?

How to efficiently keep the cache up-to-date?

Use register array



Challenges in dealing with variable length



No loop or string due to strict timing requirements

Need to optimize hardware resources consumption

- Number of table entries
- Size of action data from each entry
- Size of intermediate metadata across tables

Potential solutions

Solution 1: use action data to hold the indices

RA

Match	pkt.key == A
Action	process_array([2,3,...])

Problem: number of lookups in one register array (RA) is limited

Solution 2: use multiple RAs with the same match-action

RA1

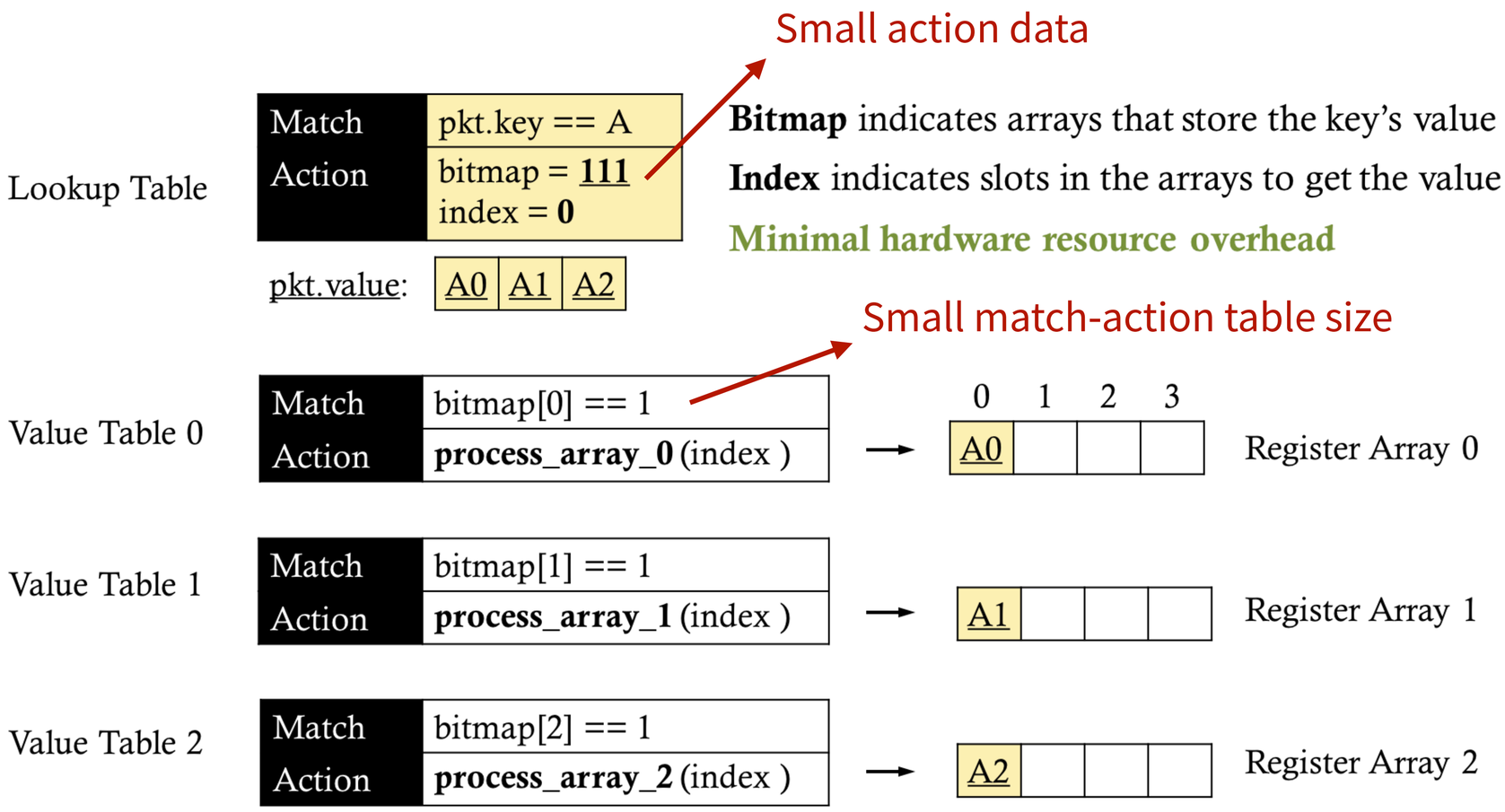
Match	pkt.key == A
Action	process_array(2)

