Advanced Topics in Communication Networks

Programming Network Data Planes

Laurent Vanbever

nsg.ee.ethz.ch

ETH Zürich

Oct 29 2019

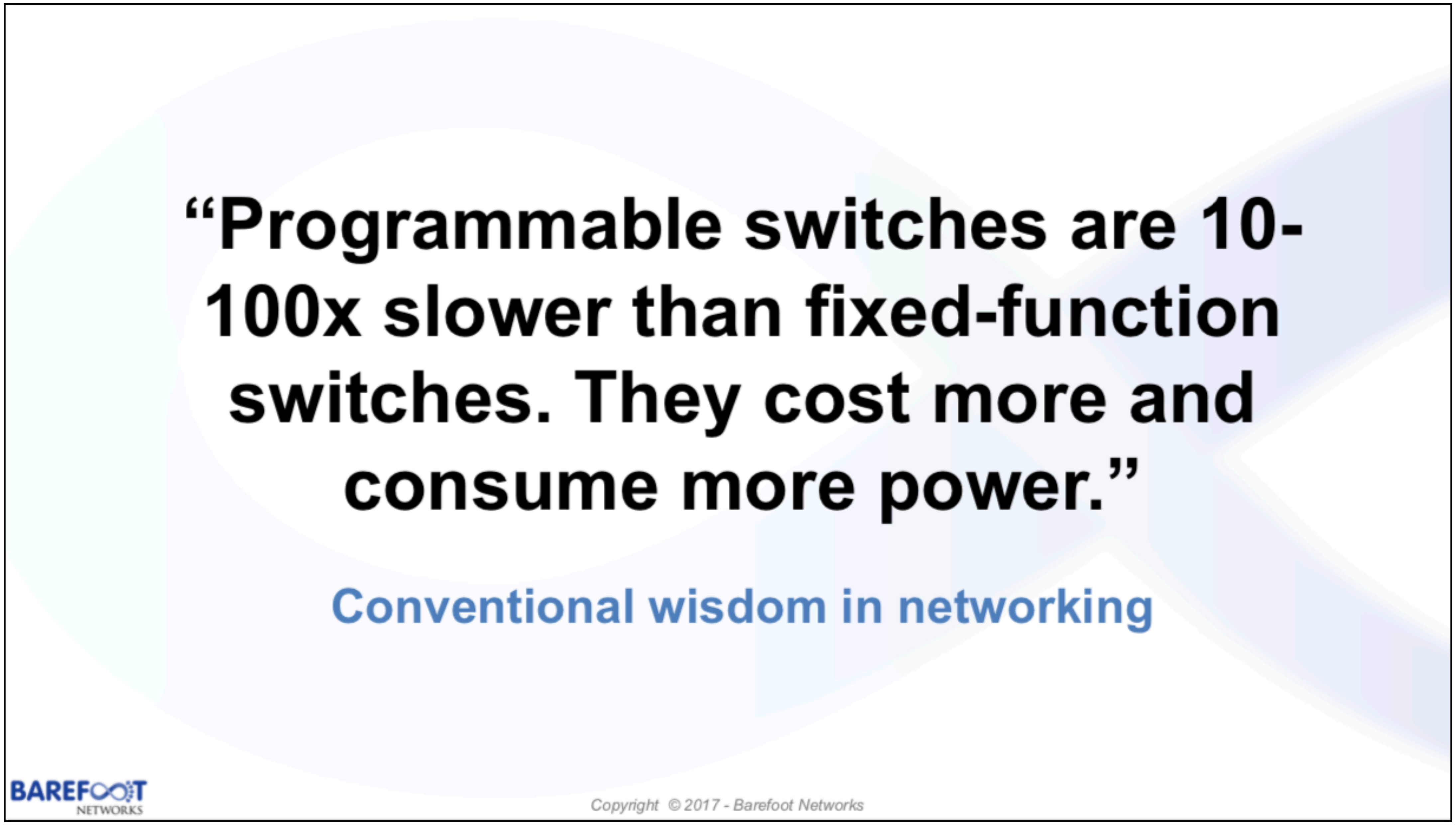
Last week on

Advanced Topics in Communication Networks

P4 hardware target

P4-based applications

How do we build a *fast* reprogrammable switch?



“Programmable switches are 10-100x slower than fixed-function switches. They cost more and consume more power.”

Conventional wisdom in networking

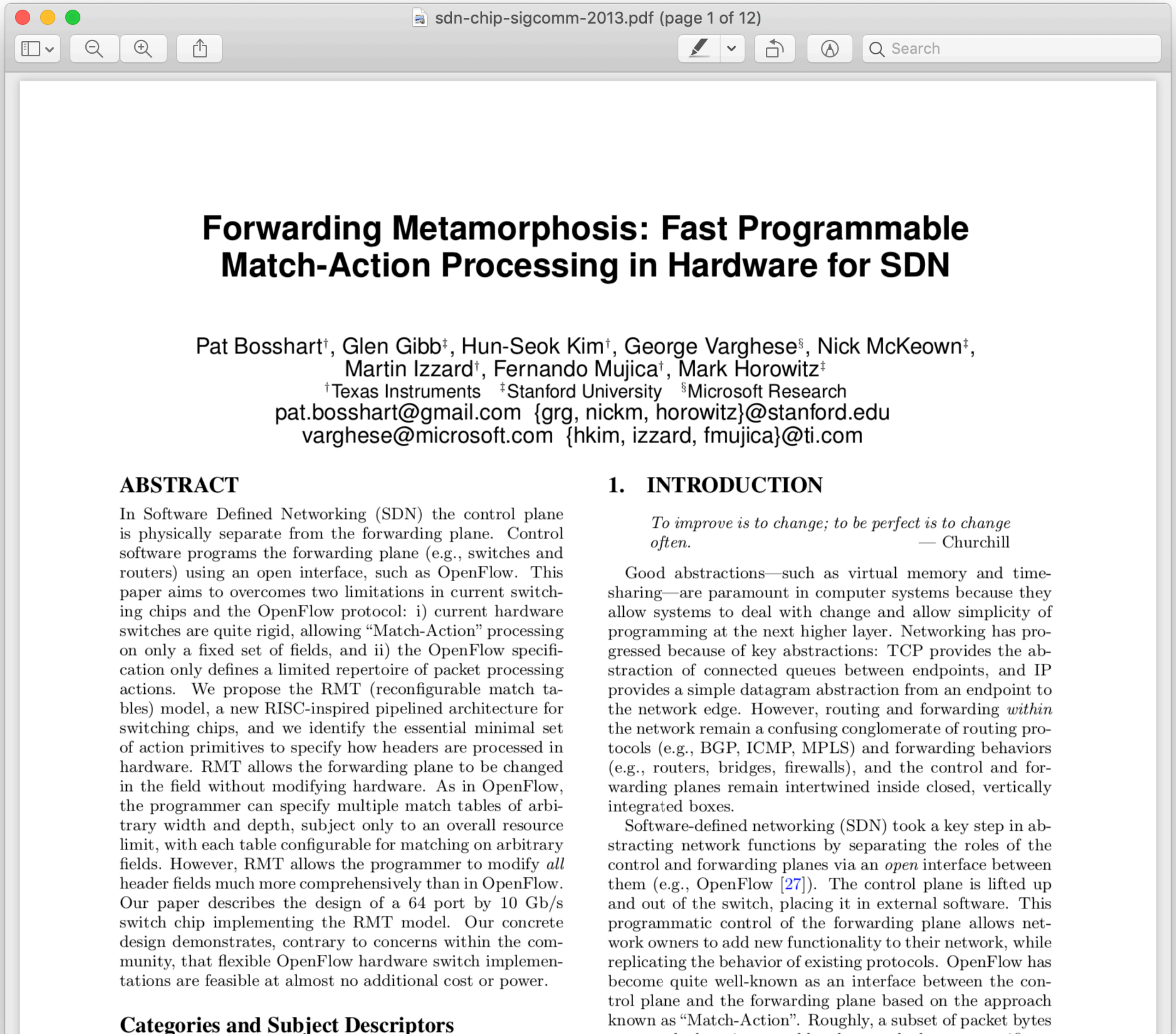
Source: Programmable Data Planes at Terabit Speeds, Vladimir Gurevich, 2017

How can we allow network programmability in the field, at reasonable cost, and without sacrificing speed

supporting Tbps of backplane throughput

Let's look at a concrete design:

Reconfigurable Match Tables (RMT)



Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN

Pat Bosshart[†], Glen Gibb[‡], Hun-Seok Kim[†], George Varghese[§], Nick McKeown[‡],
Martin Izzard[†], Fernando Mujica[†], Mark Horowitz[‡]
[†]Texas Instruments [‡]Stanford University [§]Microsoft Research
pat.bosshart@gmail.com {grg, nickm, horowitz}@stanford.edu
varghese@microsoft.com {hkim, izzard, fmujica}@ti.com

ABSTRACT

In Software Defined Networking (SDN) the control plane is physically separate from the forwarding plane. Control software programs the forwarding plane (e.g., switches and routers) using an open interface, such as OpenFlow. This paper aims to overcomes two limitations in current switching chips and the OpenFlow protocol: i) current hardware switches are quite rigid, allowing "Match-Action" processing on only a fixed set of fields, and ii) the OpenFlow specification only defines a limited repertoire of packet processing actions. We propose the RMT (reconfigurable match tables) model, a new RISC-inspired pipelined architecture for switching chips, and we identify the essential minimal set of action primitives to specify how headers are processed in hardware. RMT allows the forwarding plane to be changed in the field without modifying hardware. As in OpenFlow, the programmer can specify multiple match tables of arbitrary width and depth, subject only to an overall resource limit, with each table configurable for matching on arbitrary fields. However, RMT allows the programmer to modify *all* header fields much more comprehensively than in OpenFlow. Our paper describes the design of a 64 port by 10 Gb/s switch chip implementing the RMT model. Our concrete design demonstrates, contrary to concerns within the community, that flexible OpenFlow hardware switch implementations are feasible at almost no additional cost or power.

Categories and Subject Descriptors

1. INTRODUCTION

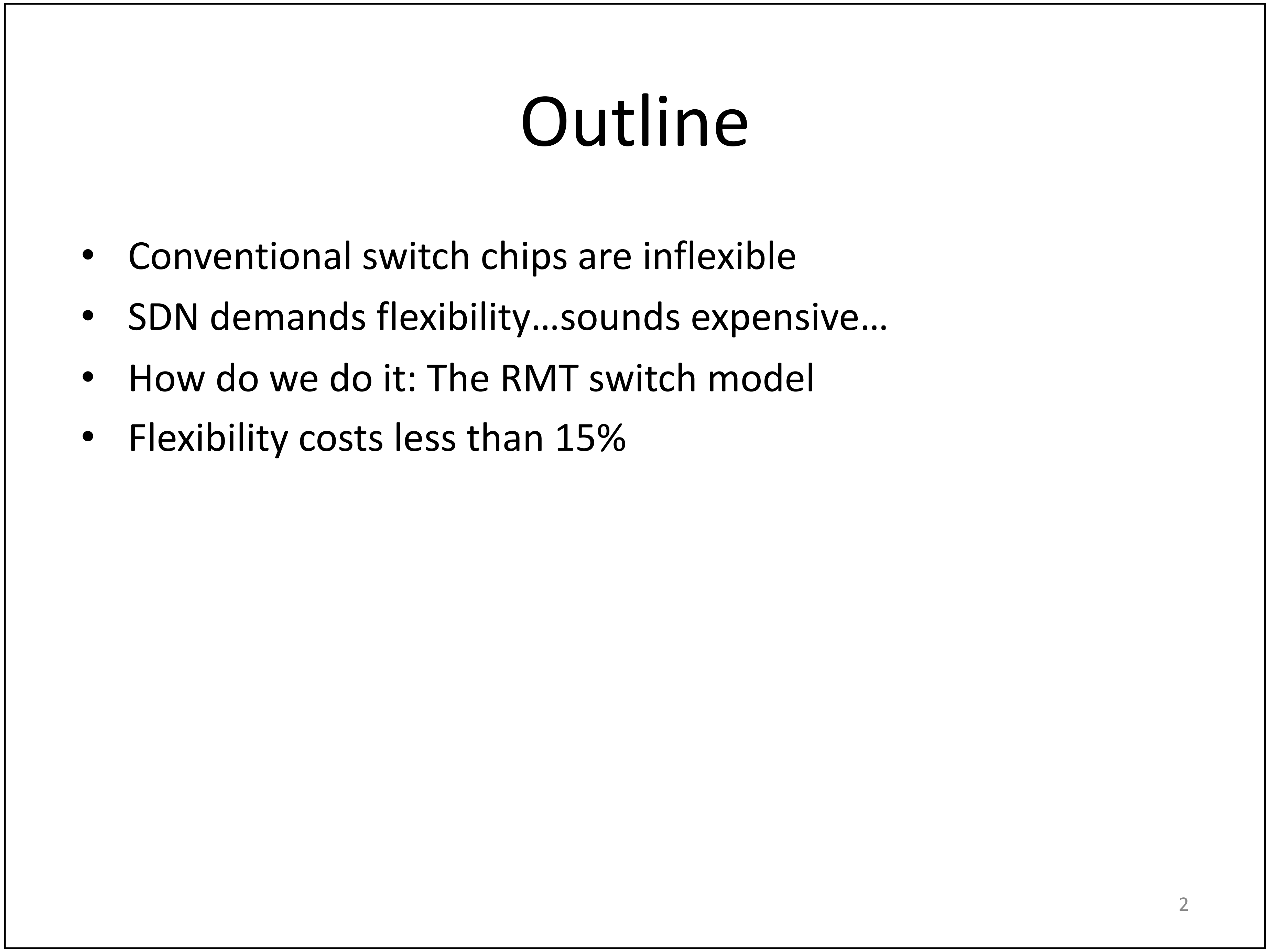
To improve is to change; to be perfect is to change often. — Churchill

Good abstractions—such as virtual memory and time-sharing—are paramount in computer systems because they allow systems to deal with change and allow simplicity of programming at the next higher layer. Networking has progressed because of key abstractions: TCP provides the abstraction of connected queues between endpoints, and IP provides a simple datagram abstraction from an endpoint to the network edge. However, routing and forwarding *within* the network remain a confusing conglomerate of routing protocols (e.g., BGP, ICMP, MPLS) and forwarding behaviors (e.g., routers, bridges, firewalls), and the control and forwarding planes remain intertwined inside closed, vertically integrated boxes.

Software-defined networking (SDN) took a key step in abstracting network functions by separating the roles of the control and forwarding planes via an *open* interface between them (e.g., OpenFlow [27]). The control plane is lifted up and out of the switch, placing it in external software. This programmatic control of the forwarding plane allows network owners to add new functionality to their network, while replicating the behavior of existing protocols. OpenFlow has become quite well-known as an interface between the control plane and the forwarding plane based on the approach known as "Match-Action". Roughly, a subset of packet bytes

[SIGCOMM'13]

The paper argues that flexibility does not come at the price of performance or cost

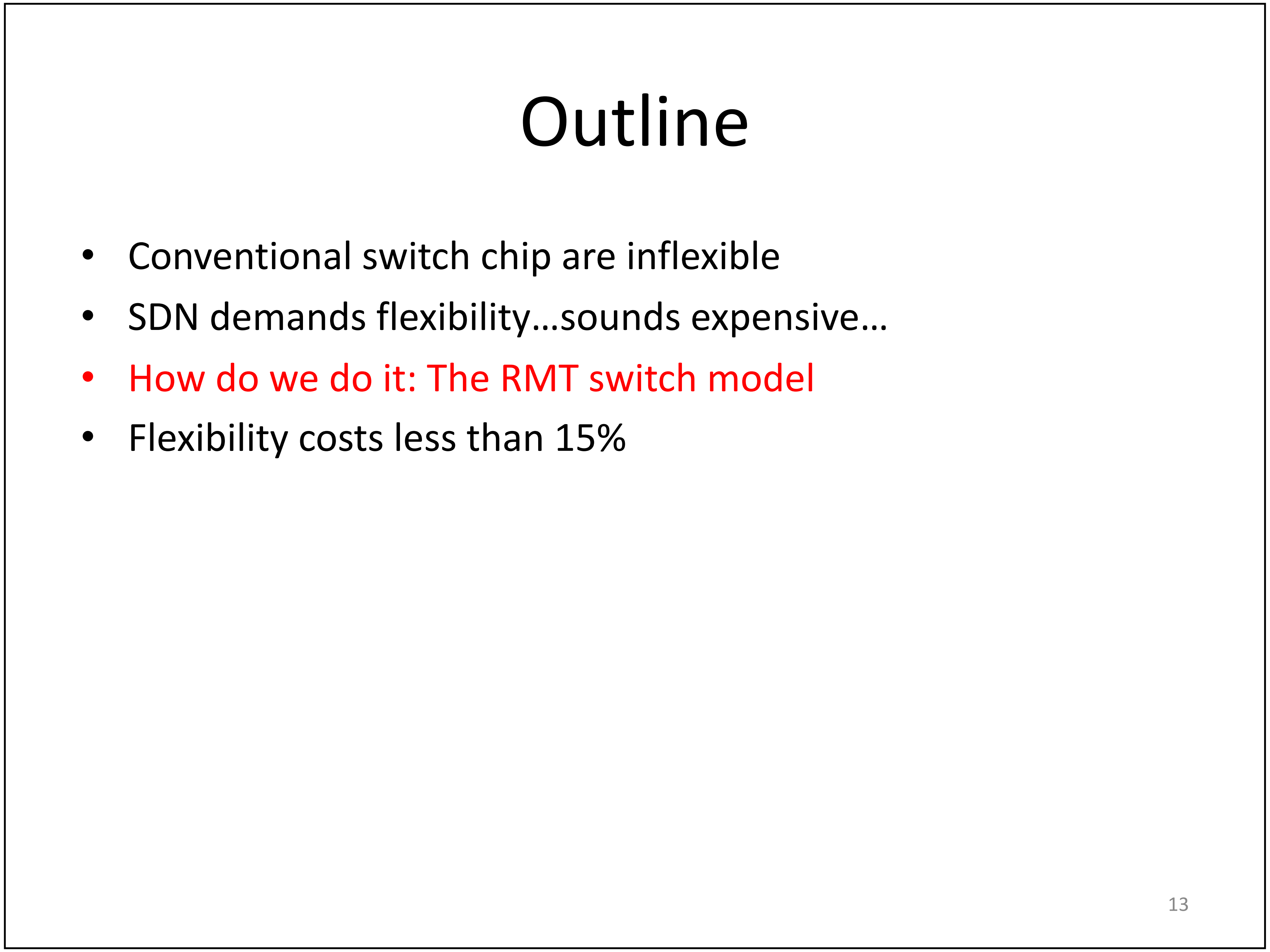


Outline

- Conventional switch chips are inflexible
- SDN demands flexibility…sounds expensive…
- How do we do it: The RMT switch model
- Flexibility costs less than 15%

Enter…

Reconfigurable Match Tables (RMT)



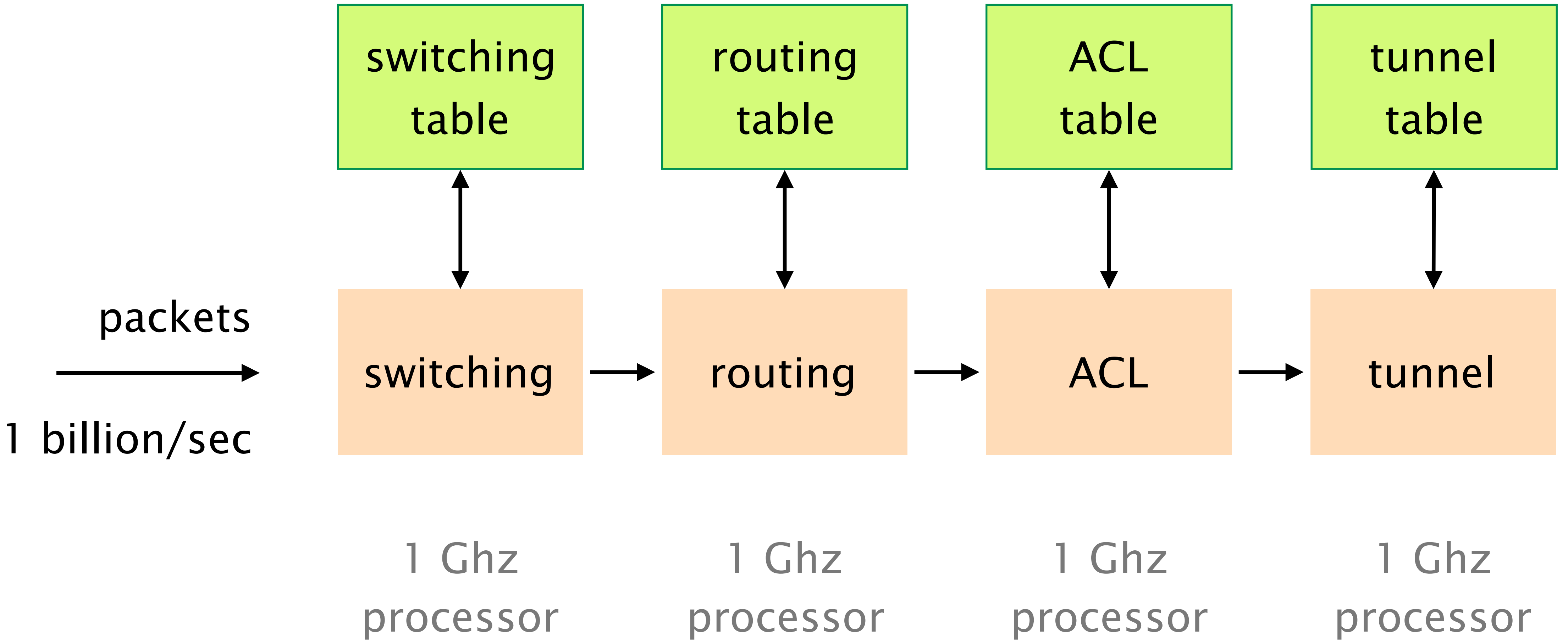
Outline

- Conventional switch chip are inflexible
- SDN demands flexibility…sounds expensive…
- How do we do it: The RMT switch model
- Flexibility costs less than 15%

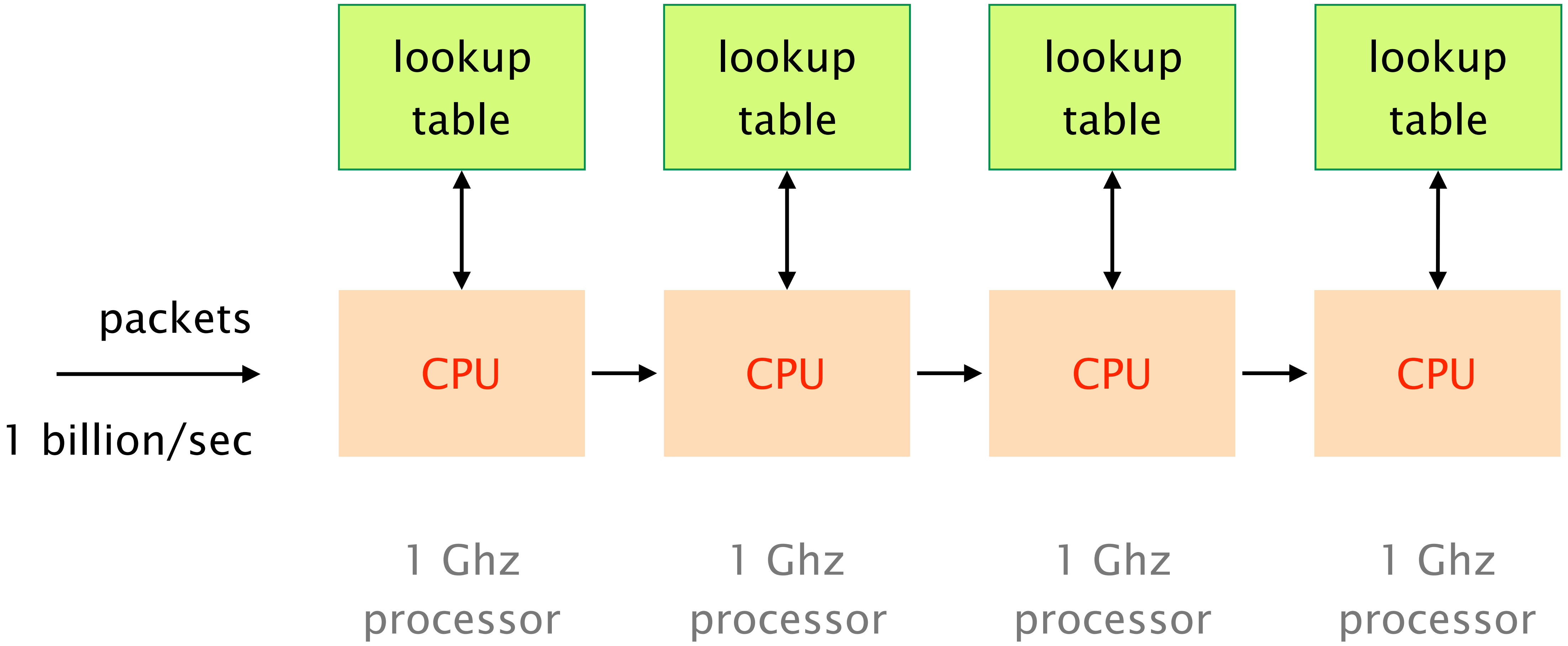
What kind of switch architecture could support flexibility and yet run at Terabits per second?