RA2

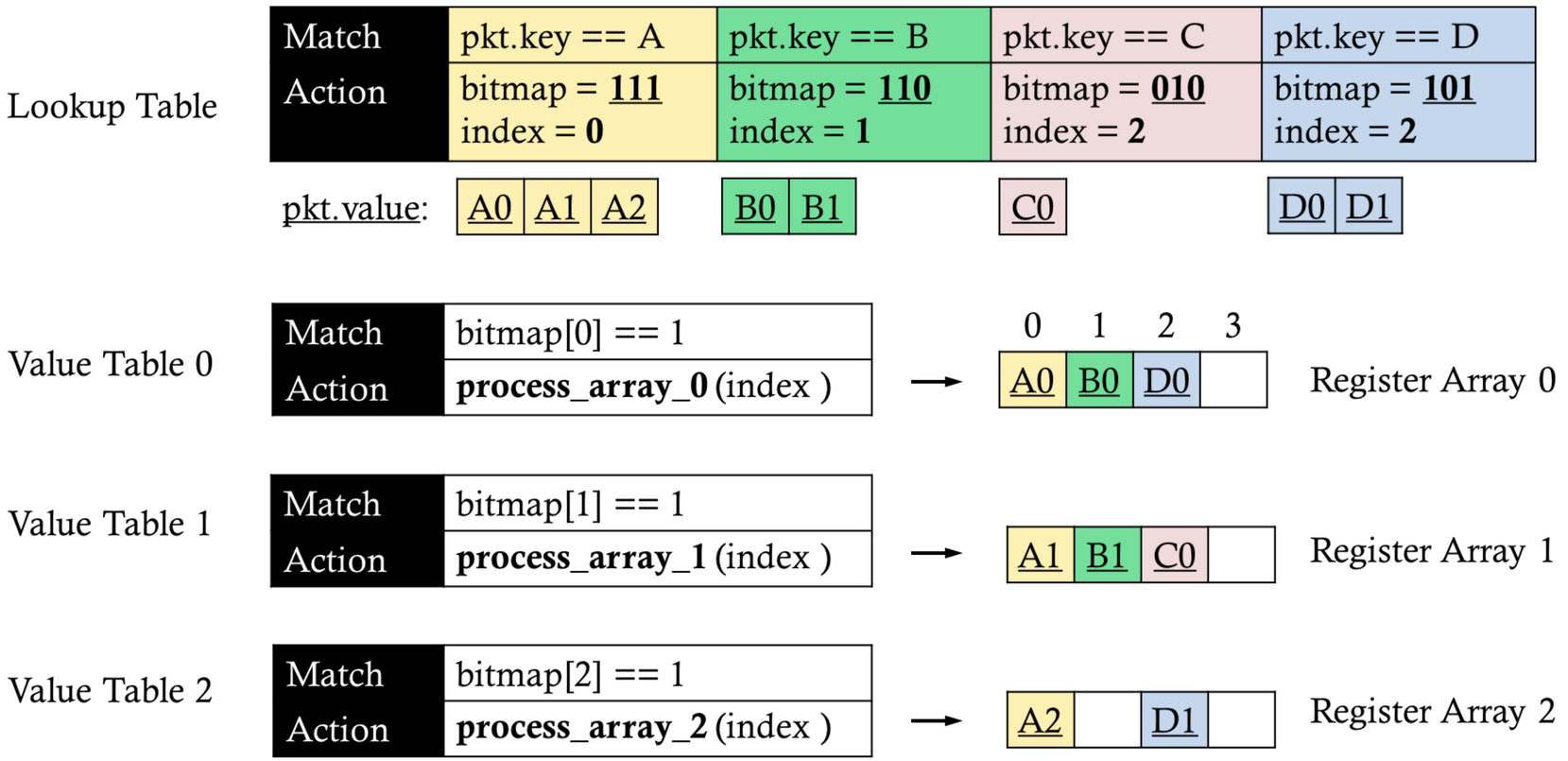
Match	pkt.key == A
Action	process_array(2)

Problem: too many match action table entries

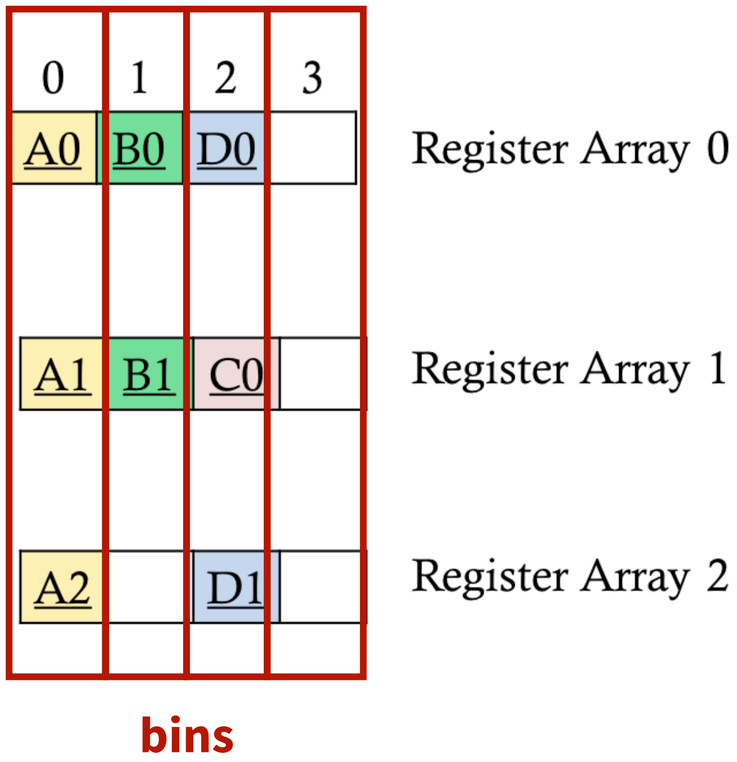
NetCache: two-level lookup



NetCache: two-level lookup



Memory management



Solving a bin-packing problem: use First-Fit heuristics

In-network caching

Key-value caching in network ASIC at line rate?! How?

We need to answer the following questions:

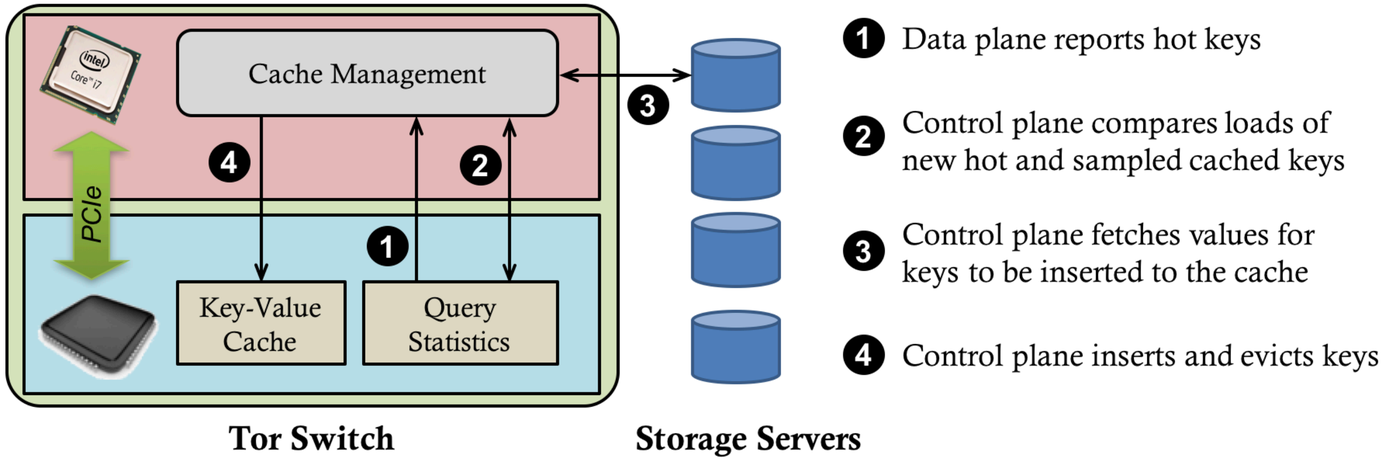
How to identify application-level packet fields?

How to store and serve variable length data?

How to efficiently keep the cache up-to-date?