Throughput aggregate	1 Tbps
Packet size average	1000 bits
# operations per packet (avg.)	10
Requirements	10 billion op./second

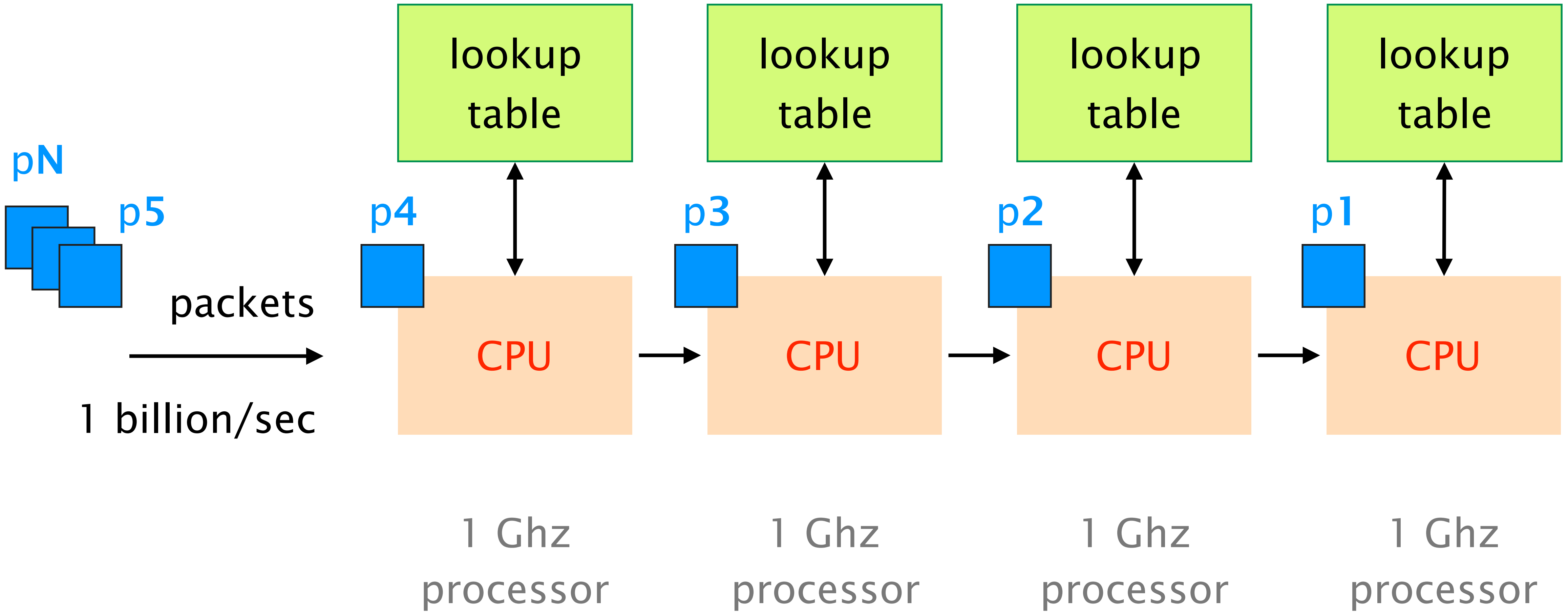
Pipelined architectures organize processing through a sequence of processing units and local memory



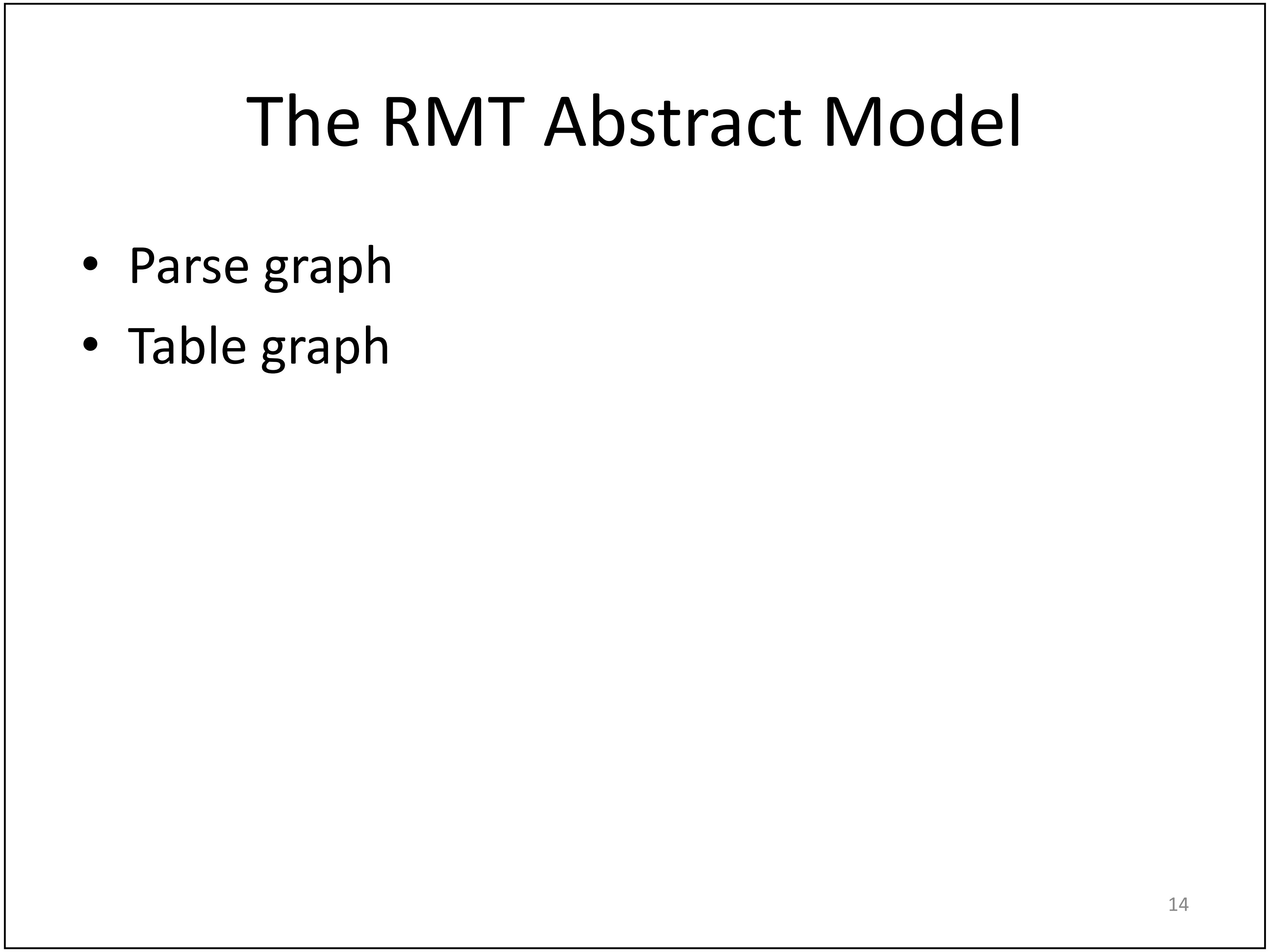
For flexibility,
each processing unit/memory can be made generic



Each CPU can process distinct packets, with up to 10 packets going through the pipeline simultaneously



The runtime behavior of the parser & the match stages is defined through the RMT abstract model

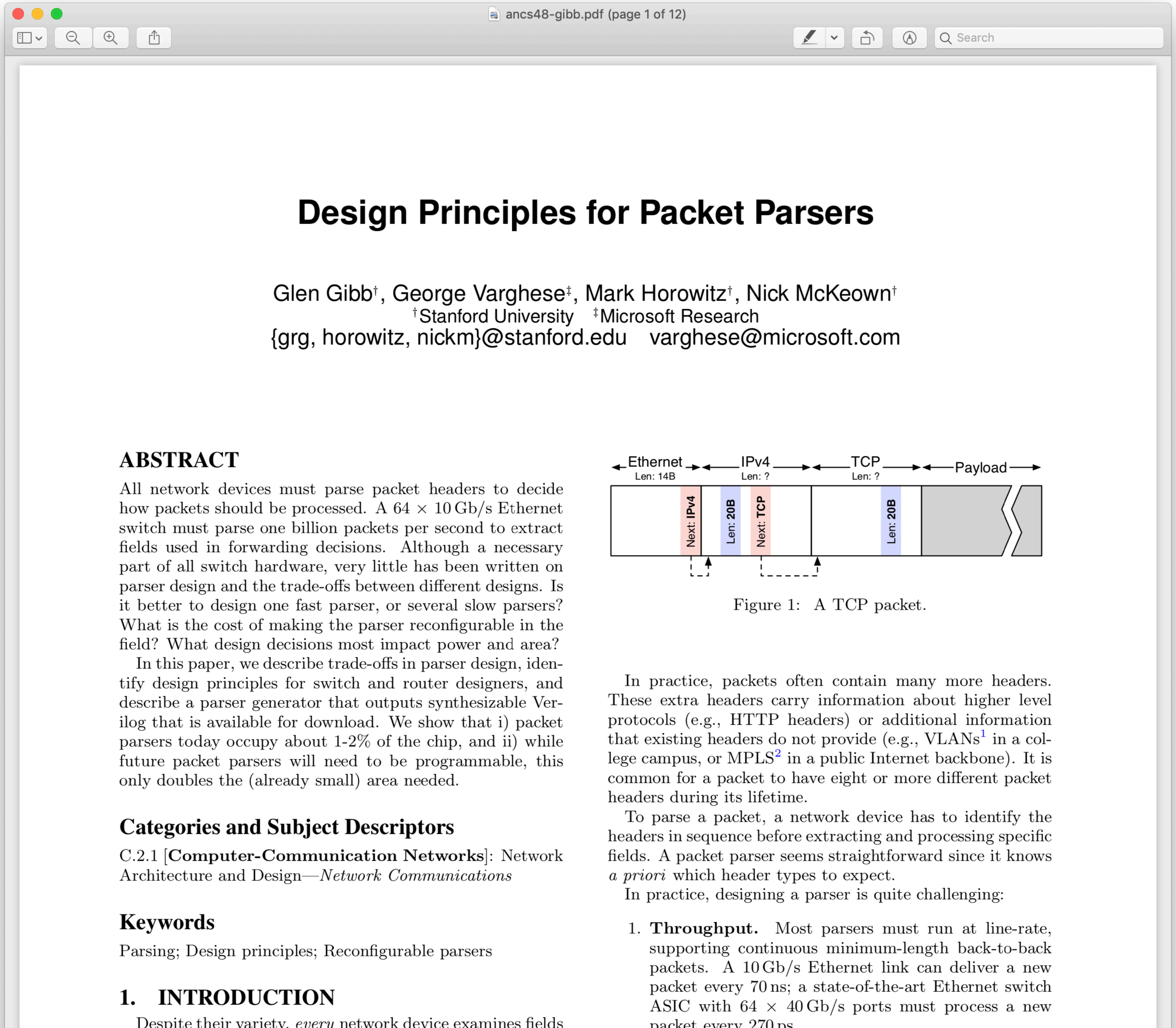


The RMT Abstract Model

- Parse graph
- Table graph

How do we implement in hardware
a programmable parser and a logical pipeline?

How do we implement in hardware a programmable parser and a logical pipeline?



Design Principles for Packet Parsers

Glen Gibb[†], George Varghese[‡], Mark Horowitz[†], Nick McKeown[†]
[†]Stanford University [‡]Microsoft Research
{grg, horowitz, nickm}@stanford.edu varghese@microsoft.com

ABSTRACT

All network devices must parse packet headers to decide how packets should be processed. A 64 × 10 Gb/s Ethernet switch must parse one billion packets per second to extract fields used in forwarding decisions. Although a necessary part of all switch hardware, very little has been written on parser design and the trade-offs between different designs. Is it better to design one fast parser, or several slow parsers? What is the cost of making the parser reconfigurable in the field? What design decisions most impact power and area?

In this paper, we describe trade-offs in parser design, identify design principles for switch and router designers, and describe a parser generator that outputs synthesizable Verilog that is available for download. We show that i) packet parsers today occupy about 1-2% of the chip, and ii) while future packet parsers will need to be programmable, this only doubles the (already small) area needed.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Network Communications*

Keywords

Parsing; Design principles; Reconfigurable parsers

1. INTRODUCTION

Despite their variety, *every* network device examines fields

Figure 1: A TCP packet.

In practice, packets often contain many more headers. These extra headers carry information about higher level protocols (e.g., HTTP headers) or additional information that existing headers do not provide (e.g., VLANs[1] in a college campus, or MPLS[2] in a public Internet backbone). It is common for a packet to have eight or more different packet headers during its lifetime.

To parse a packet, a network device has to identify the headers in sequence before extracting and processing specific fields. A packet parser seems straightforward since it knows *a priori* which header types to expect.

In practice, designing a parser is quite challenging:

1. **Throughput.** Most parsers must run at line-rate, supporting continuous minimum-length back-to-back packets. A 10 Gb/s Ethernet link can deliver a new packet every 70 ns; a state-of-the-art Ethernet switch ASIC with 64 × 40 Gb/s ports must process a new packet every 270 ps.

[ANCS'13]

Parsing is the (complex) process of identifying and extracting the appropriate fields in a packet header

Throughput

Parser must run at line-rate

parse 1 packet every 70 ns on a 10 Gbps link

Dependency

Parsing involves sequential processing
as headers typically point to the next one

Incompleteness

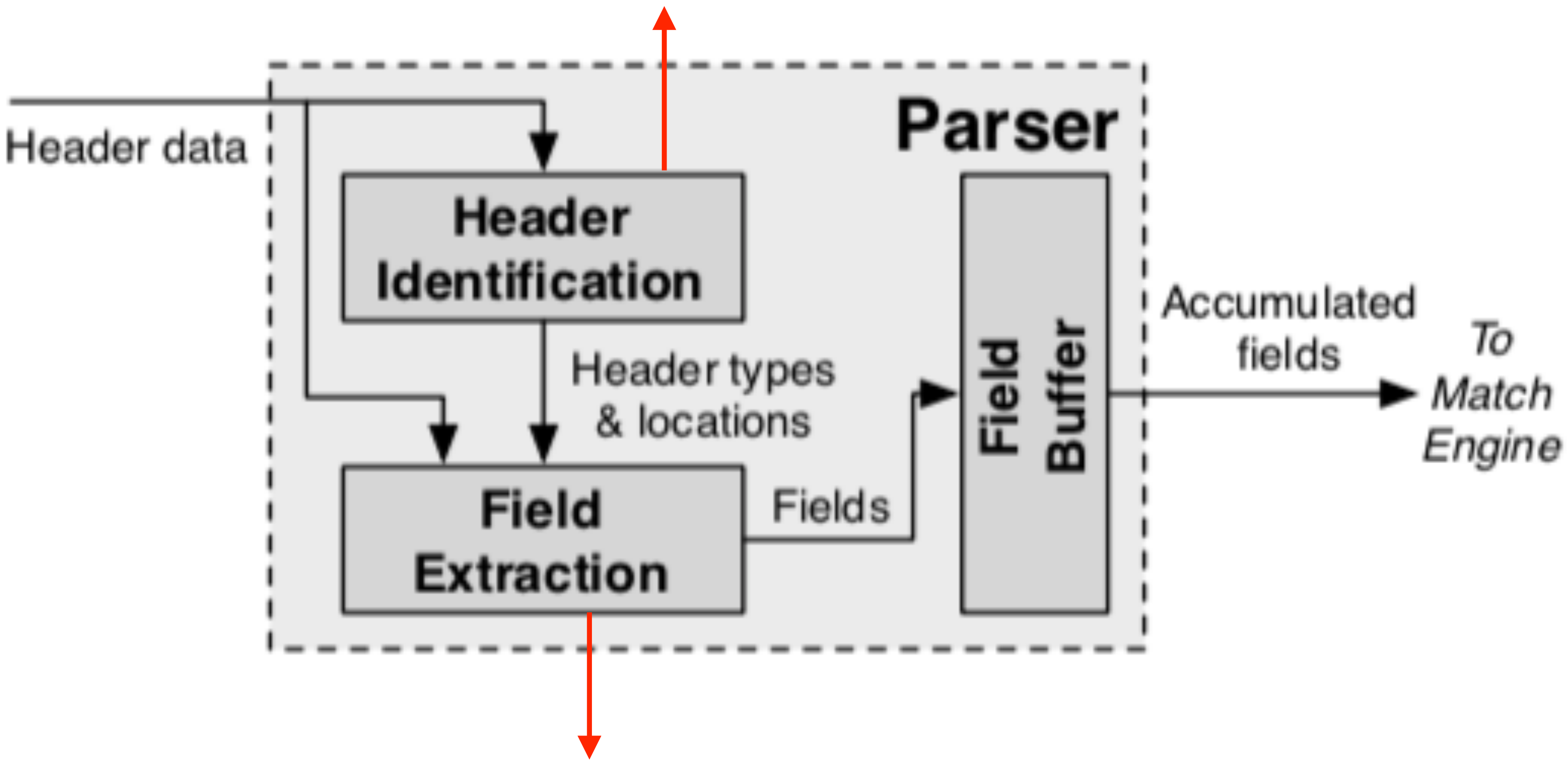
Some headers do not even identify
the subsequent header

Heterogeneity

Many header formats exist that
can appear in various orders/locations

A parser can be divided into two separate blocks: header identification and field extraction

implements the parse graph's
state machine

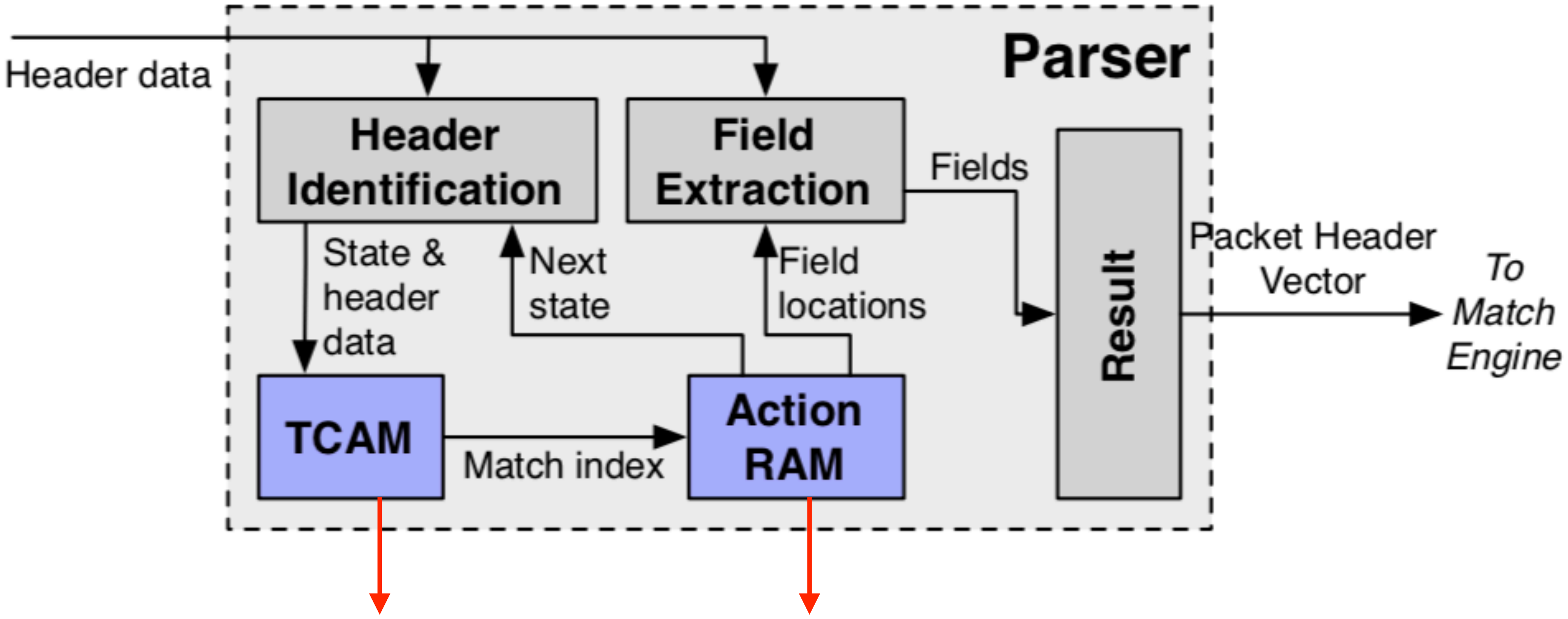


extracts the chosen fields
from identified headers

Source: Design Principles for Packet Parsers, Gibb et al.

In a programmable parser, the two modules rely on runtime information instead of hard-coded logic

stored in memory,
e.g. in RAM and/or TCAM



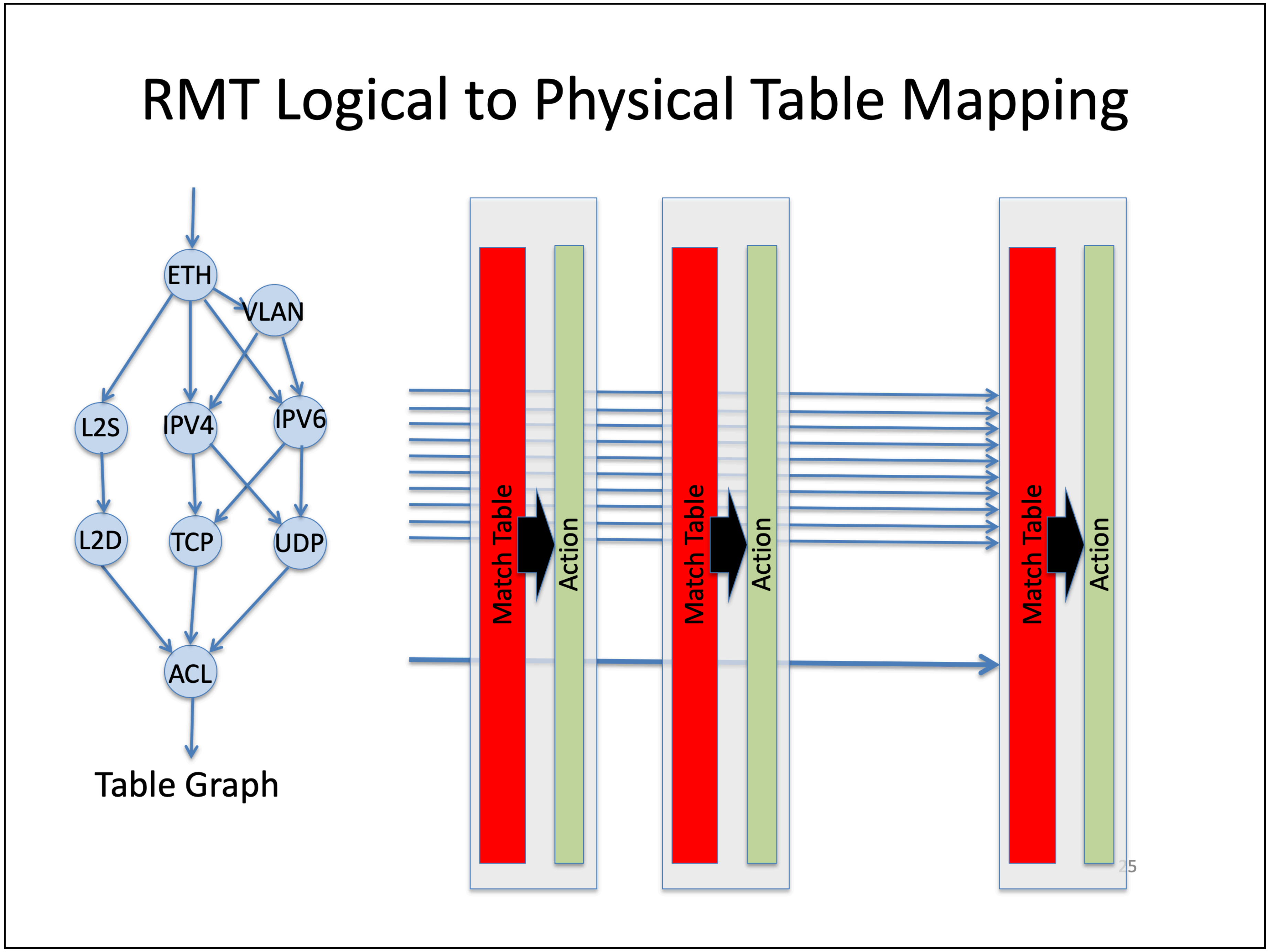
stores the bit sequences
that identify the headers

stores the next state,
the fields to extract,
and any other data (if any)

Source: Design Principles for Packet Parsers, Gibb et al.

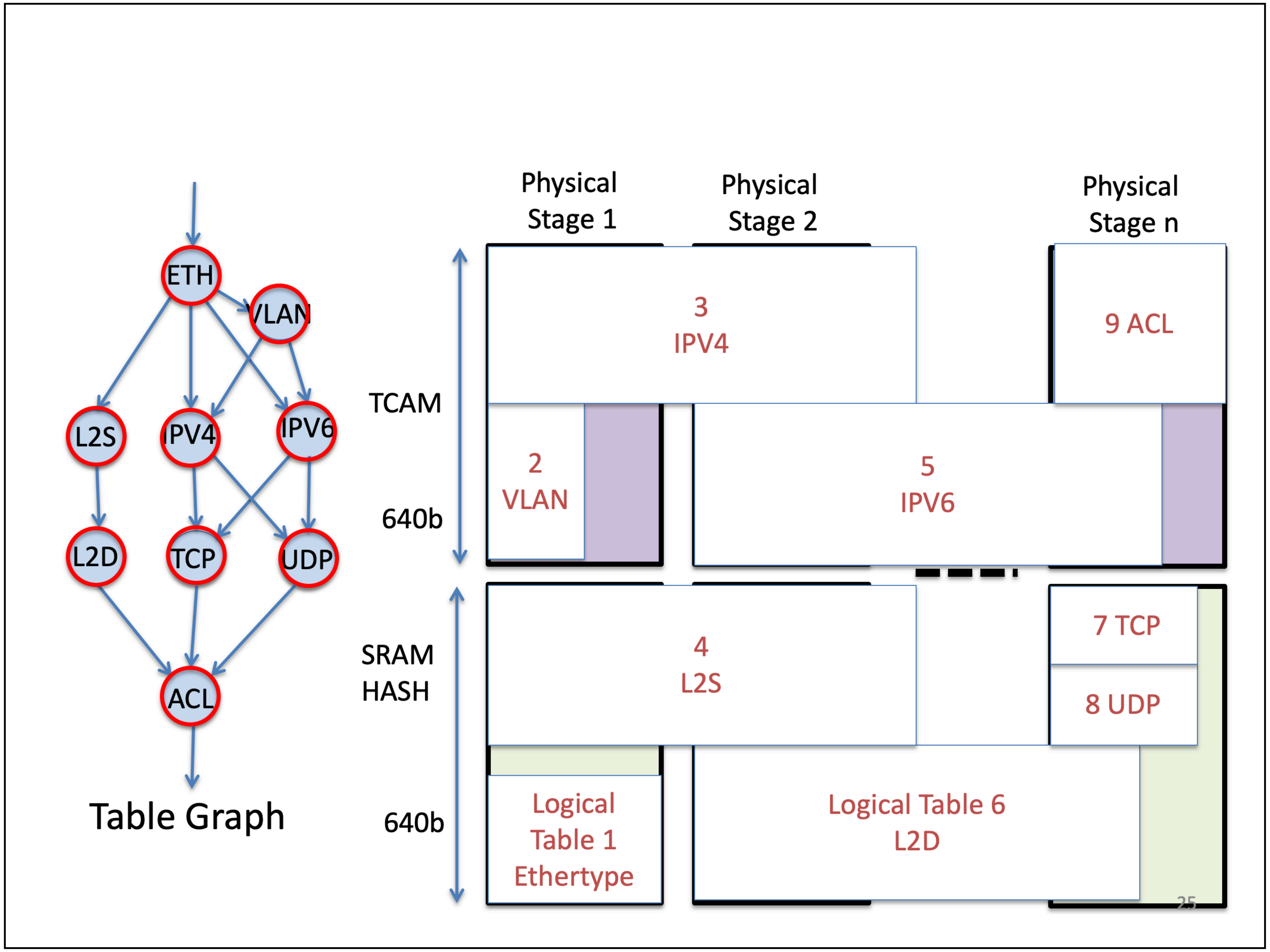
How do we implement in hardware
a programmable parser and a logical pipeline?

A compiler translates a given RMT logical pipeline (specified in P4) into a physical one



RMT Logical to Physical Table Mapping

The compiler maps each individual logical stage to one or more physical stage.



ETH
VLAN
L2S
IPV4
IPV6
L2D
TCP
UDP
ACL

Table Graph

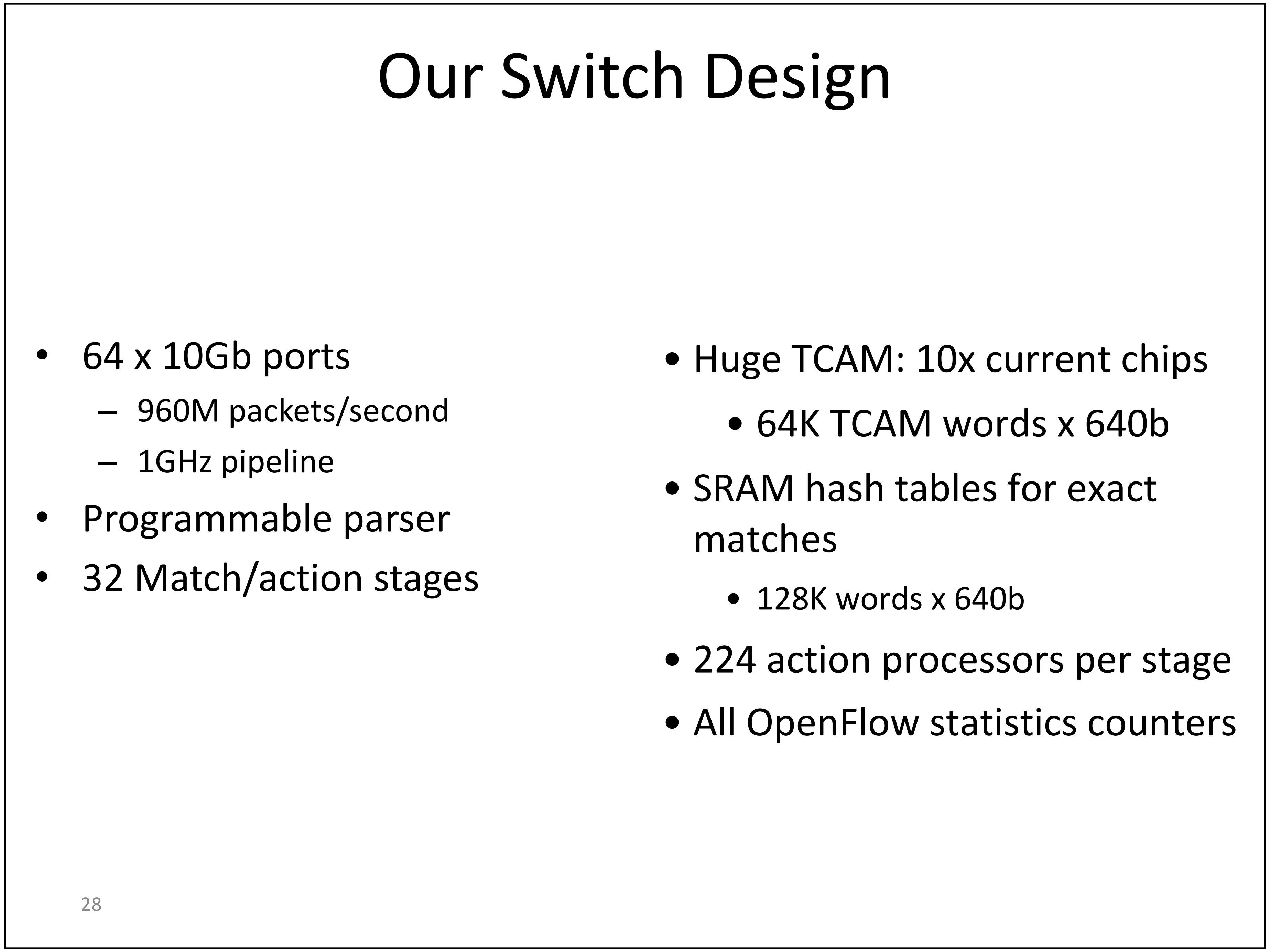
Physical Stage 1
Physical Stage 2
Physical Stage n

TCAM
640b

SRAM HASH
640b

3 IPV4
9 ACL
2 VLAN
5 IPV6
4 L2S
7 TCP
8 UDP
Logical Table 1 Ethertype
Logical Table 6 L2D

The RMT pipeline in a few statistics

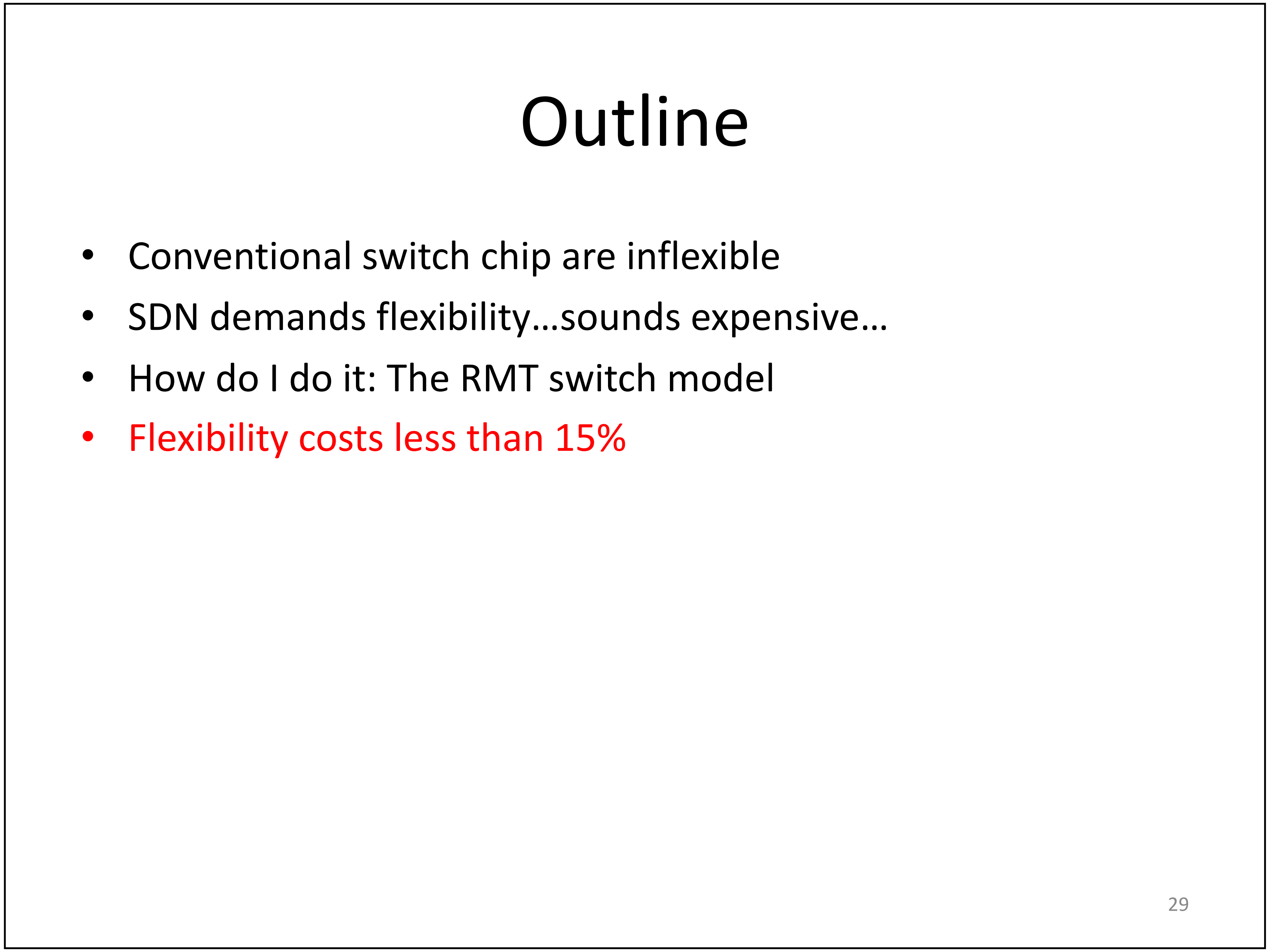


Our Switch Design

- 64 x 10Gb ports
  - 960M packets/second
  - 1GHz pipeline
- Programmable parser
- 32 Match/action stages

- Huge TCAM: 10x current chips
  - 64K TCAM words x 640b
- SRAM hash tables for exact matches
  - 128K words x 640b
- 224 action processors per stage
- All OpenFlow statistics counters

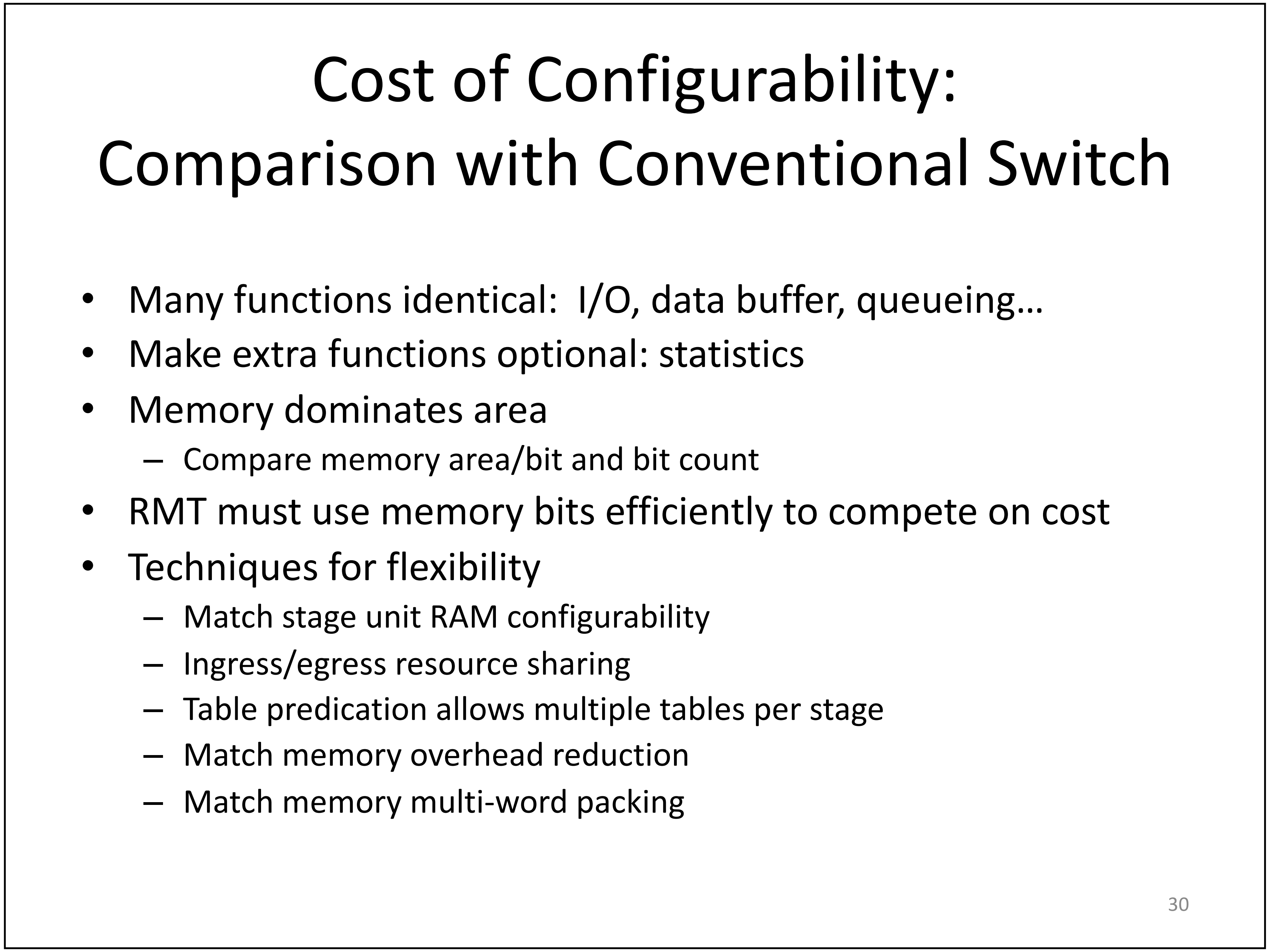
Building a RMT pipeline is only 15% more expensive than building a fixed-function switching pipeline



Outline

- Conventional switch chip are inflexible
- SDN demands flexibility…sounds expensive…
- How do I do it: The RMT switch model
- Flexibility costs less than 15%

The biggest cost is the memory… *not* the processing logic

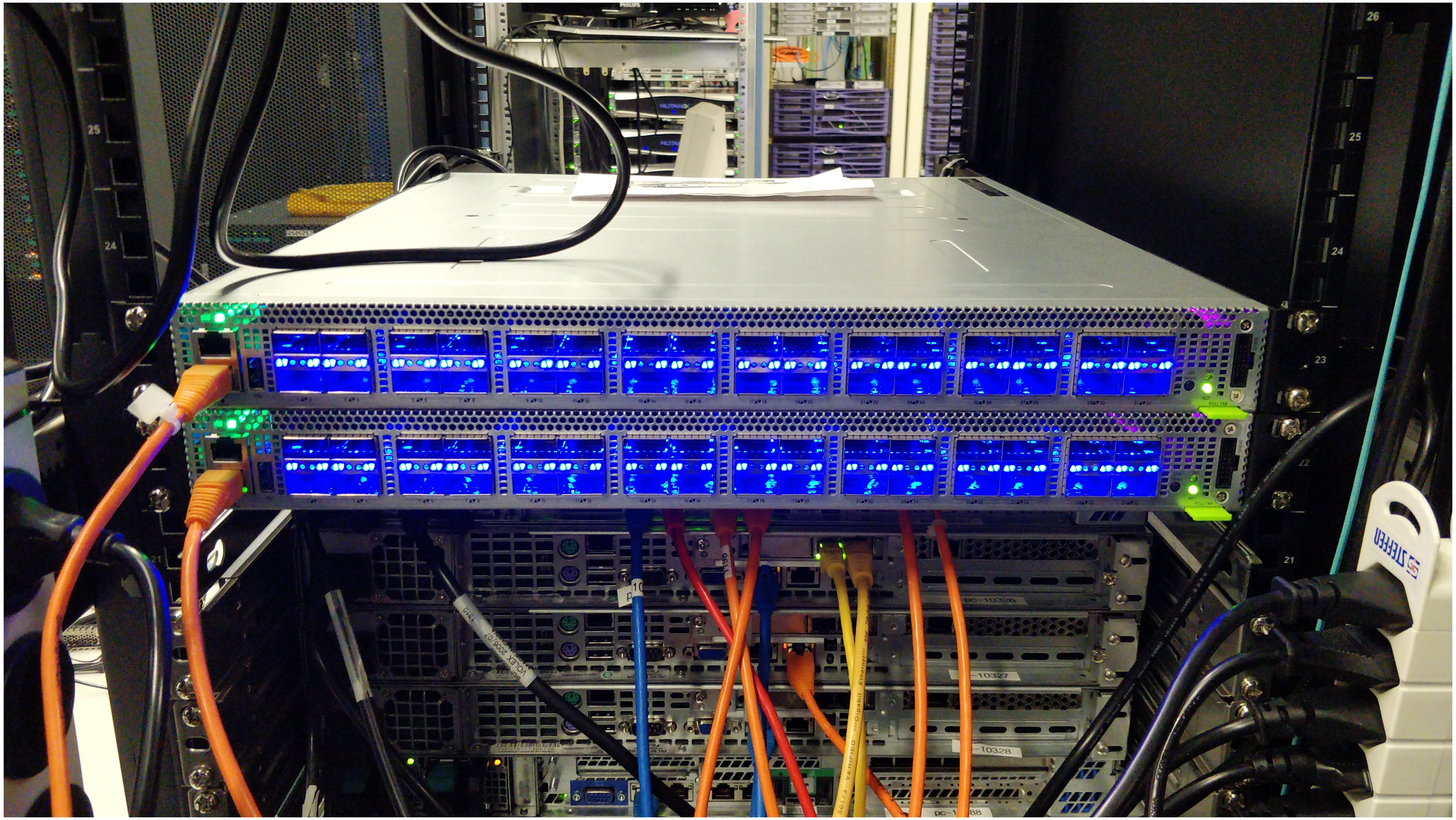


Cost of Configurability: Comparison with Conventional Switch

- Many functions identical: I/O, data buffer, queueing…
- Make extra functions optional: statistics
- Memory dominates area
  - Compare memory area/bit and bit count
- RMT must use memory bits efficiently to compete on cost
- Techniques for flexibility
  - Match stage unit RAM configurability
  - Ingress/egress resource sharing
  - Table predication allows multiple tables per stage
  - Match memory overhead reduction
  - Match memory multi-word packing

That was just an academic paper

Let's look at a real flexible pipeline



A small subset of our lab @ITET with two Tofino 3.2 Tbps, 32x 100 GbE QSFP28

That was just an academic paper

Let's look at a real flexible pipeline

BAREFOOT NETWORKS

Programmable Data Plane at Terabit Speeds

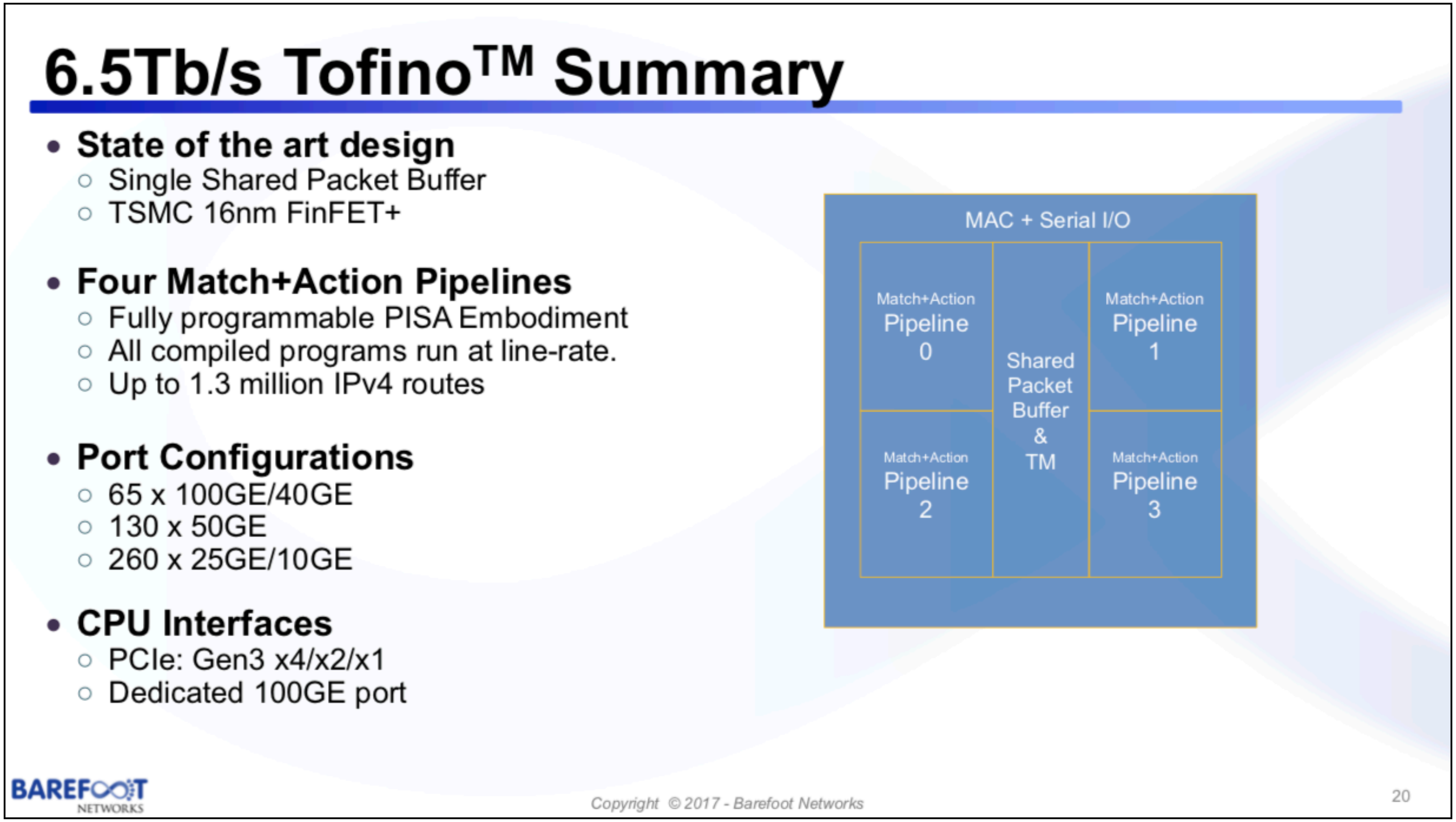
Vladimir Gurevich
May 16, 2017

BAREFOOT NETWORKS

Source: Programmable Data Planes at Terabit Speeds, Vladimir Gurevich, 2017

Barefoot Tofino 6.5 Tbps backplane

several billion packets per second at line rate



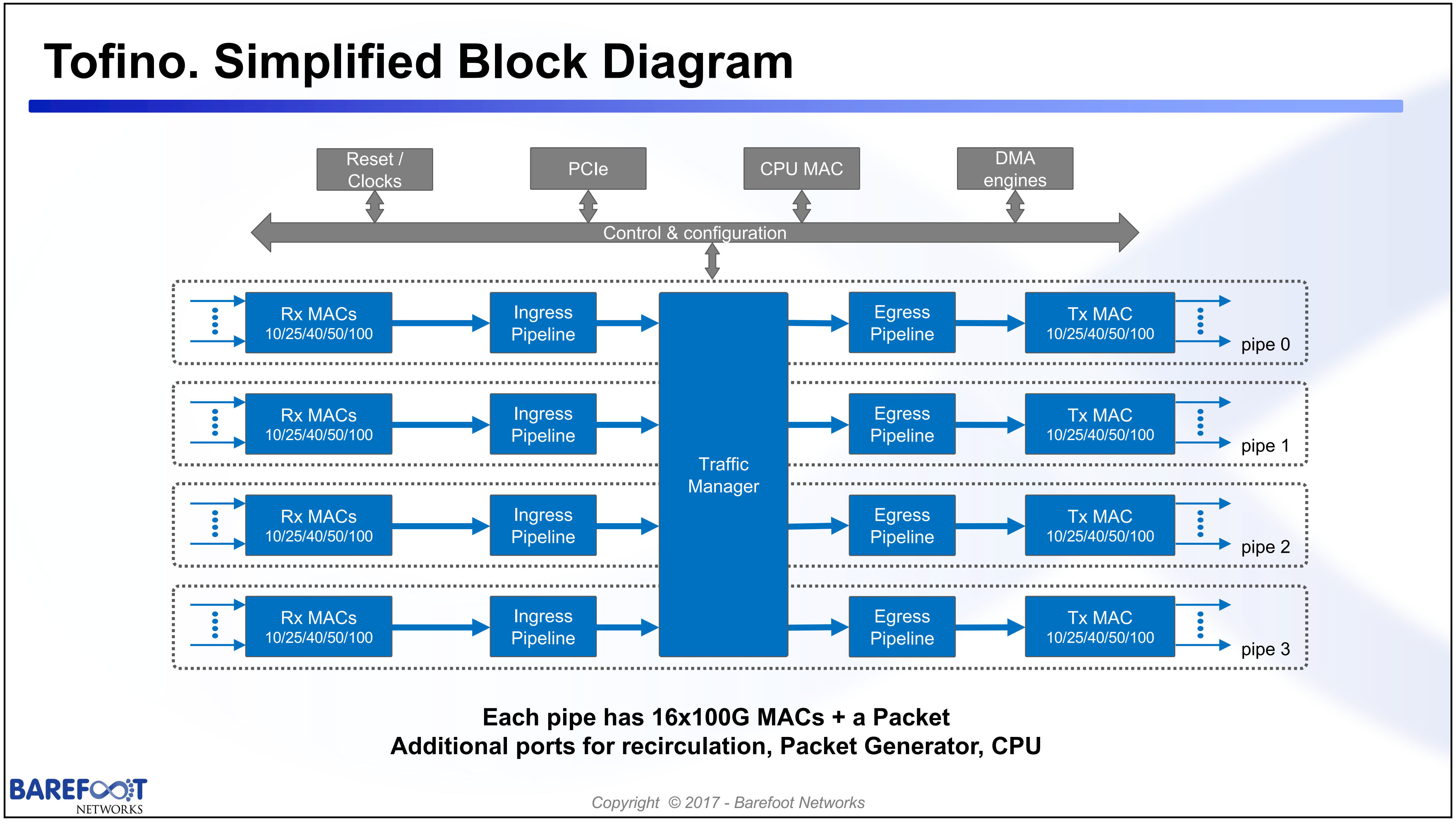
6.5Tb/s Tofino™ Summary

- **State of the art design**
  - Single Shared Packet Buffer
  - TSMC 16nm FinFET+
- **Four Match+Action Pipelines**
  - Fully programmable PISA Embodiment
  - All compiled programs run at line-rate.
  - Up to 1.3 million IPv4 routes
- **Port Configurations**
  - 65 x 100GE/40GE
  - 130 x 50GE
  - 260 x 25GE/10GE
- **CPU Interfaces**
  - PCIe: Gen3 x4/x2/x1
  - Dedicated 100GE port