Cache insertion and eviction

Goal: react quickly and effectively to workload changes with minimal updates



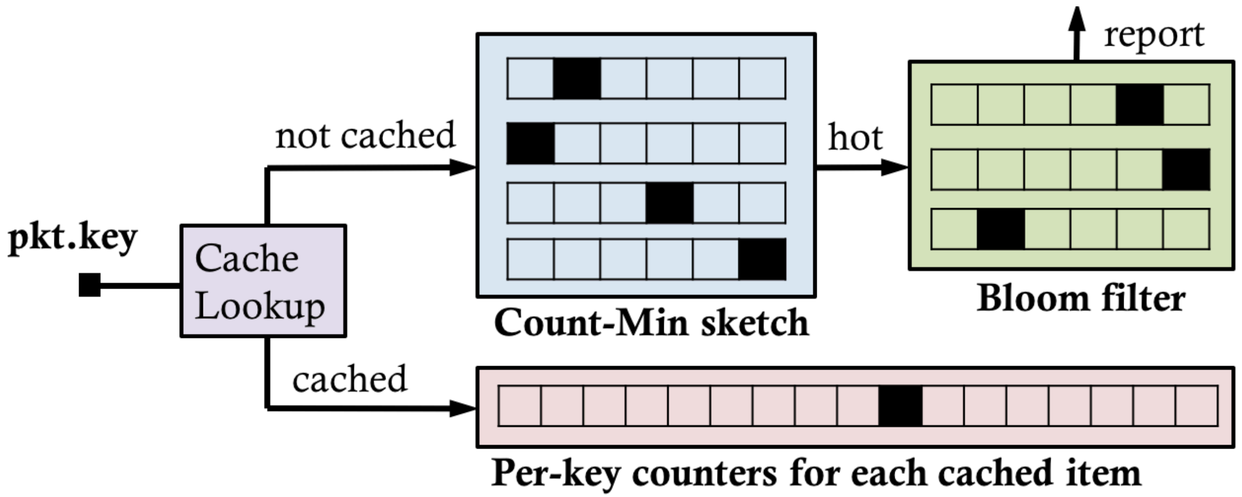
Challenge: cache the hottest O(NlogN) items with limited insertion rate

Query statistics

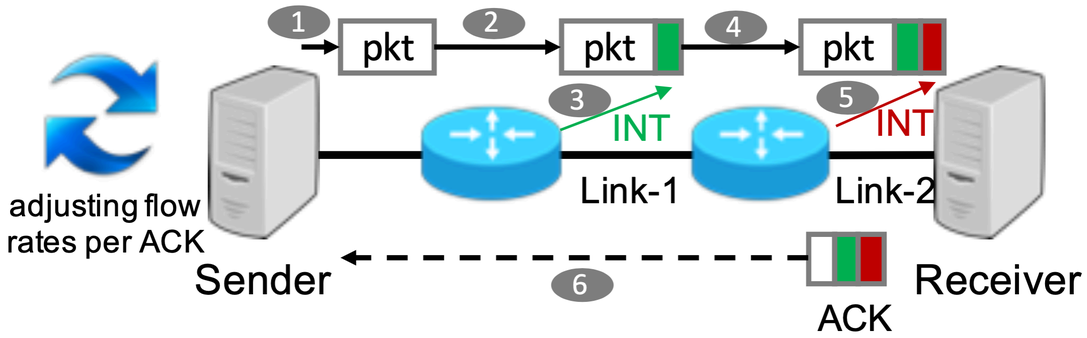
For cached key: per-key counter array

For uncached key:

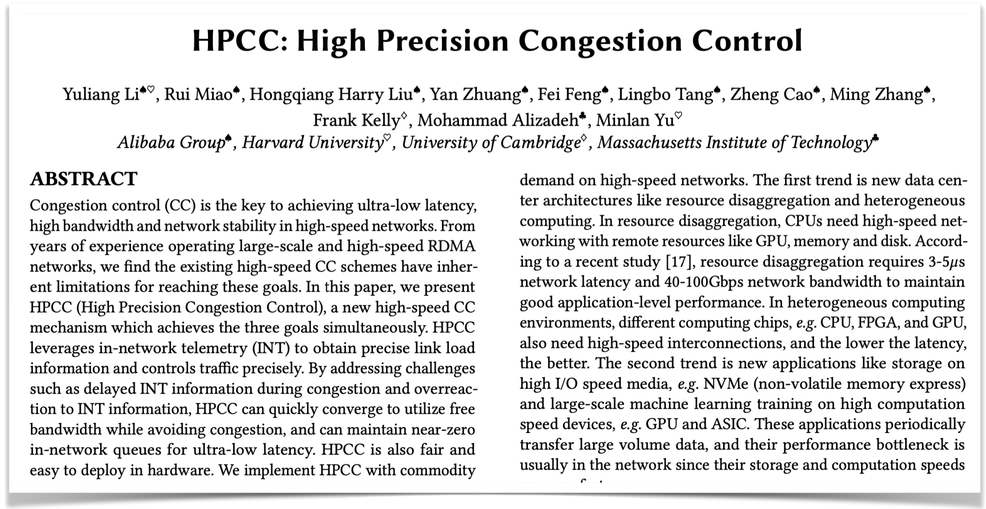
- Count-min sketch: report new hot keys
- Bloom filter: remove duplicated hot key reports