Source: Programmable Data Planes at Terabit Speeds, Vladimir Gurevich, 2017

Barefoot Tofino 6.5 Tbps backplane

several billion packets per second at line rate



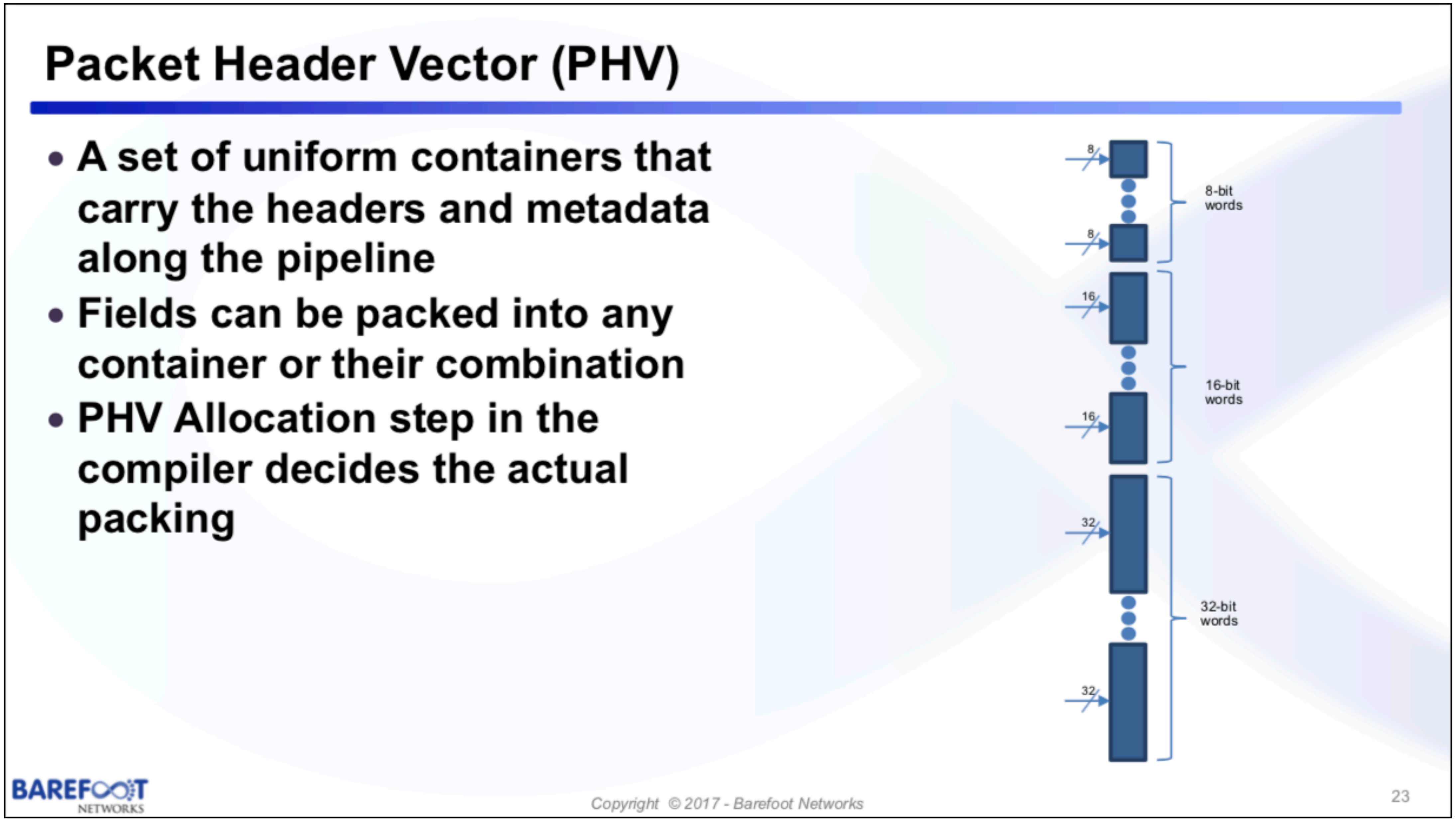
Tofino. Simplified Block Diagram

Each pipe has 16x100G MACs + a Packet
Additional ports for recirculation, Packet Generator, CPU

BAREFOOT NETWORKS

Source: Programmable Data Planes at Terabit Speeds, Vladimir Gurevich, 2017

Tofino relies on Packet Header Vector (PHV) to pass states between stages



Packet Header Vector (PHV)

- A set of uniform containers that carry the headers and metadata along the pipeline
- Fields can be packed into any container or their combination
- PHV Allocation step in the compiler decides the actual packing

8
8
8-bit words
16
16
16-bit words
32
32
32-bit words

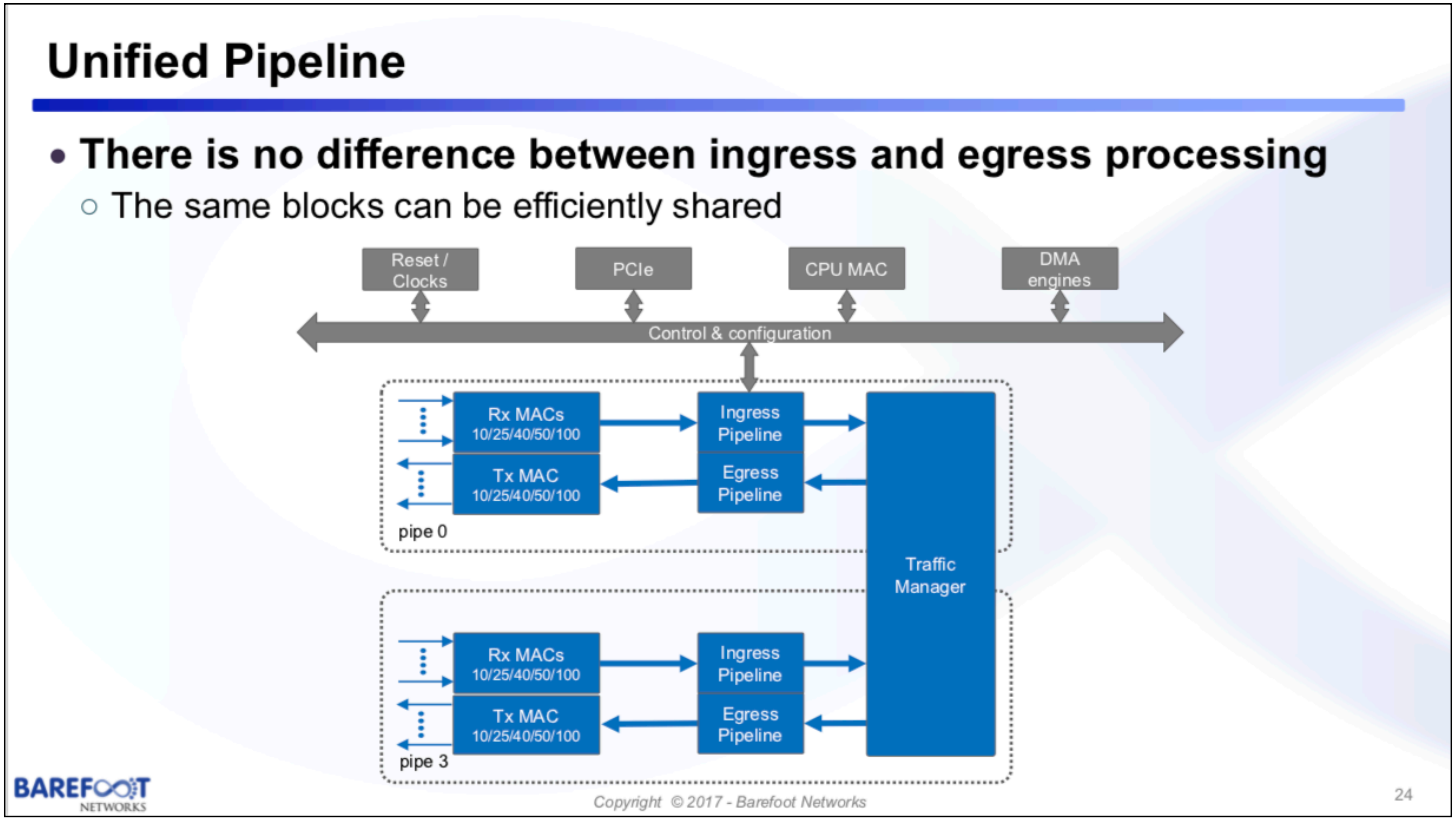
BAREFOOT NETWORKS

Copyright © 2017 - Barefoot Networks

23

Source: Programmable Data Planes at Terabit Speeds, Vladimir Gurevich, 2017

Tofino uses a folded pipeline in which the *same* stages are used for both the ingress and the egress pipeline



Unified Pipeline

- **There is no difference between ingress and egress processing**
  - ○ The same blocks can be efficiently shared

Reset / Clocks

PCIe

CPU MAC

DMA engines

Control & configuration

Rx MACs 10/25/40/50/100

Ingress Pipeline

Tx MAC 10/25/40/50/100

Egress Pipeline

pipe 0

Traffic Manager

Rx MACs 10/25/40/50/100

Ingress Pipeline

Tx MAC 10/25/40/50/100

Egress Pipeline

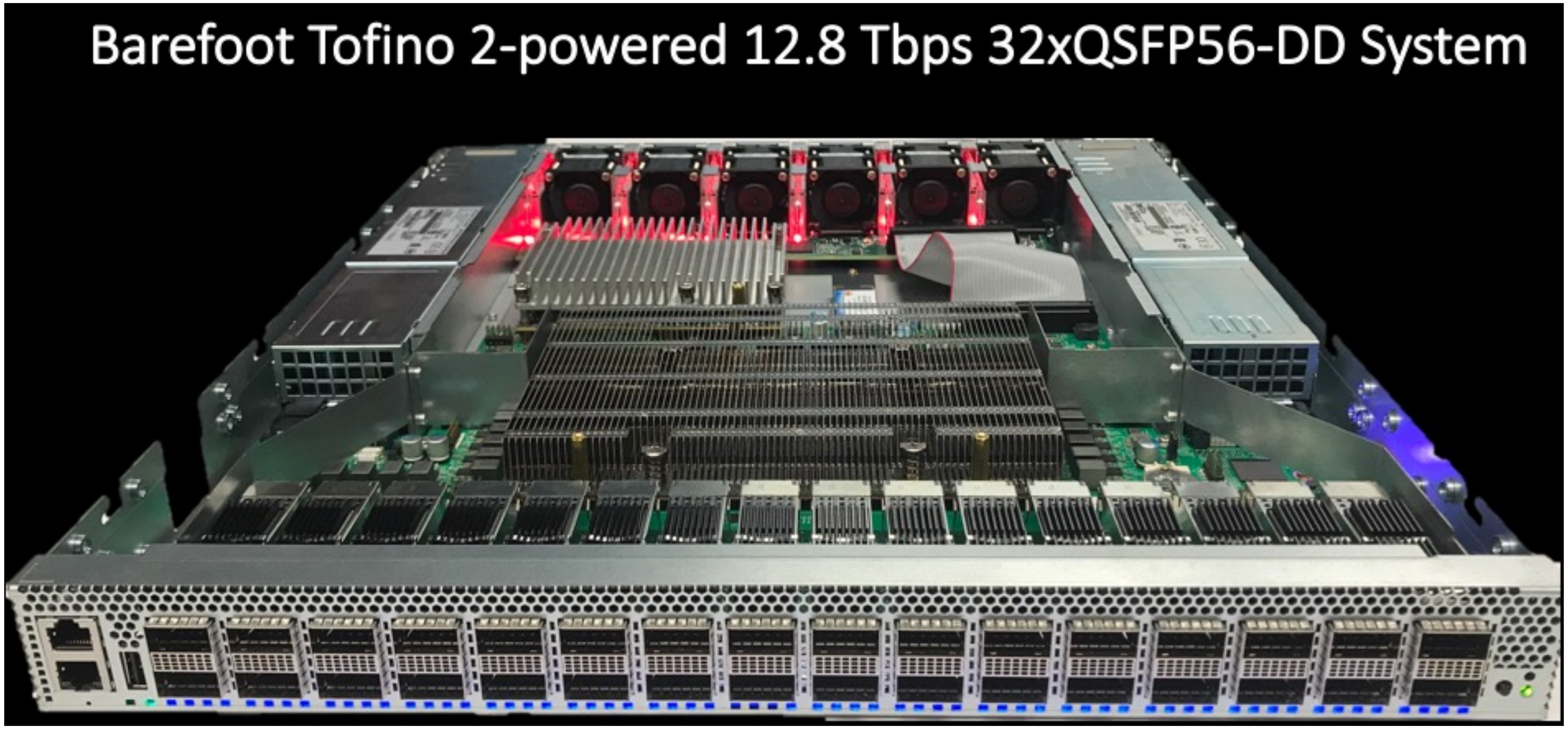
pipe 3

BAREFOOT NETWORKS

Source: Programmable Data Planes at Terabit Speeds, Vladimir Gurevich, 2017

What's next?

Tofino 2: 12.8 Tbps (7 nm switching ASIC)



Barefoot Tofino 2-powered 12.8 Tbps 32xQSFP56-DD System

https://www.barefootnetworks.com/press-releases/barefoot-networks-unveils-tofino-2-the-next-generation-of-the-worlds-first-fully-p4-programmable-network-switch-asics/

This week on

Advanced Topics in Communication Networks

P4 hardware target

P4-based applications

What cool things can we do with it?

Data plane programmability

for

Performance

Monitoring

Applications offloading

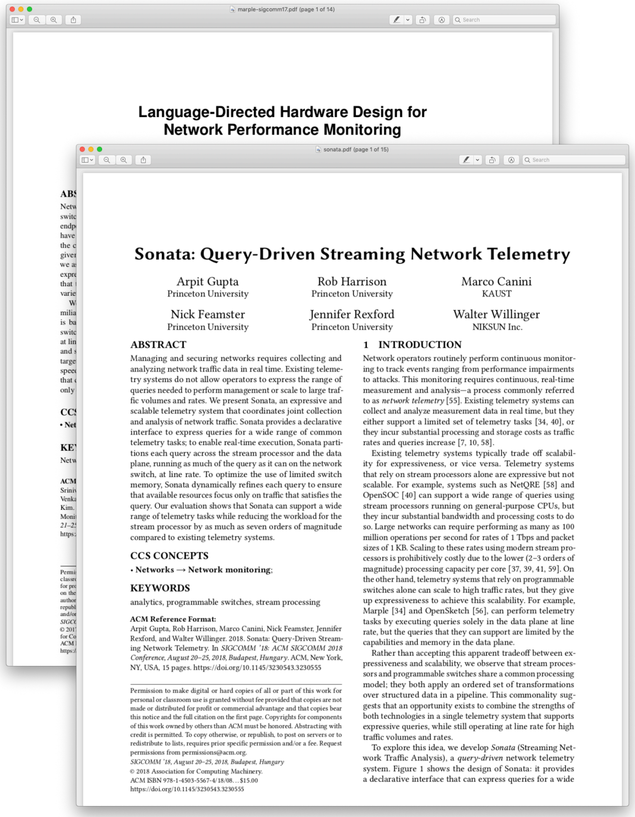
Platforms

Correctness

Management

for

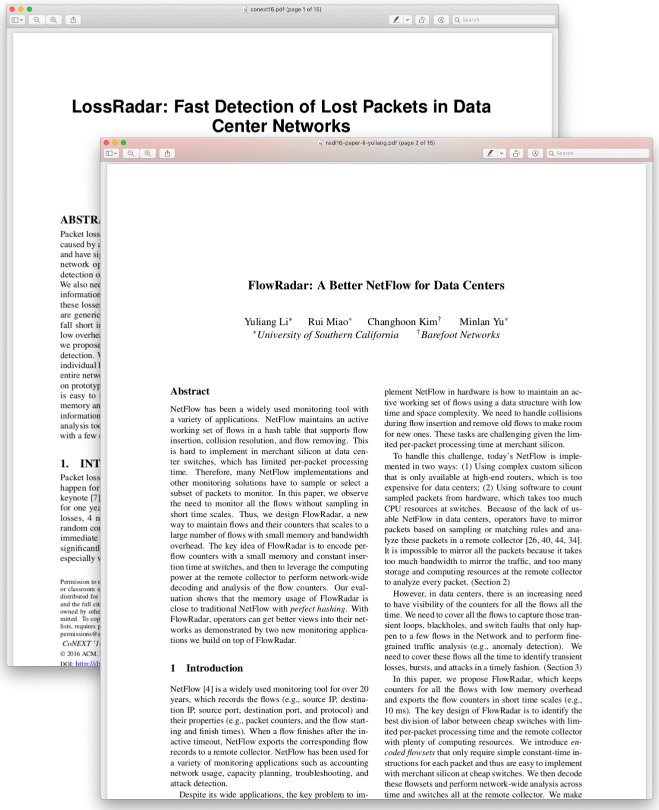
Data plane programmability



Language-Directed Hardware Design for Network Performance Monitoring

Sonata: Query-Driven Streaming Network Telemetry

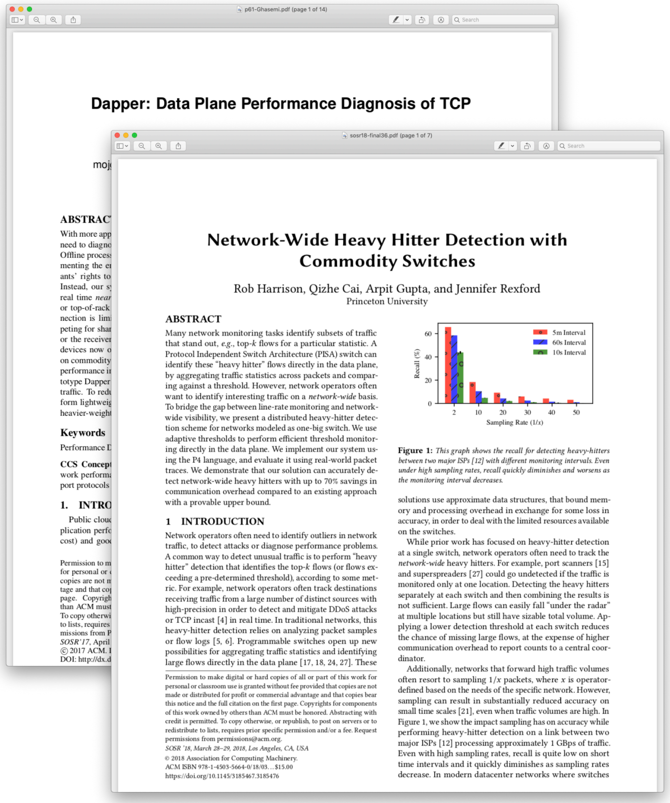
Arpit Gupta, Princeton University; Rob Harrison, Princeton University; Marco Canini, KAUST; Nick Feamster, Princeton University; Jennifer Rexford, Princeton University; Walter Willinger, NIKSUN Inc.



LossRadar: Fast Detection of Lost Packets in Data Center Networks

FlowRadar: A Better NetFlow for Data Centers

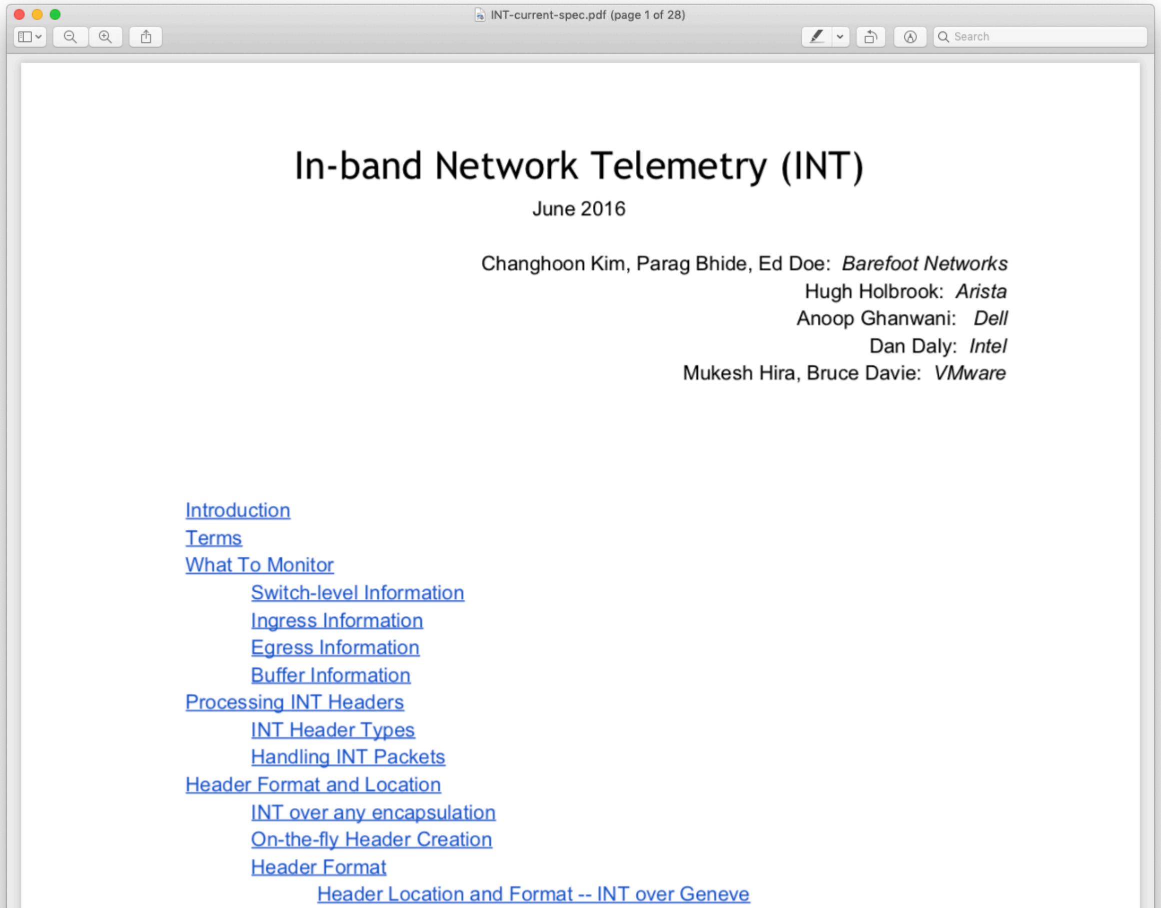
Yuliang Li, Rui Miao, Changhoon Kim, Minlan Yu
University of Southern California, Barefoot Networks



Dapper: Data Plane Performance Diagnosis of TCP

Network-Wide Heavy Hitter Detection with Commodity Switches

Rob Harrison, Qizhe Cai, Gregory Vorsanger and Jennifer Rexford
Princeton University



INT-current-spec.pdf (page 1 of 28)

In-band Network Telemetry (INT)

June 2016

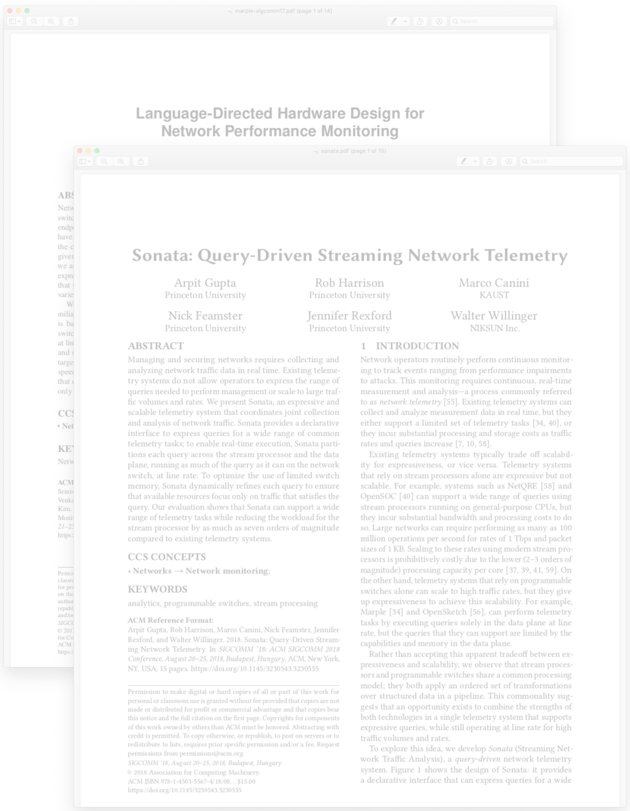
Changhoon Kim, Parag Bhide, Ed Doe: *Barefoot Networks*
Hugh Holbrook: *Arista*
Anoop Ghanwani: *Dell*
Dan Daly: *Intel*
Mukesh Hira, Bruce Davie: *VMware*

SketchLearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference

Elastic Sketch: Adaptive and Fast Network-wide Measurements

One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon

Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, Vladimir Braverman
Johns Hopkins University, Carnegie Mellon University



Language-Directed Hardware Design for Network Performance Monitoring

sonata.pdf (page 1 of 15)

Sonata: Query-Driven Streaming Network Telemetry

Arpit Gupta, Princeton University; Rob Harrison, Princeton University; Marco Canini, KAUST; Nick Feamster, Princeton University; Jennifer Rexford, Princeton University; Walter Willinger, NIKSUN Inc.

ABSTRACT

Managing and securing networks requires collecting and analyzing network traffic data in real time. Existing telemetry systems do not allow operators to express the range of queries needed to perform management or scale to large traffic volumes and rates. We present Sonata, an expressive and scalable telemetry system that coordinates joint collection and analysis of network traffic. Sonata provides a declarative interface to express queries for a wide range of common

1 INTRODUCTION

Network operators routinely perform continuous monitoring to track events ranging from performance impairments to attacks. This monitoring requires continuous, real-time measurement and analysis—a process commonly referred to as *network telemetry* [55]. Existing telemetry systems can collect and analyze measurement data in real time, but they either support a limited set of telemetry tasks [34, 40], or they incur substantial processing and storage costs as traffic

LossRadar: Fast Detection of Lost Packets in Data Center Networks

FlowRadar: A Better NetFlow for Data Centers

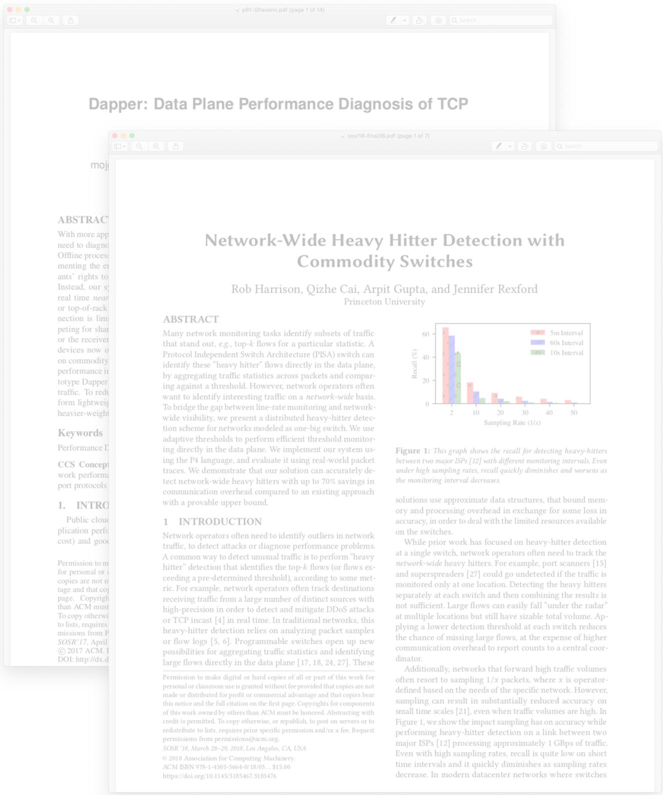
Yuliang Li*, Rui Miao*, Changhoon Kim†, Minlan Yu*

*University of Southern California †Barefoot Networks

Abstract

NetFlow has been a widely used monitoring tool with a variety of applications. NetFlow maintains an active working set of flows in a hash table that supports flow insertion, collision resolution, and flow removing.

plement NetFlow in hardware is how to maintain an active working set of flows using a data structure with low time and space complexity. We need to handle collisions during flow insertion and remove old flows to make room for new ones. These tasks are challenging given the limited per-packet processing time at merchant silicon.



Dapper: Data Plane Performance Diagnosis of TCP

sosr18-final36.pdf (page 1 of 7)

Network-Wide Heavy Hitter Detection with Commodity Switches

Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford

Princeton University

ABSTRACT

Many network monitoring tasks identify subsets of traffic that stand out, *e.g.*, top-*k* flows for a particular statistic. A Protocol Independent Switch Architecture (PISA) switch can identify these "heavy hitter" flows directly in the data plane, by aggregating traffic statistics across packets and comparing against a threshold. However, network operators often want to identify interesting traffic on a *network-wide* basis. To bridge the gap between line-rate monitoring and network-

SketchLearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference

Elastic Sketch: Adaptive and Fast Network-wide Measurements

sigcomm16_univmon.pdf (page 1 of 14)

One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon

Zaoxing Liu†, Antonis Manousis*, Gregory Vorsanger†, Vyas Sekar*, Vladimir Braverman†

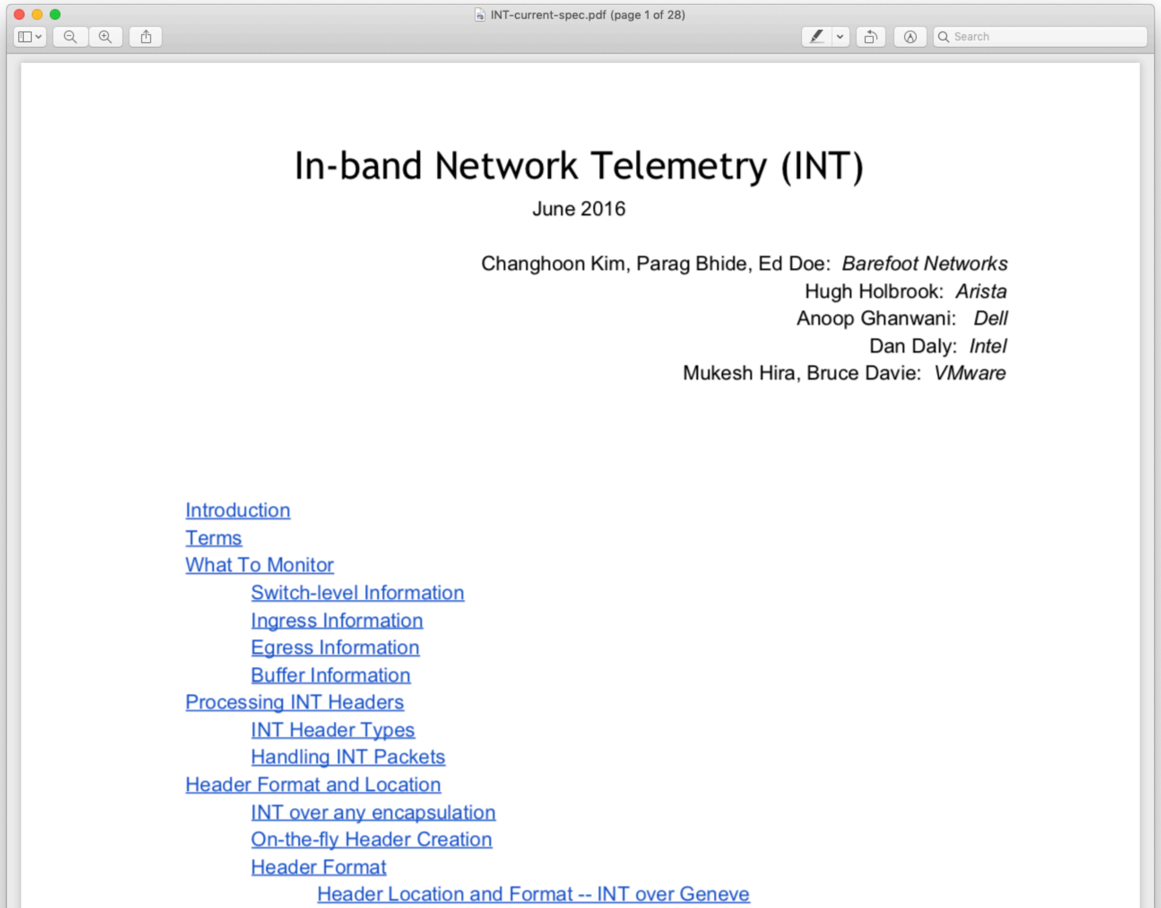
† Johns Hopkins University * Carnegie Mellon University

ABSTRACT

Network management requires accurate estimates of metrics for many applications including traffic engineering (e.g., heavy hitters), anomaly detection (e.g., entropy of source addresses), and security (e.g., DDoS detection). Obtaining accurate estimates given router CPU and memory constraints is a challenging problem. Existing approaches fall

1 Introduction

Network management is multi-faceted and encompasses a range of tasks including traffic engineering [11, 32], attack and anomaly detection [49], and forensic analysis [46]. Each such management task requires accurate and timely statistics on different application-level metrics of interest; e.g., the flow size distribution [37], heavy hitters [10], entropy measures [38, 50], or detecting changes in traffic patterns [44].

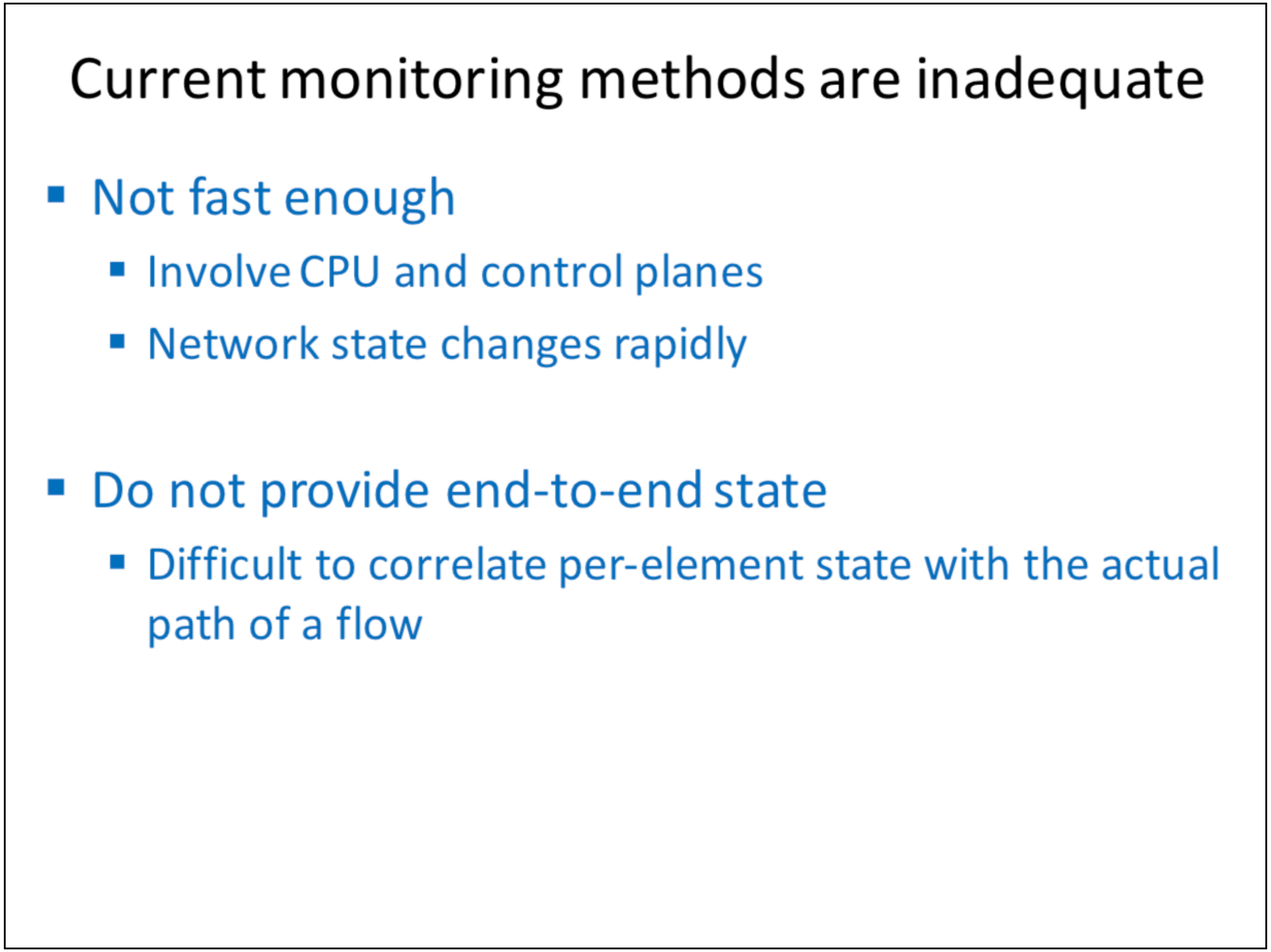


INT-current-spec.pdf (page 1 of 28)

In-band Network Telemetry (INT)

June 2016

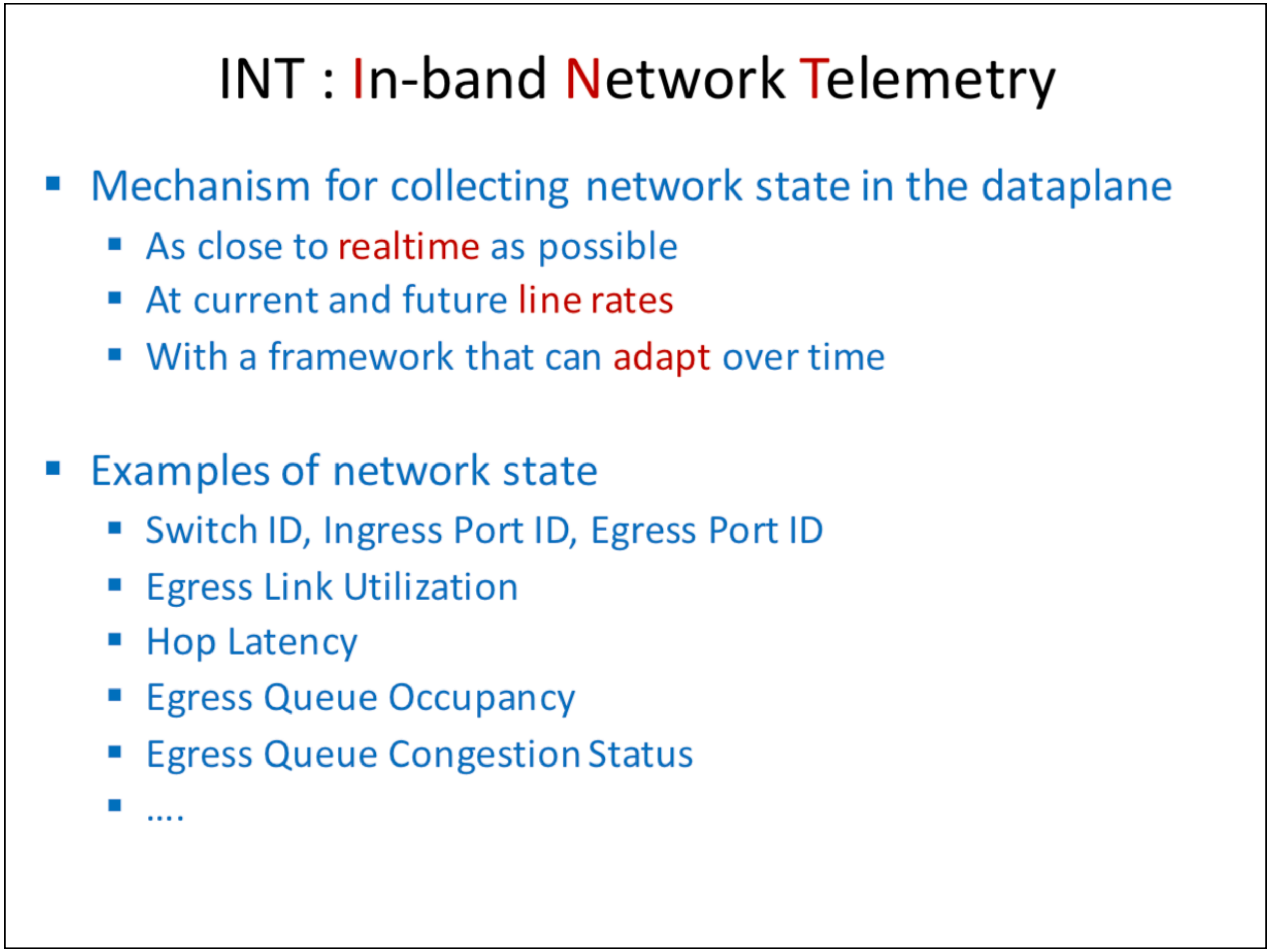
Changhoon Kim, Parag Bhide, Ed Doe: *Barefoot Networks*
Hugh Holbrook: *Arista*
Anoop Ghanwani: *Dell*
Dan Daly: *Intel*
Mukesh Hira, Bruce Davie: *VMware*



Current monitoring methods are inadequate

- Not fast enough
  - Involve CPU and control planes
  - Network state changes rapidly
- Do not provide end-to-end state
  - Difficult to correlate per-element state with the actual path of a flow

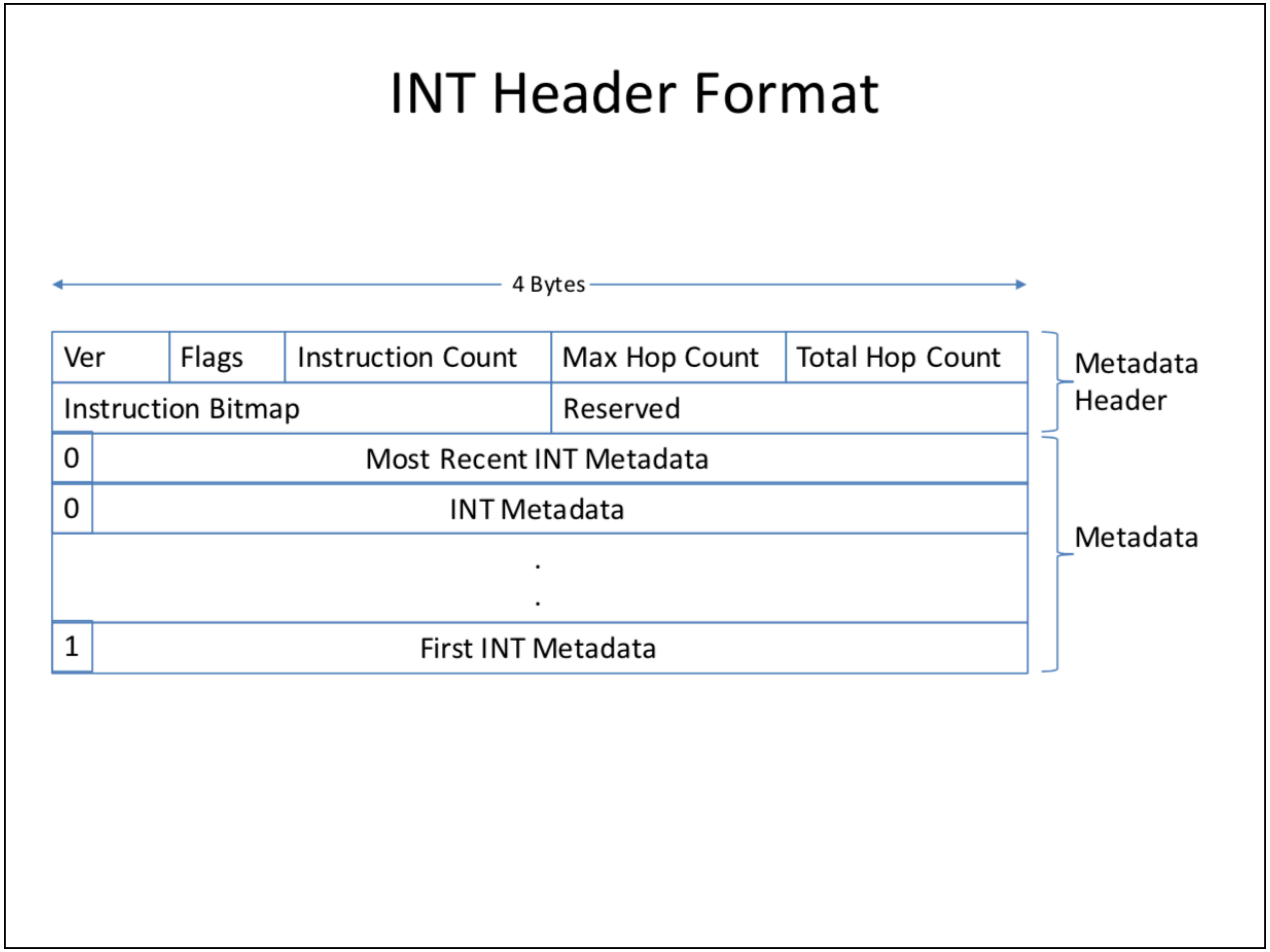
Source: In-band Network Telemetry, Mukesh Hira and Naga Katta, 2015



INT : In-band Network Telemetry

- Mechanism for collecting network state in the dataplane
  - As close to realtime as possible
  - At current and future line rates
  - With a framework that can adapt over time
- Examples of network state
  - Switch ID, Ingress Port ID, Egress Port ID
  - Egress Link Utilization
  - Hop Latency
  - Egress Queue Occupancy
  - Egress Queue Congestion Status
  - ….

Source: In-band Network Telemetry, Mukesh Hira and Naga Katta, 2015

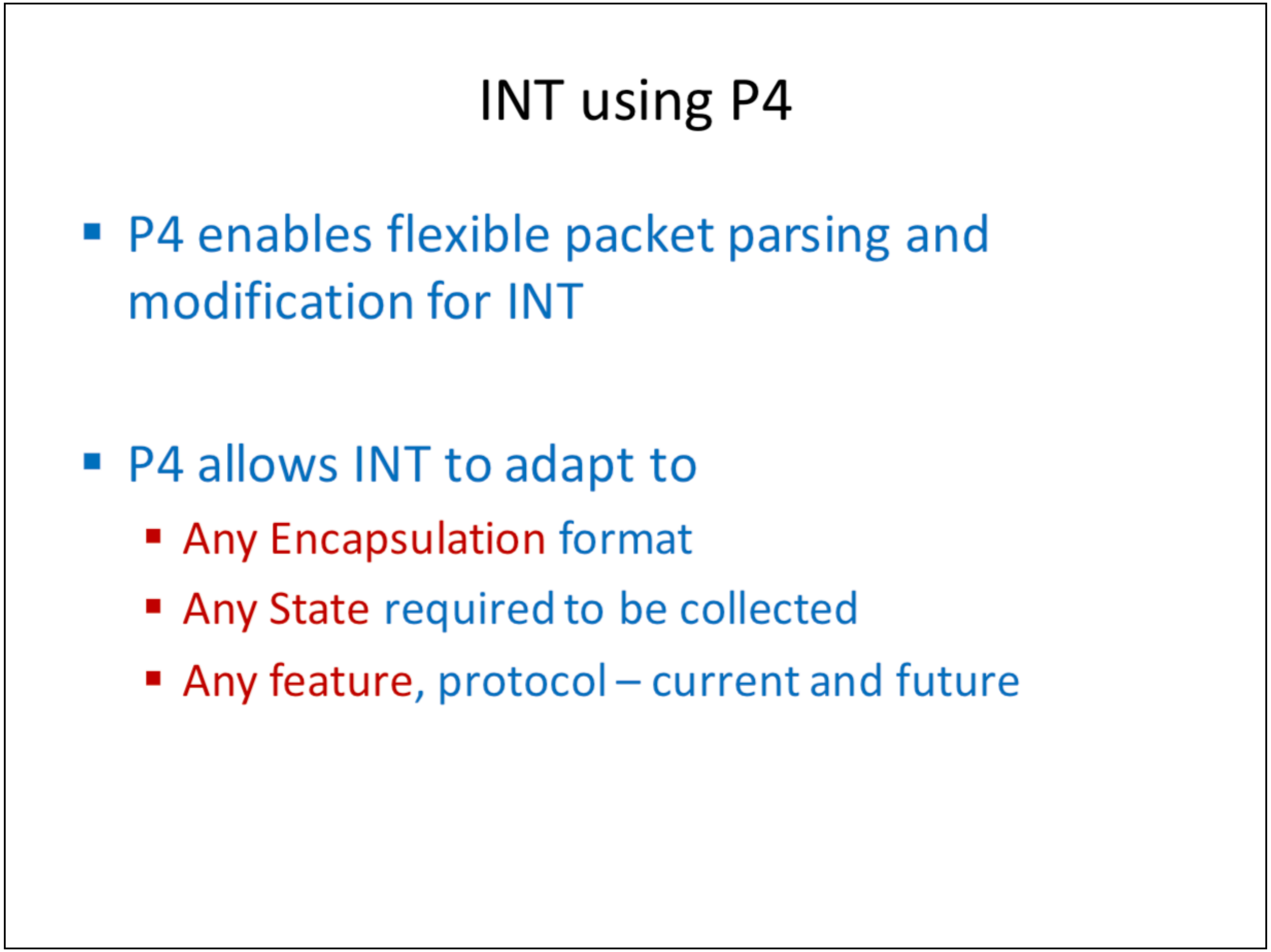


INT Header Format

4 Bytes

Ver	Flags	Instruction Count	Max Hop Count	Total Hop Count	Metadata Header
Instruction Bitmap			Reserved		
0	Most Recent INT Metadata				Metadata
0	INT Metadata				
	. .				
1	First INT Metadata				

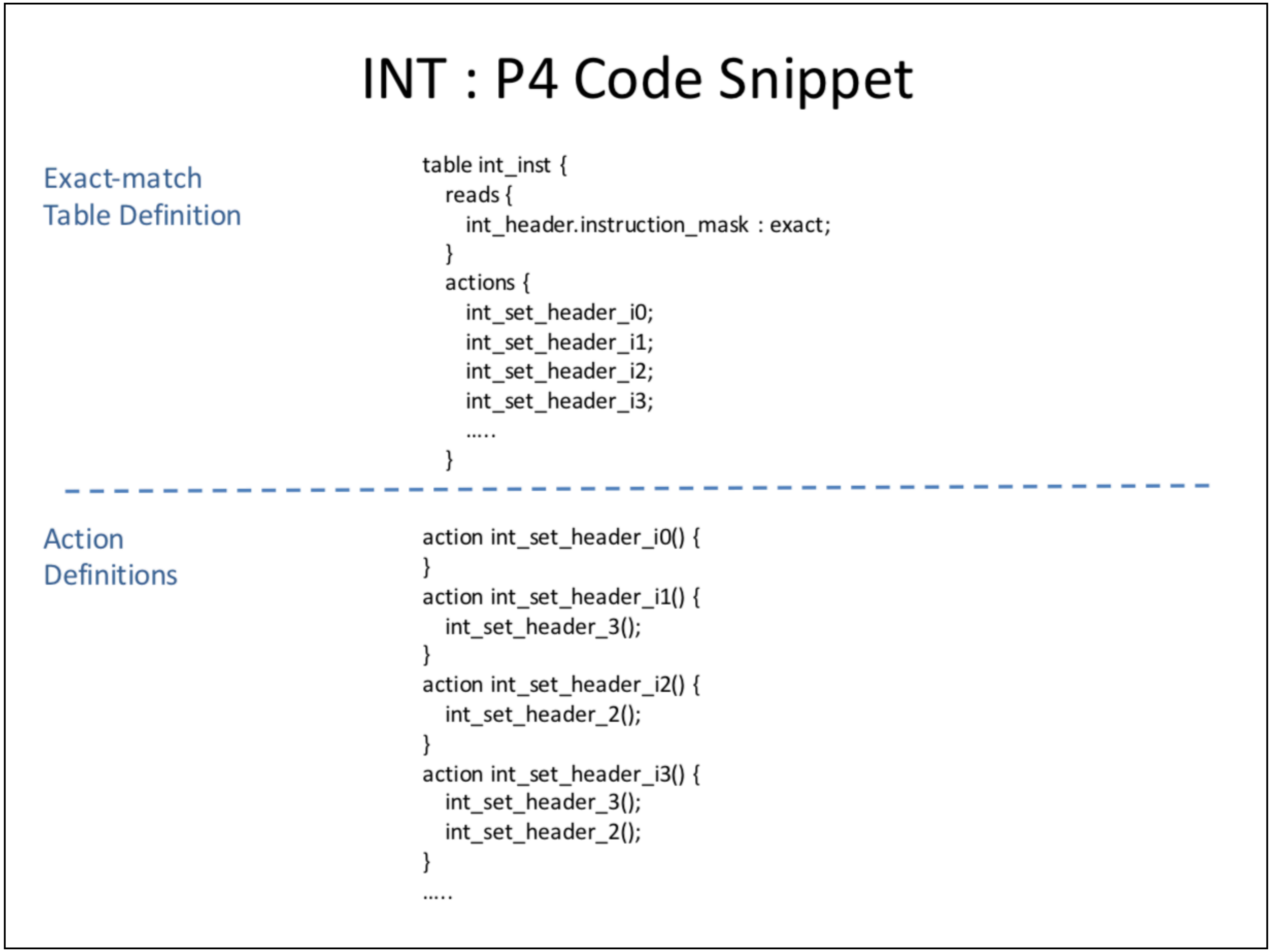
Source: In-band Network Telemetry, Mukesh Hira and Naga Katta, 2015