Other applications: congestion control



Use INT to **obtain precise network link load 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

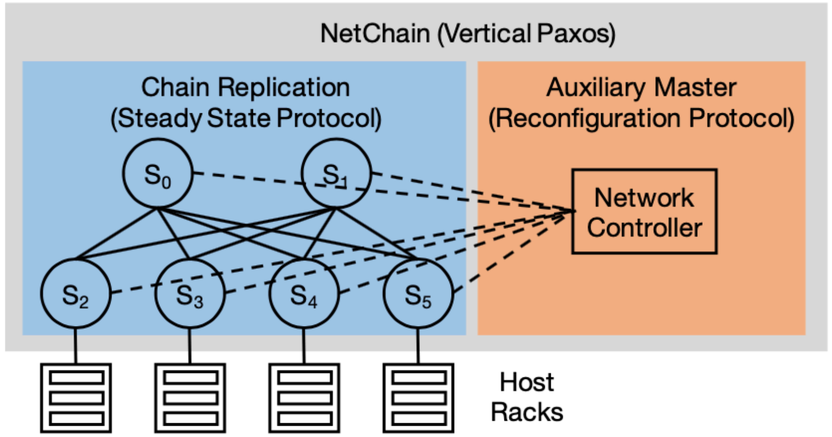
Congestion control (CC) is the key to achieving ultra-low latency, high bandwidth and network stability in high-speed networks. From years of experience operating large-scale and high-speed RDMA networks, we find the existing high-speed CC schemes have inherent limitations for reaching these goals. In 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

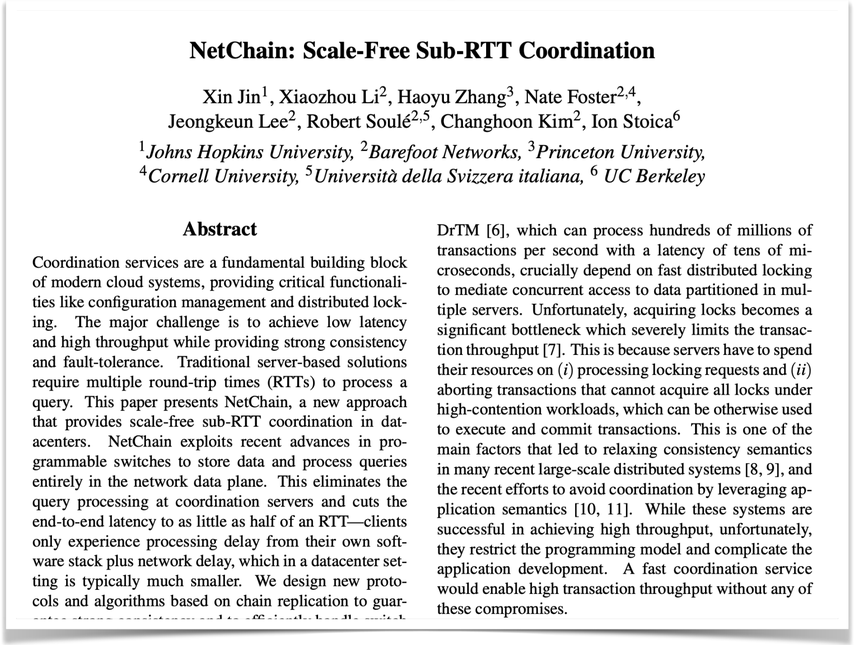
ACM SIGCOMM 2019

Think about the difference to ECN

Other applications: coordination



Use switches to maintain a distributed, **fault-tolerant** storage service for **locking/agreement**



NetChain: Scale-Free Sub-RTT Coordination

Xin Jin[1], Xiaozhou Li[2], Haoyu Zhang[3], Nate Foster[2,4],
Jeongkeun Lee[2], Robert Soulé[2,5], Changhoon Kim[2], Ion Stoica[6]
[1]*Johns Hopkins University,* [2]*Barefoot Networks,* [3]*Princeton University,*
[4]*Cornell University,* [5]*Università della Svizzera italiana,* [6] *UC Berkeley*