INT using P4

- P4 enables flexible packet parsing and modification for INT
- P4 allows INT to adapt to
  - Any Encapsulation format
  - Any State required to be collected
  - Any feature, protocol – current and future

Source: In-band Network Telemetry, Mukesh Hira and Naga Katta, 2015



INT : P4 Code Snippet

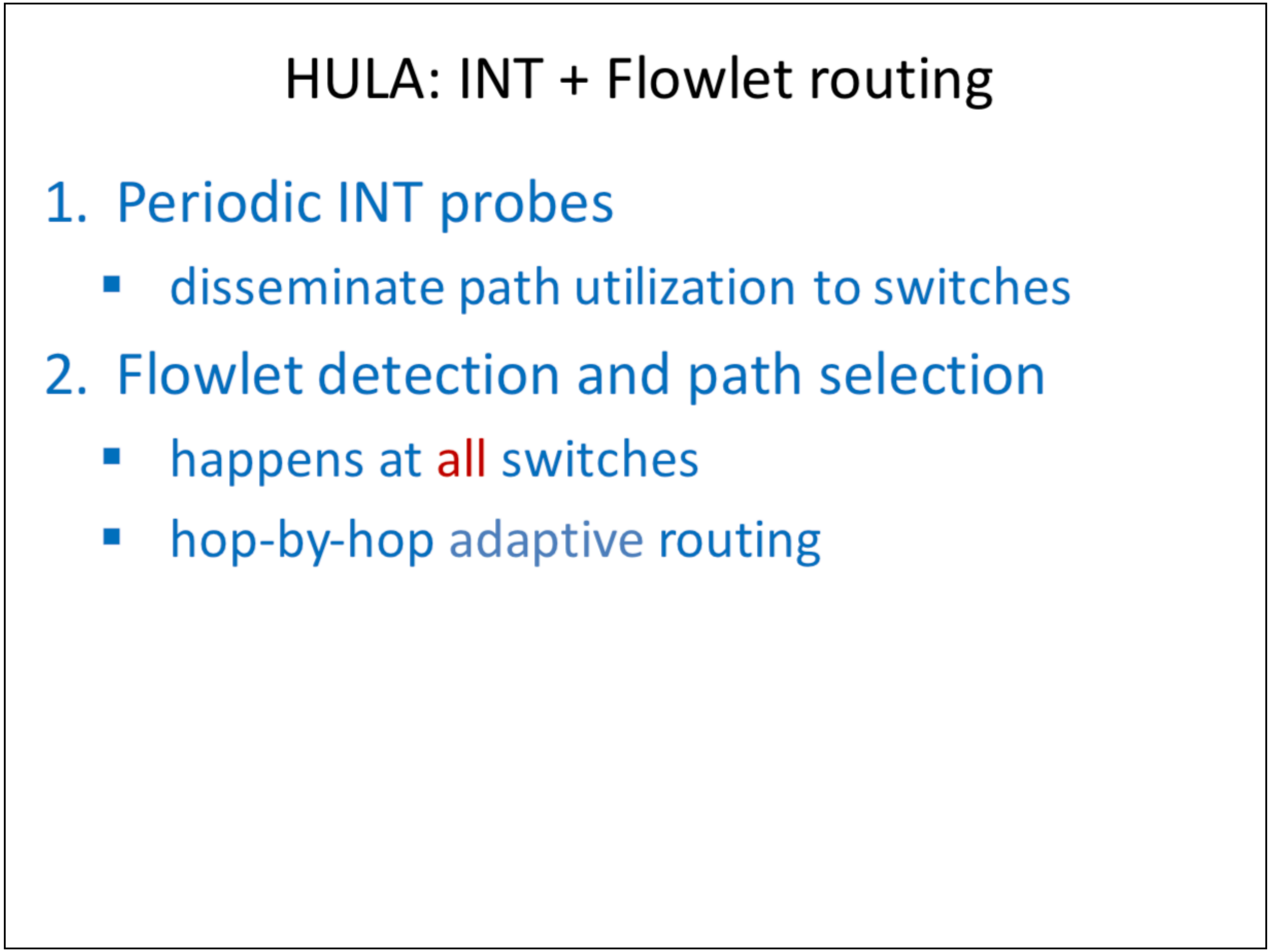
Exact-match
Table Definition

```
table int_inst {
  reads {
    int_header.instruction_mask : exact;
  }
  actions {
    int_set_header_i0;
    int_set_header_i1;
    int_set_header_i2;
    int_set_header_i3;
    .....
  }
```

Action
Definitions

```
action int_set_header_i0() {
}
action int_set_header_i1() {
  int_set_header_3();
}
action int_set_header_i2() {
  int_set_header_2();
}
action int_set_header_i3() {
  int_set_header_3();
  int_set_header_2();
}
.....
```

Source: In-band Network Telemetry, Mukesh Hira and Naga Katta, 2015

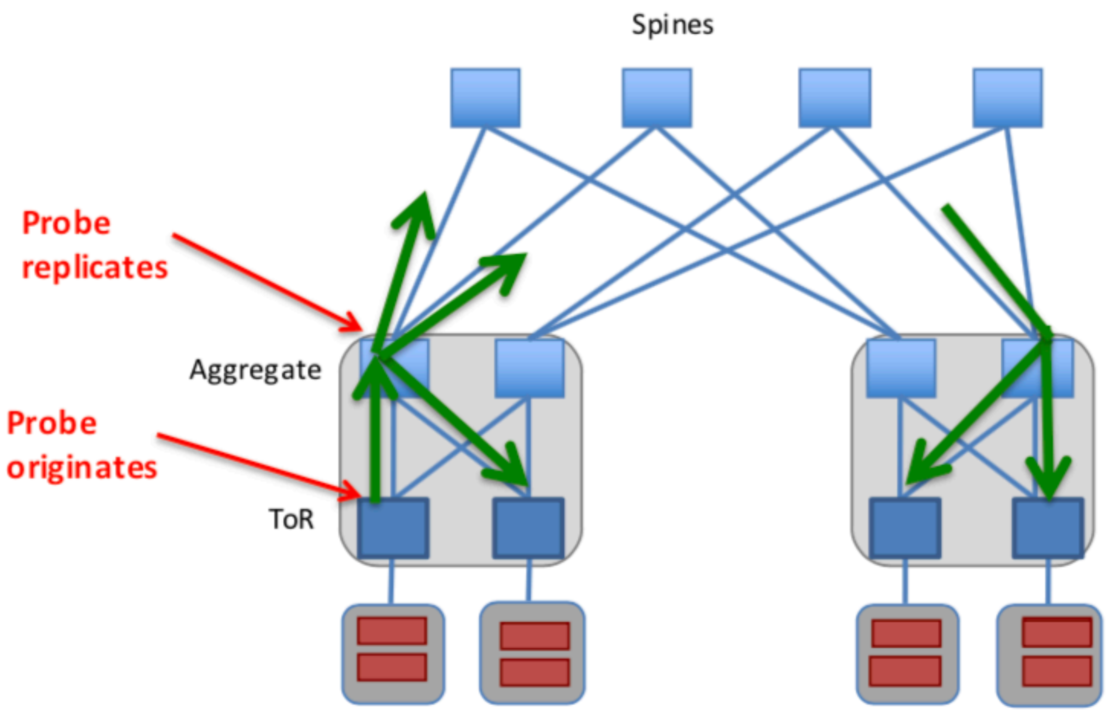


HULA: INT + Flowlet routing

1. Periodic INT probes
   - disseminate path utilization to switches
2. Flowlet detection and path selection
   - happens at all switches
   - hop-by-hop adaptive routing

Source: In-band Network Telemetry, Mukesh Hira and Naga Katta, 2015

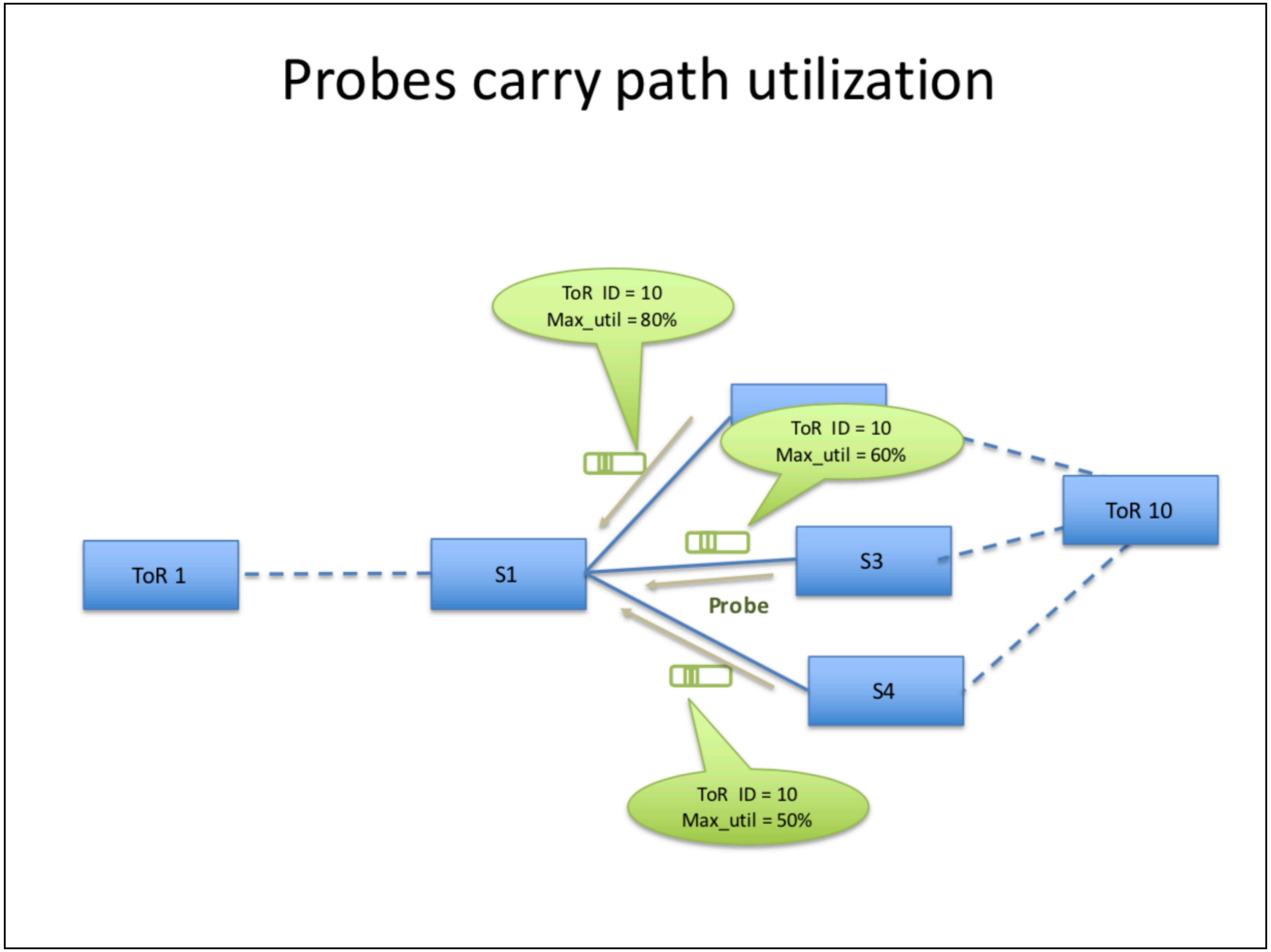
INT probes traverse multiple paths



Probe replicates

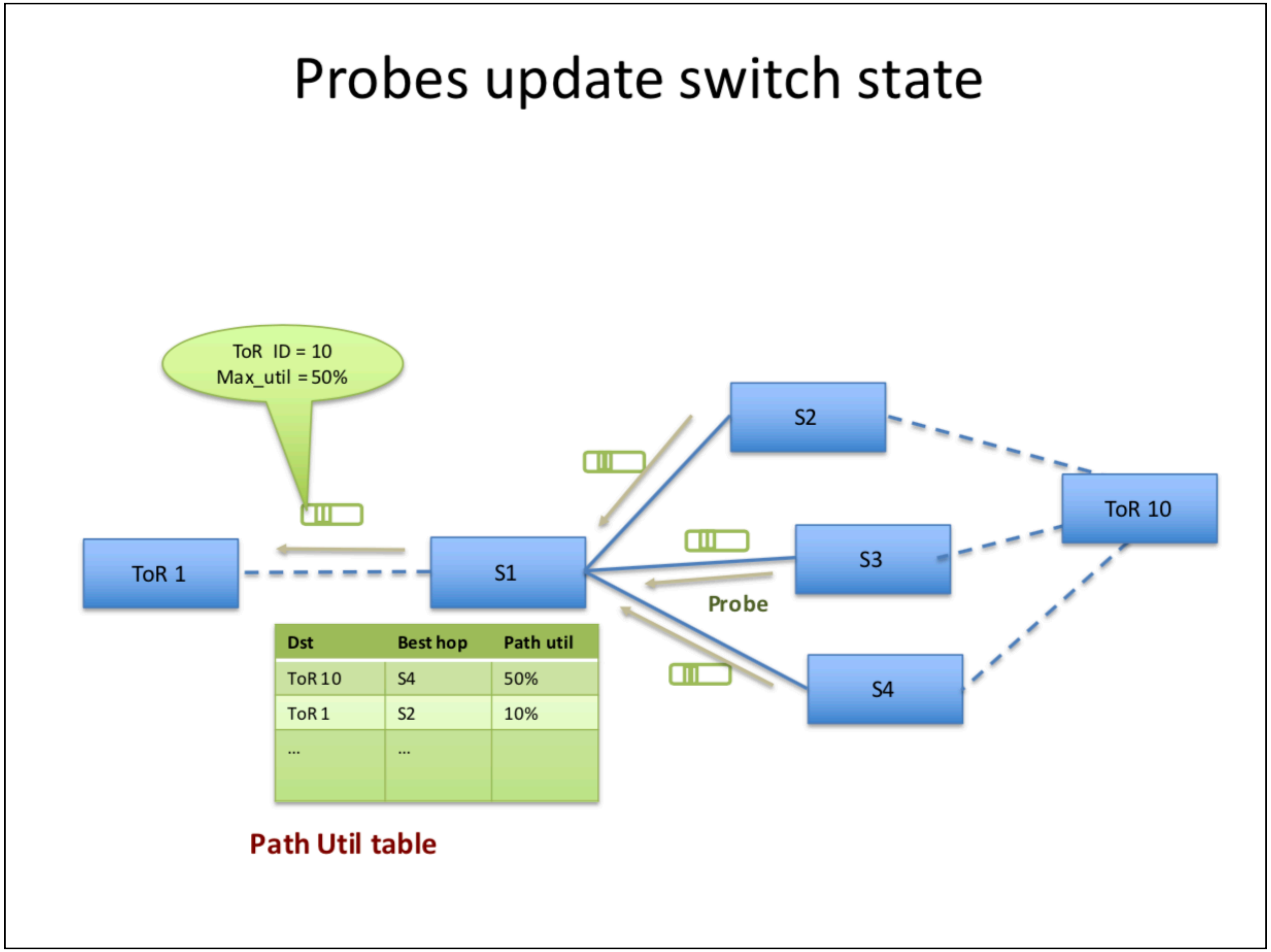
Probe originates

Source: In-band Network Telemetry, Mukesh Hira and Naga Katta, 2015



Probes carry path utilization

Source: In-band Network Telemetry, Mukesh Hira and Naga Katta, 2015



Probes update switch state

ToR ID = 10
Max_util = 50%

ToR 1

S1

S2

S3

S4

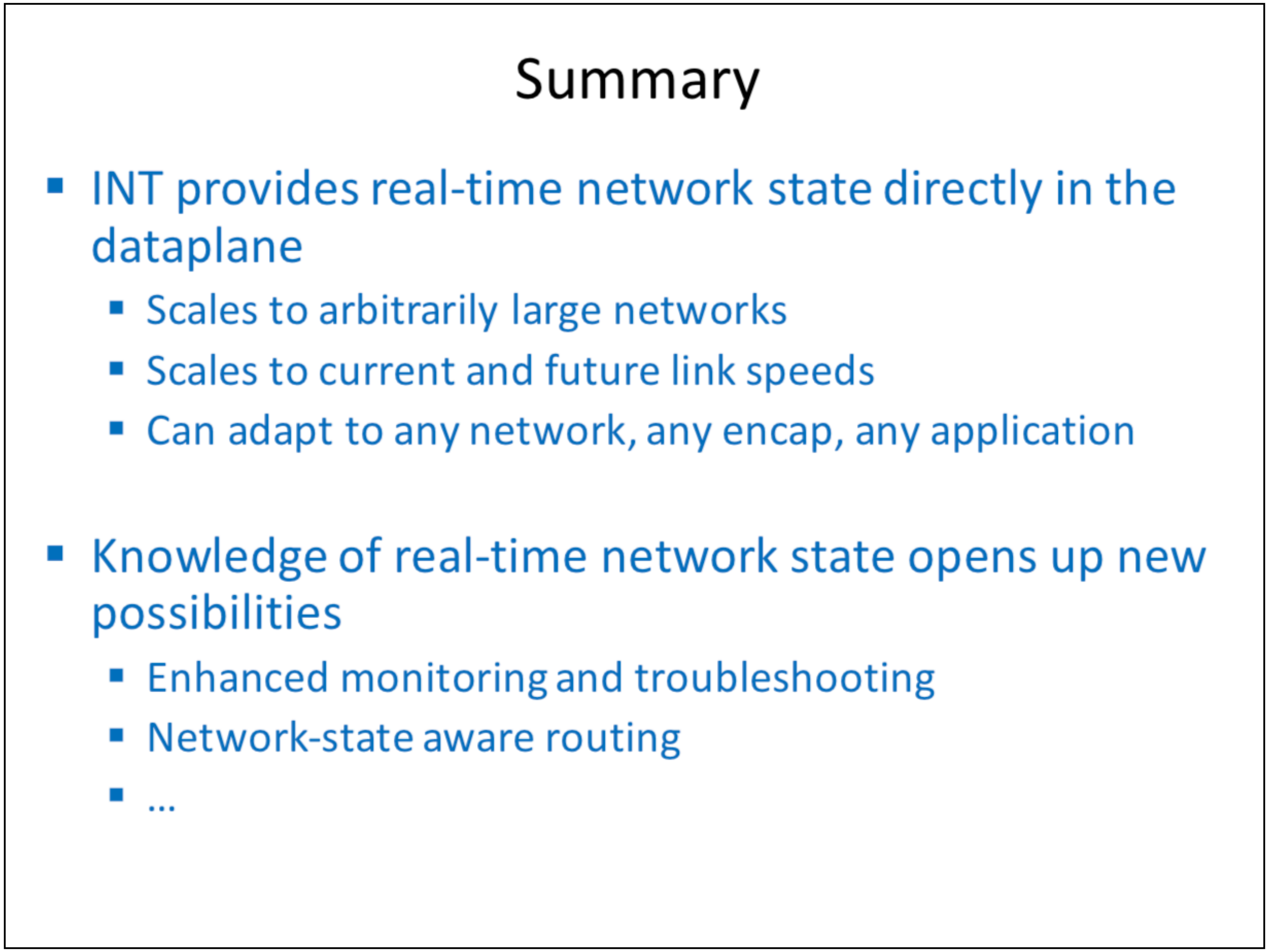
ToR 10

Probe

Dst	Best hop	Path util
ToR 10	S4	50%
ToR 1	S2	10%
…	…	

Path Util table

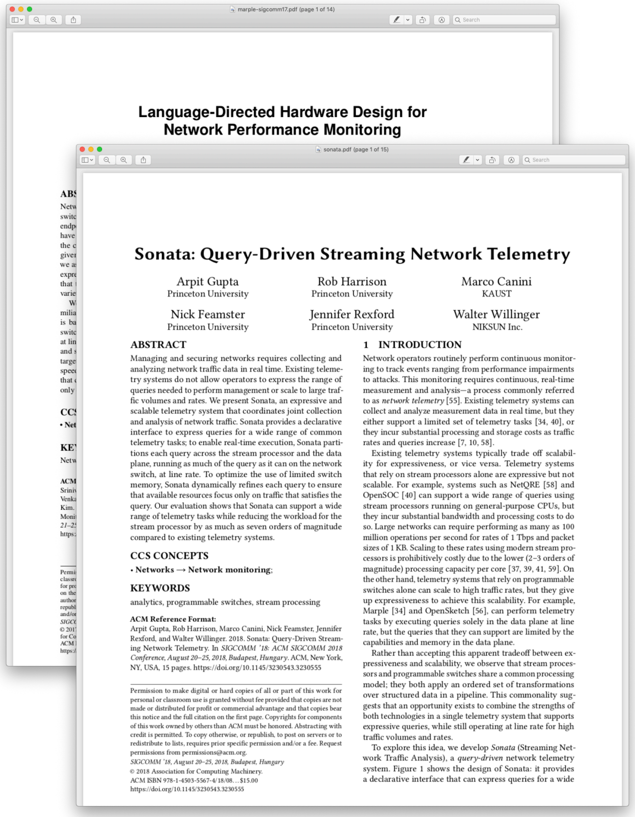
Source: In-band Network Telemetry, Mukesh Hira and Naga Katta, 2015



Summary

- INT provides real-time network state directly in the dataplane
  - Scales to arbitrarily large networks
  - Scales to current and future link speeds
  - Can adapt to any network, any encap, any application
- Knowledge of real-time network state opens up new possibilities
  - Enhanced monitoring and troubleshooting
  - Network-state aware routing
  - …

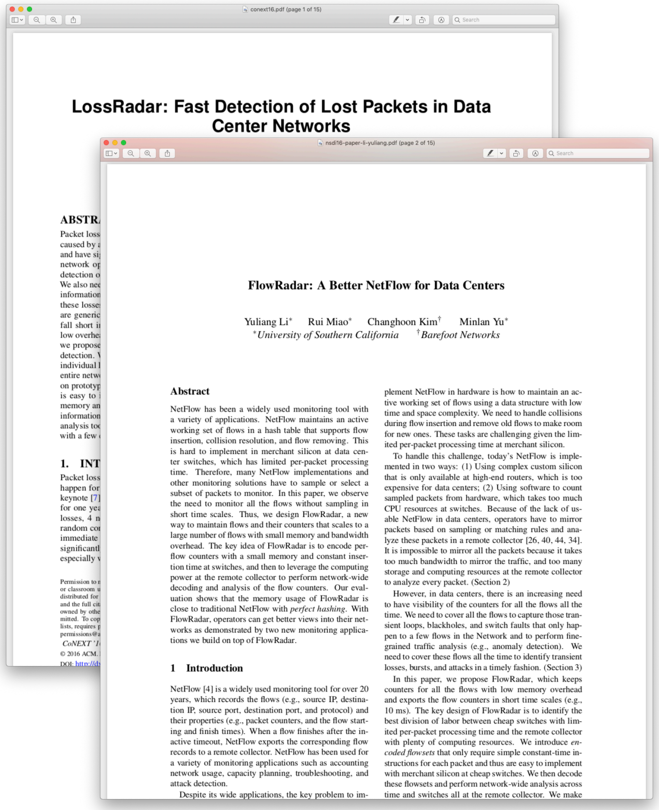
Source: In-band Network Telemetry, Mukesh Hira and Naga Katta, 2015



Language-Directed Hardware Design for Network Performance Monitoring

Sonata: Query-Driven Streaming Network Telemetry

Arpit Gupta, Princeton University
Rob Harrison, Princeton University
Marco Canini, KAUST
Nick Feamster, Princeton University
Jennifer Rexford, Princeton University
Walter Willinger, NIKSUN Inc.

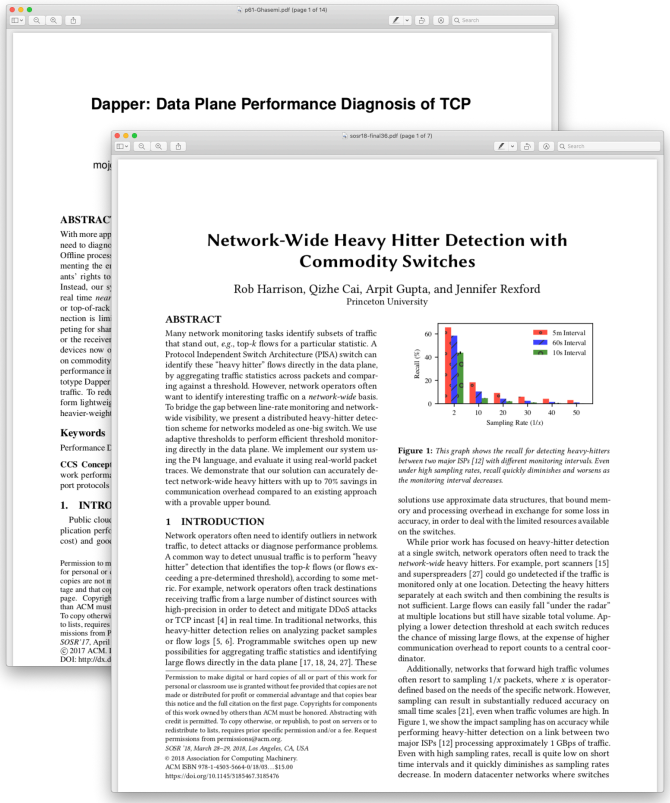


LossRadar: Fast Detection of Lost Packets in Data Center Networks

FlowRadar: A Better NetFlow for Data Centers

Yuliang Li[*] Rui Miao[*] Changhoon Kim[†] Minlan Yu[*]
[*] University of Southern California [†] Barefoot Networks

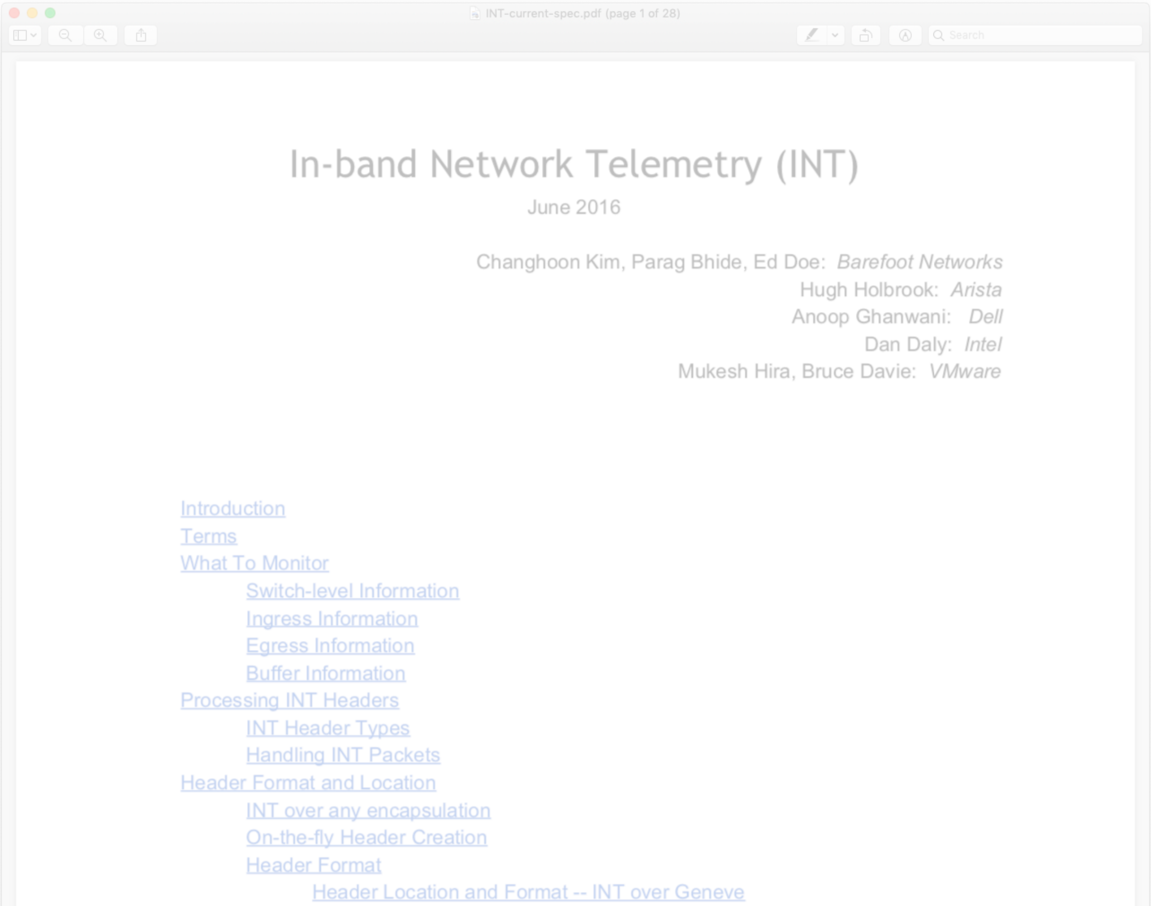
conext16.pdf (page 1 of 15)
nsdi16-paper-li-yuliang.pdf (page 2 of 15)



Dapper: Data Plane Performance Diagnosis of TCP

Network-Wide Heavy Hitter Detection with Commodity Switches

Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford
Princeton University



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

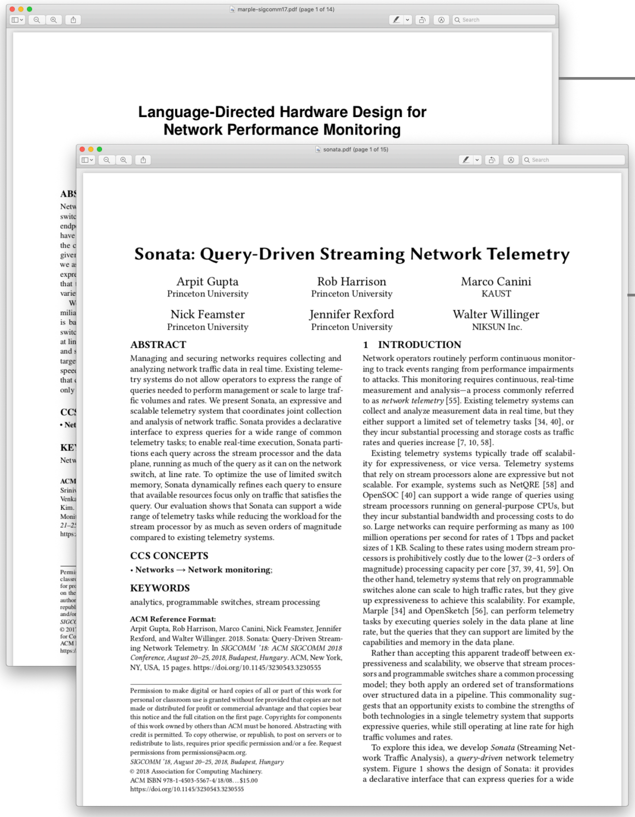
SketchLearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference

Elastic Sketch: Adaptive and Fast Network-wide Measurements

One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon

Zaoxing Liu[†], Antonis Manousis[*], Gregory Vorsanger[†], Vyas Sekar[*], Vladimir Braverman[†]
[†] Johns Hopkins University [*] Carnegie Mellon University

p576-huang.pdf (page 1 of 15)
p561-yang.pdf (page 1 of 15)
sigcomm16_univmon.pdf (page 1 of 14)


marple-sigcomm17.pdf (page 1 of 14)
Search
sonata.pdf (page 1 of 15)
Search

Language-Directed Hardware Design for
Network Performance Monitoring

Sonata: Query-Driven Streaming Network Telemetry

Arpit Gupta, Princeton University
Rob Harrison, Princeton University
Marco Canini, KAUST
Nick Feamster, Princeton University
Jennifer Rexford, Princeton University
Walter Willinger, NIKSUN Inc.

ABSTRACT

Managing and securing networks requires collecting and analyzing network traffic data in real time. Existing telemetry systems do not allow operators to express the range of queries needed to perform management or scale to large traffic volumes and rates. We present Sonata, an expressive and scalable telemetry system that coordinates joint collection and analysis of network traffic. Sonata provides a declarative interface to express queries for a wide range of common telemetry tasks; to enable real-time execution, Sonata partitions each query across the stream processor and the data plane, running as much of the query as it can on the network switch, at line rate. To optimize the use of limited switch memory, Sonata dynamically refines each query to ensure that available resources focus only on traffic that satisfies the query. Our evaluation shows that Sonata can support a wide range of telemetry tasks while reducing the workload for the stream processor by as much as seven orders of magnitude compared to existing telemetry systems.

CCS CONCEPTS

• **Networks → Network monitoring**;

KEYWORDS

analytics, programmable switches, stream processing

ACM Reference Format:
Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-Driven Streaming Network Telemetry. In *SIGCOMM '18: ACM SIGCOMM 2018 Conference, August 20–25, 2018, Budapest, Hungary*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3230543.3230555

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCOMM '18, August 20–25, 2018, Budapest, Hungary
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5567-4/18/08…$15.00
https://doi.org/10.1145/3230543.3230555

1 INTRODUCTION

Network operators routinely perform continuous monitoring to track events ranging from performance impairments to attacks. This monitoring requires continuous, real-time measurement and analysis—a process commonly referred to as *network telemetry* [55]. Existing telemetry systems can collect and analyze measurement data in real time, but they either support a limited set of telemetry tasks [34, 40], or they incur substantial processing and storage costs as traffic rates and queries increase [7, 10, 58].

Existing telemetry systems typically trade off scalability for expressiveness, or vice versa. Telemetry systems that rely on stream processors alone are expressive but not scalable. For example, systems such as NetQRE [58] and OpenSOC [40] can support a wide range of queries using stream processors running on general-purpose CPUs, but they incur substantial bandwidth and processing costs to do so. Large networks can require performing as many as 100 million operations per second for rates of 1 Tbps and packet sizes of 1 KB. Scaling to these rates using modern stream processors is prohibitively costly due to the lower (2–3 orders of magnitude) processing capacity per core [37, 39, 41, 59]. On the other hand, telemetry systems that rely on programmable switches alone can scale to high traffic rates, but they give up expressiveness to achieve this scalability. For example, Marple [34] and OpenSketch [56], can perform telemetry tasks by executing queries solely in the data plane at line rate, but the queries that they can support are limited by the capabilities and memory in the data plane.

Rather than accepting this apparent tradeoff between expressiveness and scalability, we observe that stream processors and programmable switches share a common processing model; they both apply an ordered set of transformations over structured data in a pipeline. This commonality suggests that an opportunity exists to combine the strengths of both technologies in a single telemetry system that supports expressive queries, while still operating at line rate for high traffic volumes and rates.

To explore this idea, we develop *Sonata* (Streaming Network Traffic Analysis), a *query-driven* network telemetry system. Figure 1 shows the design of Sonata: it provides a declarative interface that can express queries for a wide

MARPLE [SIGCOMM'17]

SONATA [SIGCOMM'18]

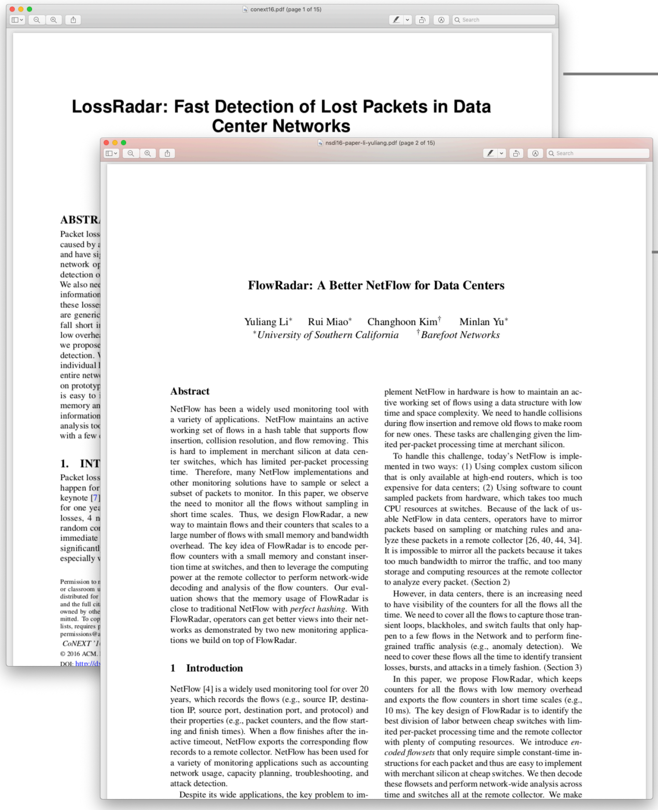
Both papers enable operators to express monitoring queries

```
result = filter(pktstream, qid == Q and switch == S
                      and t_out - t_in > 1ms)
```

returns a stream of packets experiencing high queuing latencies

A compiler then compiles these queries to: switch programs +
control code

The two papers differ among others in the types of queries they support


conext16.pdf (page 1 of 15)
Search
nsdi16-paper-li-yuliang.pdf (page 2 of 15)
Search

LossRadar: Fast Detection of Lost Packets in Data Center Networks

FlowRadar: A Better NetFlow for Data Centers

Yuliang Li* Rui Miao* Changhoon Kim† Minlan Yu*
*University of Southern California †Barefoot Networks

Abstract

NetFlow has been a widely used monitoring tool with a variety of applications. NetFlow maintains an active working set of flows in a hash table that supports flow insertion, collision resolution, and flow removing. This is hard to implement in merchant silicon at data center switches, which has limited per-packet processing time. Therefore, many NetFlow implementations and other monitoring solutions have to sample or select a subset of packets to monitor. In this paper, we observe the need to monitor all the flows without sampling in short time scales. Thus, we design FlowRadar, a new way to maintain flows and their counters that scales to a large number of flows with small memory and bandwidth overhead. The key idea of FlowRadar is to encode per-flow counters with a small memory and constant insertion time at switches, and then to leverage the computing power at the remote collector to perform network-wide decoding and analysis of the flow counters. Our evaluation shows that the memory usage of FlowRadar is close to traditional NetFlow with *perfect hashing*. With FlowRadar, operators can get better views into their networks as demonstrated by two new monitoring applications we build on top of FlowRadar.

1 Introduction

NetFlow [4] is a widely used monitoring tool for over 20 years, which records the flows (e.g., source IP, destination IP, source port, destination port, and protocol) and their properties (e.g., packet counters, and the flow starting and finish times). When a flow finishes after the inactive timeout, NetFlow exports the corresponding flow records to a remote collector. NetFlow has been used for a variety of monitoring applications such as accounting network usage, capacity planning, troubleshooting, and attack detection.

Despite its wide applications, the key problem to implement NetFlow in hardware is how to maintain an active working set of flows using a data structure with low time and space complexity. We need to handle collisions during flow insertion and remove old flows to make room for new ones. These tasks are challenging given the limited per-packet processing time at merchant silicon.

To handle this challenge, today's NetFlow is implemented in two ways: (1) Using complex custom silicon that is only available at high-end routers, which is too expensive for data centers; (2) Using software to count sampled packets from hardware, which takes too much CPU resources at switches. Because of the lack of usable NetFlow in data centers, operators have to mirror packets based on sampling or matching rules and analyze these packets in a remote collector [26, 40, 44, 34]. It is impossible to mirror all the packets because it takes too much bandwidth to mirror the traffic, and too many storage and computing resources at the remote collector to analyze every packet. (Section 2)

However, in data centers, there is an increasing need to have visibility of the counters for all the flows all the time. We need to cover all the flows to capture those transient loops, blackholes, and switch faults that only happen to a few flows in the Network and to perform fine-grained traffic analysis (e.g., anomaly detection). We need to cover these flows all the time to identify transient losses, bursts, and attacks in a timely fashion. (Section 3)

In this paper, we propose FlowRadar, which keeps counters for all the flows with low memory overhead and exports the flow counters in short time scales (e.g., 10 ms). The key design of FlowRadar is to identify the best division of labor between cheap switches with limited per-packet processing time and the remote collector with plenty of computing resources. We introduce *encoded flowsets* that only require simple constant-time instructions for each packet and thus are easy to implement with merchant silicon at cheap switches. We then decode these flowsets and perform network-wide analysis across time and switches all at the remote collector. We make

LossRadar [CoNEXT'16]

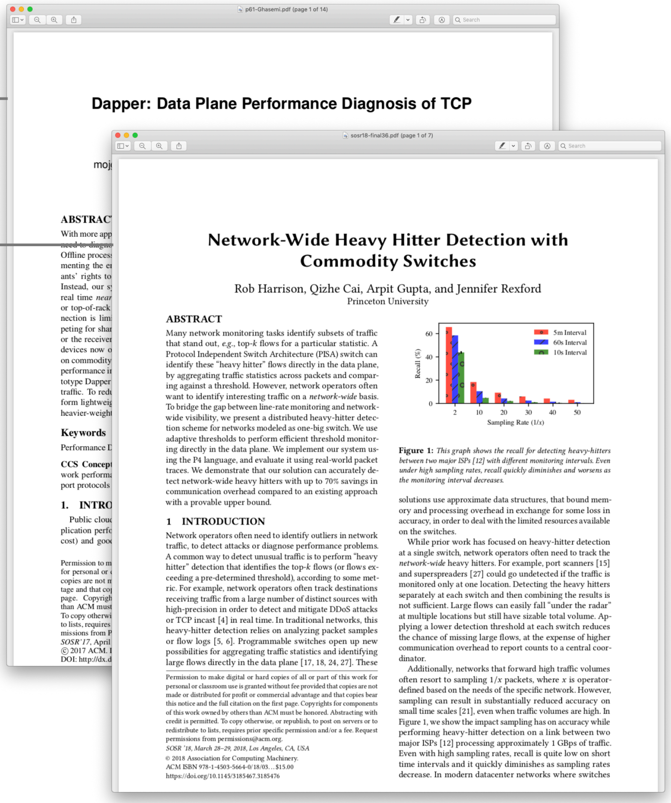
FlowRadar [NSDI'16]

Develop techniques and tools to monitor *all flows* by

- relying on in-switch data structures (Bloom Filters) and
- decoding them at the controller-level

DAPPER [SOSR'17]

Network-Wide HH [SOSR'18]



Dapper: Data Plane Performance Diagnosis of TCP

moj

ABSTRAC
With more app
Offline process
menting the e
ants' rights to
Instead, our sy
real time *near*
or top-of-rack
nection is limi
peting for shar
or the receiver
devices now o
on commodity
performance in
totype Dapper
traffic. To redu
form lightweig
heavier-weight

Keywords
Performance D

CCS Concept
work performa
port protocols

1. INTRO
Public cloud
plication perfo
cost) and good

Permission to m
for personal or c
copies are not m
tage and that cop
page. Copyrigh
than ACM must
To copy otherwi
to lists, requires
missions from P
SOSR'17, April
© 2017 ACM. I
DOI: http://dx.d

Network-Wide Heavy Hitter Detection with Commodity Switches

Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford
Princeton University

ABSTRACT

Many network monitoring tasks identify subsets of traffic that stand out, *e.g.*, top-*k* flows for a particular statistic. A Protocol Independent Switch Architecture (PISA) switch can identify these "heavy hitter" flows directly in the data plane, by aggregating traffic statistics across packets and comparing against a threshold. However, network operators often want to identify interesting traffic on a *network-wide* basis. To bridge the gap between line-rate monitoring and network-wide visibility, we present a distributed heavy-hitter detection scheme for networks modeled as one-big switch. We use adaptive thresholds to perform efficient threshold monitoring directly in the data plane. We implement our system using the P4 language, and evaluate it using real-world packet traces. We demonstrate that our solution can accurately detect network-wide heavy hitters with up to 70% savings in communication overhead compared to an existing approach with a provable upper bound.

1 INTRODUCTION

Network operators often need to identify outliers in network traffic, to detect attacks or diagnose performance problems. A common way to detect unusual traffic is to perform "heavy hitter" detection that identifies the top-*k* flows (or flows exceeding a pre-determined threshold), according to some metric. For example, network operators often track destinations receiving traffic from a large number of distinct sources with high-precision in order to detect and mitigate DDoS attacks or TCP incast [4] in real time. In traditional networks, this heavy-hitter detection relies on analyzing packet samples or flow logs [5, 6]. Programmable switches open up new possibilities for aggregating traffic statistics and identifying large flows directly in the data plane [17, 18, 24, 27]. These solutions use approximate data structures, that bound memory and processing overhead in exchange for some loss in accuracy, in order to deal with the limited resources available on the switches.

While prior work has focused on heavy-hitter detection at a single switch, network operators often need to track the *network-wide* heavy hitters. For example, port scanners [15] and superspreaders [27] could go undetected if the traffic is monitored only at one location. Detecting the heavy hitters separately at each switch and then combining the results is not sufficient. Large flows can easily fall "under the radar" at multiple locations but still have sizable total volume. Applying a lower detection threshold at each switch reduces the chance of missing large flows, at the expense of higher communication overhead to report counts to a central coordinator.

Additionally, networks that forward high traffic volumes often resort to sampling $1/x$ packets, where x is operator-defined based on the needs of the specific network. However, sampling can result in substantially reduced accuracy on small time scales [21], even when traffic volumes are high. In Figure 1, we show the impact sampling has on accuracy while performing heavy-hitter detection on a link between two major ISPs [12] processing approximately 1 GBps of traffic. Even with high sampling rates, recall is quite low on short time intervals and it quickly diminishes as sampling rates decrease. In modern datacenter networks where switches

Figure 1: *This graph shows the recall for detecting heavy-hitters between two major ISPs [12] with different monitoring intervals. Even under high sampling rates, recall quickly diminishes and worsens as the monitoring interval decreases.*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSR '18, March 28–29, 2018, Los Angeles, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5664-0/18/03…$15.00
https://doi.org/10.1145/3185467.3185476

Develop P4-based detection mechanisms to

- diagnose TCP performance issue (e.g. small receiver buffers)
- heavy-hitter (e.g. port scanners, superspreader, DDoS)

Introduce techniques to make sketch-based monitoring more practical (by making sketches adaptive or "universal")

SketchLearn [SIGCOMM'18]

Elastic Sketch [SIGCOMM'18]

UnivMon [SIGCOMM'16]

p576-huang.pdf (page 1 of 15)
Search
p561-yang.pdf (page 1 of 15)
Search
sigcomm16_univmon.pdf (page 1 of 14)
Search

SketchLearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference

Elastic Sketch: Adaptive and Fast Network-wide Measurements

One Sketch to Rule Them All:
Rethinking Network Flow Monitoring with UnivMon

Zaoxing Liu†, Antonis Manousis⋆, Gregory Vorsanger†, Vyas Sekar⋆, Vladimir Braverman†
† Johns Hopkins University ⋆ Carnegie Mellon University

ABSTRACT

Network management requires accurate estimates of metrics for many applications including traffic engineering (e.g., heavy hitters), anomaly detection (e.g., entropy of source addresses), and security (e.g., DDoS detection). Obtaining accurate estimates given router CPU and memory constraints is a challenging problem. Existing approaches fall in one of two undesirable extremes: (1) low fidelity general-purpose approaches such as sampling, or (2) high fidelity but complex algorithms customized to specific application-level metrics. Ideally, a solution should be both general (i.e., supports many applications) and provide accuracy comparable to custom algorithms. This paper presents *UnivMon*, a framework for flow monitoring which leverages recent theoretical advances and demonstrates that it is possible to achieve both generality and high accuracy. UnivMon uses an application-agnostic data plane monitoring primitive; different (and possibly unforeseen) estimation algorithms run in the control plane, and use the statistics from the data plane to compute application-level metrics. We present a proof-of-concept implementation of UnivMon using P4 and develop simple coordination techniques to provide a "one-big-switch" abstraction for network-wide monitoring. We evaluate the effectiveness of UnivMon using a range of trace-driven evaluations and show that it offers comparable (and sometimes better) accuracy relative to custom sketching solutions across a range of monitoring tasks.

CCS Concepts

•Networks → Network monitoring; Network measurement;

Keywords

Flow Monitoring, Sketching, Streaming Algorithms

SIGCOMM '16, August 22–26, 2016, Florianopolis, Brazil
© 2016 ACM. ISBN 978-1-4503-4193-6/16/08…$15.00
DOI: http://dx.doi.org/10.1145/2934872.2934906

1 Introduction

Network management is multi-faceted and encompasses a range of tasks including traffic engineering [11, 32], attack and anomaly detection [49], and forensic analysis [46]. Each such management task requires accurate and timely statistics on different application-level metrics of interest; e.g., the flow size distribution [37], heavy hitters [10], entropy measures [38, 50], or detecting changes in traffic patterns [44].

At a high level, there are two classes of techniques to estimate these metrics of interest. The first class of approaches relies on *generic flow monitoring*, typically with some form of packet sampling (e.g., NetFlow [25]). While generic flow monitoring is good for coarse-grained visibility, prior work has shown that it provides low accuracy for more fine-grained metrics [30, 31, 43]. These well-known limitations of sampling motivated an alternative class of techniques based on *sketching* or *streaming* algorithms. Here, custom online algorithms and data structures are designed for specific metrics of interest that can yield provable resource-accuracy trade-offs (e.g., [17, 18, 20, 31, 36, 38, 43]).

While the body of work in data streaming and sketching has made significant contributions, we argue that this trajectory of crafting special-purpose algorithms is untenable in the long term. As the number of monitoring tasks grows, this entails significant investment in algorithm design and hardware support for new metrics of interest. While recent tools like OpenSketch [47] and SCREAM [41] provide libraries to reduce the implementation effort and offer efficient resource allocation, they do not address the fundamental need to design and operate new custom sketches for each task. Furthermore, at any given point in time the data plane resources have to be committed (a priori) to a specific set of metrics to monitor and will have fundamental blind spots for other metrics that are not currently being tracked.

Ideally, we want a monitoring framework that offers both *generality* by delaying the binding to specific applications of interest but at the same time provides the required *fidelity* for estimating these metrics. Achieving generality and high fidelity simultaneously has been an elusive goal both in theory [33] (Question 24) as well as in practice [45].

In this paper, we present the *UnivMon* (short for Universal Monitoring) framework that can simultaneously achieve both generality and high fidelity across a broad spectrum of monitoring tasks [31, 36, 38, 51]. UnivMon builds on and

Data plane programmability for

Performance

Monitoring

Applications offloading

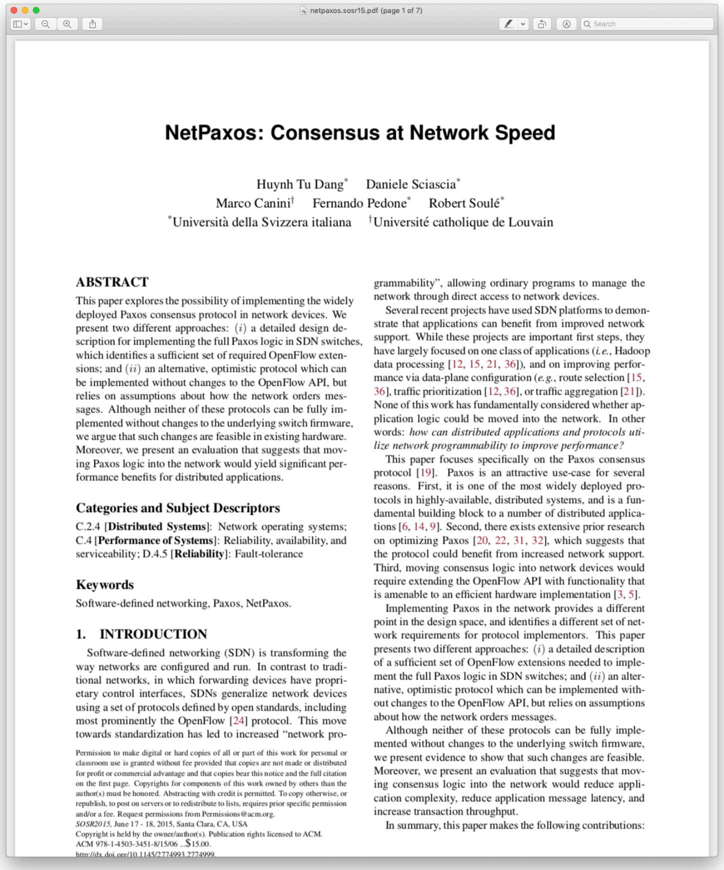
Platforms

Correctness

Management

for Data plane programmability

[SOSR'15]



NetPaxos: Consensus at Network Speed

Huynh Tu Dang* Daniele Sciascia*
Marco Canini† Fernando Pedone* Robert Soulé*
*Università della Svizzera italiana †Université catholique de Louvain

ABSTRACT

This paper explores the possibility of implementing the widely deployed Paxos consensus protocol in network devices. We present two different approaches: (i) a detailed design description for implementing the full Paxos logic in SDN switches, which identifies a sufficient set of required OpenFlow extensions; and (ii) an alternative, optimistic protocol which can be implemented without changes to the OpenFlow API, but relies on assumptions about how the network orders messages. Although neither of these protocols can be fully implemented without changes to the underlying switch firmware, we argue that such changes are feasible in existing hardware. Moreover, we present an evaluation that suggests that moving Paxos logic into the network would yield significant performance benefits for distributed applications.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Network operating systems; C.4 [**Performance of Systems**]: Reliability, availability, and serviceability; D.4.5 [**Reliability**]: Fault-tolerance

Keywords

Software-defined networking, Paxos, NetPaxos.