Abstract

Coordination services are a fundamental building block of modern cloud systems, providing critical functionalities like configuration management and distributed locking. The major challenge is to achieve low latency and high throughput while providing strong consistency and fault-tolerance. Traditional server-based solutions require multiple round-trip times (RTTs) to process a query. This paper presents NetChain, a new approach that provides scale-free sub-RTT coordination in datacenters. NetChain exploits recent advances in programmable switches to store data and process queries entirely in the network data plane. This eliminates the query processing at coordination servers and cuts the end-to-end latency to as little as half of an RTT—clients only experience processing delay from their own software stack plus network delay, which in a datacenter setting is typically much smaller. We design new protocols and algorithms based on chain replication to guar-

DrTM [6], which can process hundreds of millions of transactions per second with a latency of tens of microseconds, crucially depend on fast distributed locking to mediate concurrent access to data partitioned in multiple servers. Unfortunately, acquiring locks becomes a significant bottleneck which severely limits the transaction throughput [7]. This is because servers have to spend their resources on (*i*) processing locking requests and (*ii*) aborting transactions that cannot acquire all locks under high-contention workloads, which can be otherwise used to execute and commit transactions. This is one of the main factors that led to relaxing consistency semantics in many recent large-scale distributed systems [8, 9], and the recent efforts to avoid coordination by leveraging application semantics [10, 11]. While these systems are successful in achieving high throughput, unfortunately, they restrict the programming model and complicate the application development. A fast coordination service would enable high transaction throughput without any of these compromises.

USENIX NSDI 2018

Summary

Why do we need programmable data plane?

- OpenFlow is centralized around known packet header fields and does not support new protocols
- Updating the hardware to support new protocols is time-consuming

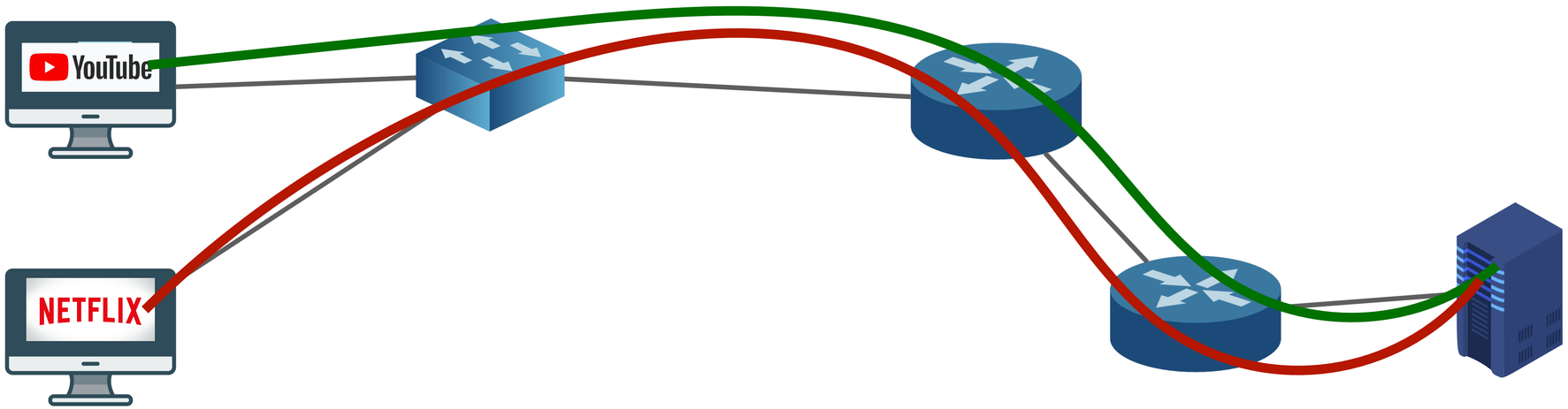
How to enable data plane programmability?

- Abstraction: PISA architecture
- Language: P4

NetCache: leveraging programmable data plane for accelerating key-value store

- Use PISA parser to obtain query information
- Use multiple register arrays to hold items with variable length
- Use CountMin sketch and Bloom filter to report hot items

Next time: video streaming



Video is almost
58% of the total downstream volume of traffic on the internet