1. INTRODUCTION

Software-defined networking (SDN) is transforming the way networks are configured and run. In contrast to traditional networks, in which forwarding devices have proprietary control interfaces, SDNs generalize network devices using a set of protocols defined by open standards, including most prominently the OpenFlow [24] protocol. This move towards standardization has led to increased "network programmability", allowing ordinary programs to manage the network through direct access to network devices.

Several recent projects have used SDN platforms to demonstrate that applications can benefit from improved network support. While these projects are important first steps, they have largely focused on one class of applications (*i.e.*, Hadoop data processing [12, 15, 21, 36]), and on improving performance via data-plane configuration (*e.g.*, route selection [15, 36], traffic prioritization [12, 36], or traffic aggregation [21]). None of this work has fundamentally considered whether application logic could be moved into the network. In other words: *how can distributed applications and protocols utilize network programmability to improve performance?*

This paper focuses specifically on the Paxos consensus protocol [19]. Paxos is an attractive use-case for several reasons. First, it is one of the most widely deployed protocols in highly-available, distributed systems, and is a fundamental building block to a number of distributed applications [6, 14, 9]. Second, there exists extensive prior research on optimizing Paxos [20, 22, 31, 32], which suggests that the protocol could benefit from increased network support. Third, moving consensus logic into network devices would require extending the OpenFlow API with functionality that is amenable to an efficient hardware implementation [3, 5].

Implementing Paxos in the network provides a different point in the design space, and identifies a different set of network requirements for protocol implementors. This paper presents two different approaches: (i) a detailed description of a sufficient set of OpenFlow extensions needed to implement the full Paxos logic in SDN switches; and (ii) an alternative, optimistic protocol which can be implemented without changes to the OpenFlow API, but relies on assumptions about how the network orders messages.

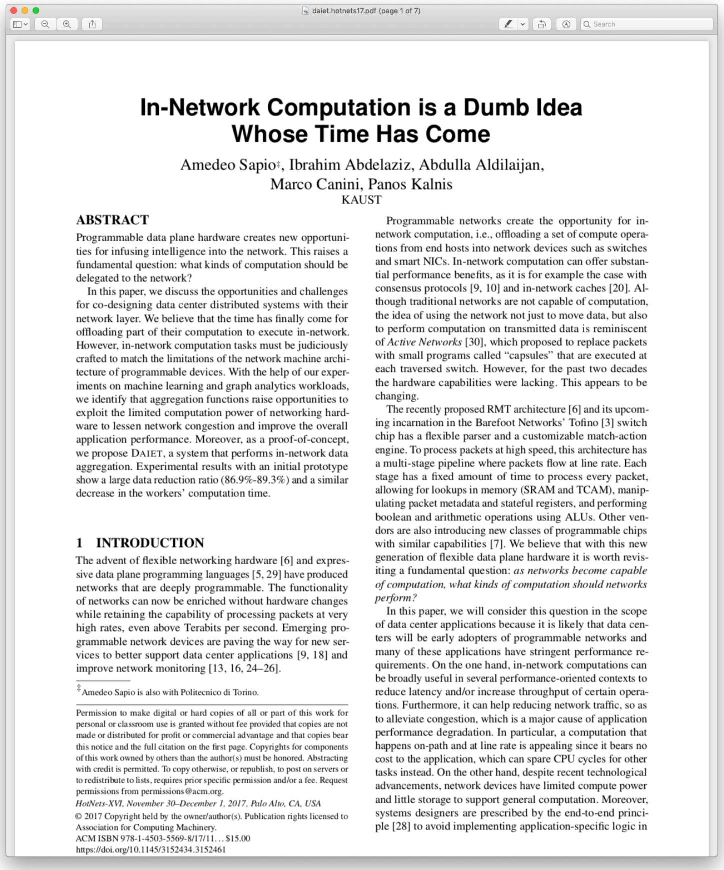
Although neither of these protocols can be fully implemented without changes to the underlying switch firmware, we present evidence to show that such changes are feasible. Moreover, we present an evaluation that suggests that moving consensus logic into the network would reduce application complexity, reduce application message latency, and increase transaction throughput.

In summary, this paper makes the following contributions:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SOSR2015, June 17 - 18, 2015, Santa Clara, CA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3451-8/15/06 ...$15.00.
http://dx.doi.org/10.1145/2774993.2774999.

Consensus at network speed

[HotNets'17]



In-Network Computation is a Dumb Idea Whose Time Has Come

Amedeo Sapio‡, Ibrahim Abdelaziz, Abdulla Aldilaijan,
Marco Canini, Panos Kalnis
KAUST

ABSTRACT

Programmable data plane hardware creates new opportunities for infusing intelligence into the network. This raises a fundamental question: what kinds of computation should be delegated to the network?

In this paper, we discuss the opportunities and challenges for co-designing data center distributed systems with their network layer. We believe that the time has finally come for offloading part of their computation to execute in-network. However, in-network computation tasks must be judiciously crafted to match the limitations of the network machine architecture of programmable devices. With the help of our experiments on machine learning and graph analytics workloads, we identify that aggregation functions raise opportunities to exploit the limited computation power of networking hardware to lessen network congestion and improve the overall application performance. Moreover, as a proof-of-concept, we propose DAIET, a system that performs in-network data aggregation. Experimental results with an initial prototype show a large data reduction ratio (86.9%-89.3%) and a similar decrease in the workers' computation time.

1 INTRODUCTION

The advent of flexible networking hardware [6] and expressive data plane programming languages [5, 29] have produced networks that are deeply programmable. The functionality of networks can now be enriched without hardware changes while retaining the capability of processing packets at very high rates, even above Terabits per second. Emerging programmable network devices are paving the way for new services to better support data center applications [9, 18] and improve network monitoring [13, 16, 24–26].

Programmable networks create the opportunity for in-network computation, i.e., offloading a set of compute operations from end hosts into network devices such as switches and smart NICs. In-network computation can offer substantial performance benefits, as it is for example the case with consensus protocols [9, 10] and in-network caches [20]. Although traditional networks are not capable of computation, the idea of using the network not just to move data, but also to perform computation on transmitted data is reminiscent of *Active Networks* [30], which proposed to replace packets with small programs called "capsules" that are executed at each traversed switch. However, for the past two decades the hardware capabilities were lacking. This appears to be changing.

The recently proposed RMT architecture [6] and its upcoming incarnation in the Barefoot Networks' Tofino [3] switch chip has a flexible parser and a customizable match-action engine. To process packets at high speed, this architecture has a multi-stage pipeline where packets flow at line rate. Each stage has a fixed amount of time to process every packet, allowing for lookups in memory (SRAM and TCAM), manipulating packet metadata and stateful registers, and performing boolean and arithmetic operations using ALUs. Other vendors are also introducing new classes of programmable chips with similar capabilities [7]. We believe that with this new generation of flexible data plane hardware it is worth revisiting a fundamental question: *as networks become capable of computation, what kinds of computation should networks perform?*

In this paper, we will consider this question in the scope of data center applications because it is likely that data centers will be early adopters of programmable networks and many of these applications have stringent performance requirements. On the one hand, in-network computations can be broadly useful in several performance-oriented contexts to reduce latency and/or increase throughput of certain operations. Furthermore, it can help reducing network traffic, so as to alleviate congestion, which is a major cause of application performance degradation. In particular, a computation that happens on-path and at line rate is appealing since it bears no cost to the application, which can spare CPU cycles for other tasks instead. On the other hand, despite recent technological advancements, network devices have limited compute power and little storage to support general computation. Moreover, systems designers are prescribed by the end-to-end principle [28] to avoid implementing application-specific logic in

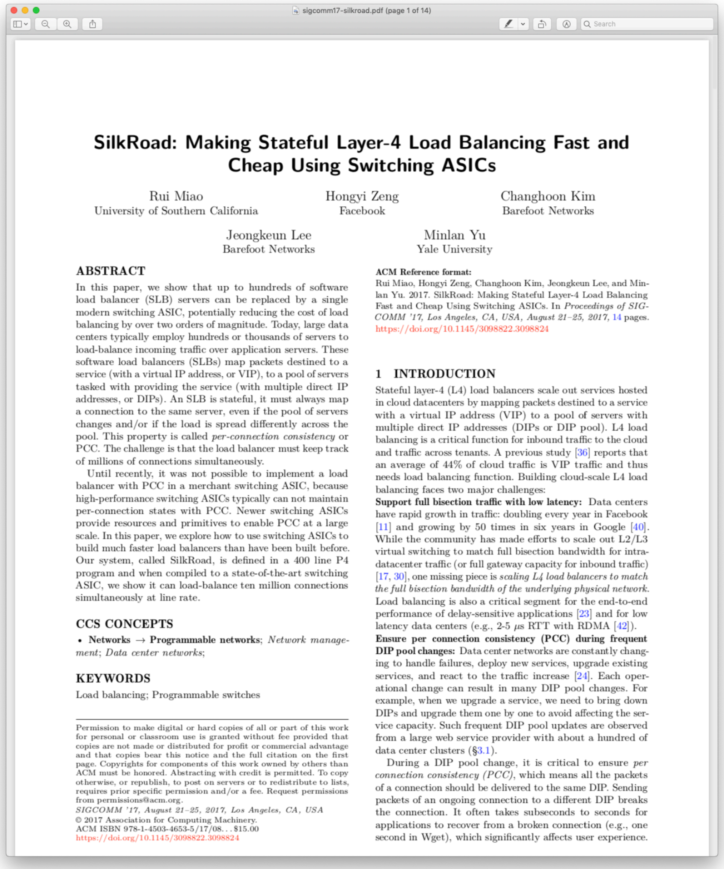
[‡] Amedeo Sapio is also with Politecnico di Torino.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HotNets-XVI, November 30–December 1, 2017, Palo Alto, CA, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5569-8/17/11…$15.00
https://doi.org/10.1145/3152434.3152461

In-Network Aggregation

(e.g., for MapReduce, graph analytics, ML)

[SIGCOMM'17]



SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs

Rui Miao
University of Southern California

Hongyi Zeng
Facebook

Changhoon Kim
Barefoot Networks

Jeongkeun Lee
Barefoot Networks

Minlan Yu
Yale University

ABSTRACT

In this paper, we show that up to hundreds of software load balancer (SLB) servers can be replaced by a single modern switching ASIC, potentially reducing the cost of load balancing by over two orders of magnitude. Today, large data centers typically employ hundreds or thousands of servers to load-balance incoming traffic over application servers. These software load balancers (SLBs) map packets destined to a service (with a virtual IP address, or VIP), to a pool of servers tasked with providing the service (with multiple direct IP addresses, or DIPs). An SLB is stateful, it must always map a connection to the same server, even if the pool of servers changes and/or if the load is spread differently across the pool. This property is called *per-connection consistency* or PCC. The challenge is that the load balancer must keep track of millions of connections simultaneously.

Until recently, it was not possible to implement a load balancer with PCC in a merchant switching ASIC, because high-performance switching ASICs typically can not maintain per-connection states with PCC. Newer switching ASICs provide resources and primitives to enable PCC at a large scale. In this paper, we explore how to use switching ASICs to build much faster load balancers than have been built before. Our system, called SilkRoad, is defined in a 400 line P4 program and when compiled to a state-of-the-art switching ASIC, we show it can load-balance ten million connections simultaneously at line rate.

CCS CONCEPTS

- **Networks** → **Programmable networks**; *Network management*; *Data center networks*;

KEYWORDS

Load balancing; Programmable switches

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCOMM '17, August 21–25, 2017, Los Angeles, CA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4653-5/17/08. . . $15.00
https://doi.org/10.1145/3098822.3098824

ACM Reference format:
Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21–25, 2017,* 14 pages.
https://doi.org/10.1145/3098822.3098824

1 INTRODUCTION

Stateful layer-4 (L4) load balancers scale out services hosted in cloud datacenters by mapping packets destined to a service with a virtual IP address (VIP) to a pool of servers with multiple direct IP addresses (DIPs or DIP pool). L4 load balancing is a critical function for inbound traffic to the cloud and traffic across tenants. A previous study [36] reports that an average of 44% of cloud traffic is VIP traffic and thus needs load balancing function. Building cloud-scale L4 load balancing faces two major challenges:

Support full bisection traffic with low latency: Data centers have rapid growth in traffic: doubling every year in Facebook [11] and growing by 50 times in six years in Google [40]. While the community has made efforts to scale out L2/L3 virtual switching to match full bisection bandwidth for intra-datacenter traffic (or full gateway capacity for inbound traffic) [17, 30], one missing piece is *scaling L4 load balancers to match the full bisection bandwidth of the underlying physical network.* Load balancing is also a critical segment for the end-to-end performance of delay-sensitive applications [23] and for low latency data centers (e.g., 2-5 μs RTT with RDMA [42]).

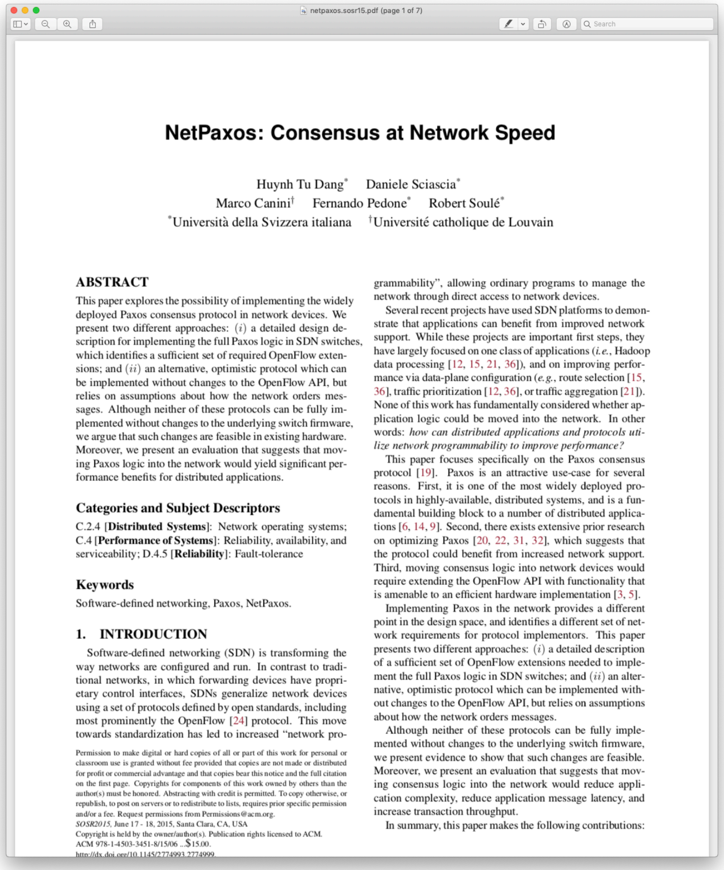
Ensure per connection consistency (PCC) during frequent DIP pool changes: Data center networks are constantly changing to handle failures, deploy new services, upgrade existing services, and react to the traffic increase [24]. Each operational change can result in many DIP pool changes. For example, when we upgrade a service, we need to bring down DIPs and upgrade them one by one to avoid affecting the service capacity. Such frequent DIP pool updates are observed from a large web service provider with about a hundred of data center clusters (§3.1).

During a DIP pool change, it is critical to ensure *per connection consistency (PCC)*, which means all the packets of a connection should be delivered to the same DIP. Sending packets of an ongoing connection to a different DIP breaks the connection. It often takes subseconds to seconds for applications to recover from a broken connection (e.g., one second in Wget), which significantly affects user experience.

Stateful layer-4 load balancers

+ NetCache [SOSP'17], NetChain [NSDI'18]

[SOSR'15]



NetPaxos: Consensus at Network Speed

Huynh Tu Dang[*] Daniele Sciascia[*]
Marco Canini[†] Fernando Pedone[*] Robert Soulé[*]
[*]Università della Svizzera italiana [†]Université catholique de Louvain

ABSTRACT

This paper explores the possibility of implementing the widely deployed Paxos consensus protocol in network devices. We present two different approaches: (i) a detailed design description for implementing the full Paxos logic in SDN switches, which identifies a sufficient set of required OpenFlow extensions; and (ii) an alternative, optimistic protocol which can be implemented without changes to the OpenFlow API, but relies on assumptions about how the network orders messages. Although neither of these protocols can be fully implemented without changes to the underlying switch firmware, we argue that such changes are feasible in existing hardware. Moreover, we present an evaluation that suggests that moving Paxos logic into the network would yield significant performance benefits for distributed applications.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Network operating systems; C.4 [**Performance of Systems**]: Reliability, availability, and serviceability; D.4.5 [**Reliability**]: Fault-tolerance

Keywords

Software-defined networking, Paxos, NetPaxos.

1. INTRODUCTION

Software-defined networking (SDN) is transforming the way networks are configured and run. In contrast to traditional networks, in which forwarding devices have proprietary control interfaces, SDNs generalize network devices using a set of protocols defined by open standards, including most prominently the OpenFlow [24] protocol. This move towards standardization has led to increased "network programmability", allowing ordinary programs to manage the network through direct access to network devices.

Several recent projects have used SDN platforms to demonstrate that applications can benefit from improved network support. While these projects are important first steps, they have largely focused on one class of applications (*i.e.*, Hadoop data processing [12, 15, 21, 36]), and on improving performance via data-plane configuration (*e.g.*, route selection [15, 36], traffic prioritization [12, 36], or traffic aggregation [21]). None of this work has fundamentally considered whether application logic could be moved into the network. In other words: *how can distributed applications and protocols utilize network programmability to improve performance?*

This paper focuses specifically on the Paxos consensus protocol [19]. Paxos is an attractive use-case for several reasons. First, it is one of the most widely deployed protocols in highly-available, distributed systems, and is a fundamental building block to a number of distributed applications [6, 14, 9]. Second, there exists extensive prior research on optimizing Paxos [20, 22, 31, 32], which suggests that the protocol could benefit from increased network support. Third, moving consensus logic into network devices would require extending the OpenFlow API with functionality that is amenable to an efficient hardware implementation [3, 5].

Implementing Paxos in the network provides a different point in the design space, and identifies a different set of network requirements for protocol implementors. This paper presents two different approaches: (i) a detailed description of a sufficient set of OpenFlow extensions needed to implement the full Paxos logic in SDN switches; and (ii) an alternative, optimistic protocol which can be implemented without changes to the OpenFlow API, but relies on assumptions about how the network orders messages.

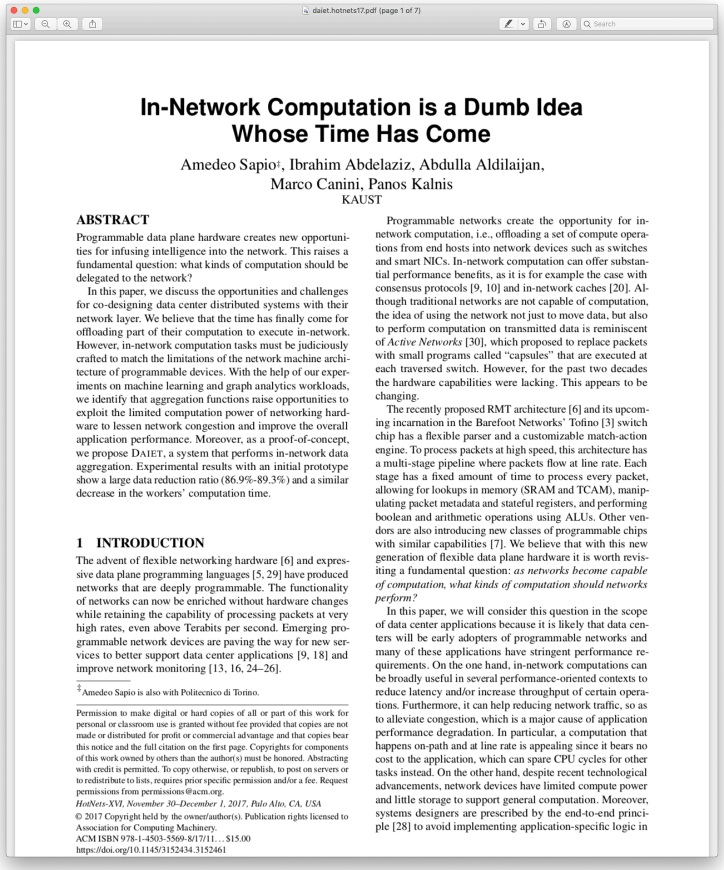
Although neither of these protocols can be fully implemented without changes to the underlying switch firmware, we present evidence to show that such changes are feasible. Moreover, we present an evaluation that suggests that moving consensus logic into the network would reduce application complexity, reduce application message latency, and increase transaction throughput.

In summary, this paper makes the following contributions:

SOSR2015, June 17 - 18, 2015, Santa Clara, CA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3451-8/15/06 ...$15.00.
http://dx.doi.org/10.1145/2774993.2774999.

Consensus at network speed

[HotNets'17]



In-Network Computation is a Dumb Idea Whose Time Has Come

Amedeo Sapio[‡], Ibrahim Abdelaziz, Abdulla Aldilaijan,
Marco Canini, Panos Kalnis
KAUST

ABSTRACT

Programmable data plane hardware creates new opportunities for infusing intelligence into the network. This raises a fundamental question: what kinds of computation should be delegated to the network?

In this paper, we discuss the opportunities and challenges for co-designing data center distributed systems with their network layer. We believe that the time has finally come for offloading part of their computation to execute in-network. However, in-network computation tasks must be judiciously crafted to match the limitations of the network machine architecture of programmable devices. With the help of our experiments on machine learning and graph analytics workloads, we identify that aggregation functions raise opportunities to exploit the limited computation power of networking hardware to lessen network congestion and improve the overall application performance. Moreover, as a proof-of-concept, we propose DAIET, a system that performs in-network data aggregation. Experimental results with an initial prototype show a large data reduction ratio (86.9%-89.3%) and a similar decrease in the workers' computation time.

1 INTRODUCTION

The advent of flexible networking hardware [6] and expressive data plane programming languages [5, 29] have produced networks that are deeply programmable. The functionality of networks can now be enriched without hardware changes while retaining the capability of processing packets at very high rates, even above Terabits per second. Emerging programmable network devices are paving the way for new services to better support data center applications [9, 18] and improve network monitoring [13, 16, 24–26].

Programmable networks create the opportunity for in-network computation, i.e., offloading a set of compute operations from end hosts into network devices such as switches and smart NICs. In-network computation can offer substantial performance benefits, as it is for example the case with consensus protocols [9, 10] and in-network caches [20]. Although traditional networks are not capable of computation, the idea of using the network not just to move data, but also to perform computation on transmitted data is reminiscent of *Active Networks* [30], which proposed to replace packets with small programs called "capsules" that are executed at each traversed switch. However, for the past two decades the hardware capabilities were lacking. This appears to be changing.

The recently proposed RMT architecture [6] and its upcoming incarnation in the Barefoot Networks' Tofino [3] switch chip has a flexible parser and a customizable match-action engine. To process packets at high speed, this architecture has a multi-stage pipeline where packets flow at line rate. Each stage has a fixed amount of time to process every packet, allowing for lookups in memory (SRAM and TCAM), manipulating packet metadata and stateful registers, and performing boolean and arithmetic operations using ALUs. Other vendors are also introducing new classes of programmable chips with similar capabilities [7]. We believe that with this new generation of flexible data plane hardware it is worth revisiting a fundamental question: *as networks become capable of computation, what kinds of computation should networks perform?*

In this paper, we will consider this question in the scope of data center applications because it is likely that data centers will be early adopters of programmable networks and many of these applications have stringent performance requirements. On the one hand, in-network computations can be broadly useful in several performance-oriented contexts to reduce latency and/or increase throughput of certain operations. Furthermore, it can help reducing network traffic, so as to alleviate congestion, which is a major cause of application performance degradation. In particular, a computation that happens on-path and at line rate is appealing since it bears no cost to the application, which can spare CPU cycles for other tasks instead. On the other hand, despite recent technological advancements, network devices have limited compute power and little storage to support general computation. Moreover, systems designers are prescribed by the end-to-end principle [28] to avoid implementing application-specific logic in

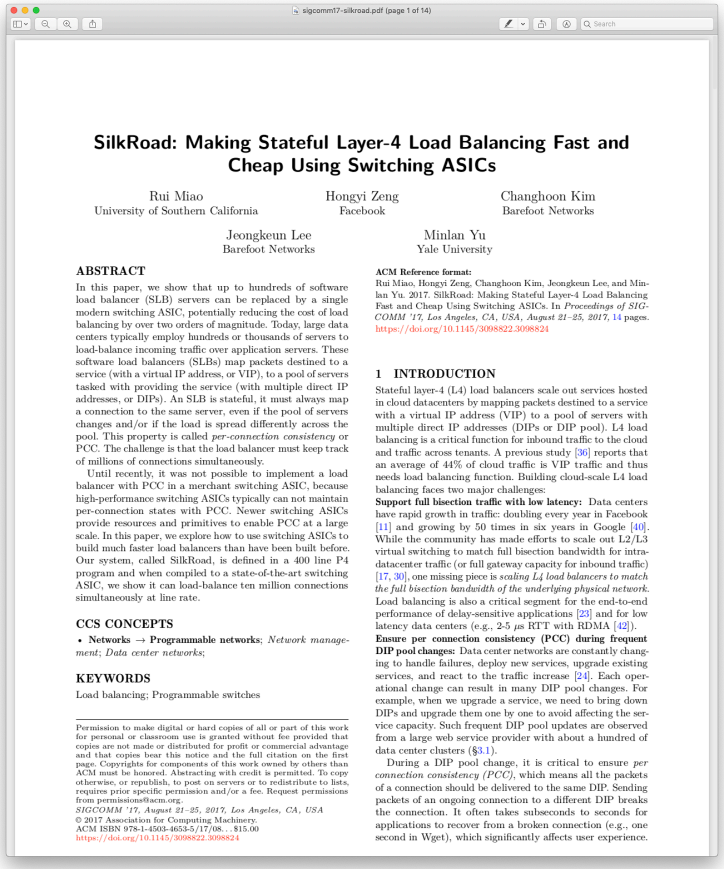
[‡] Amedeo Sapio is also with Politecnico di Torino.

HotNets-XVI, November 30–December 1, 2017, Palo Alto, CA, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5569-8/17/11…$15.00
https://doi.org/10.1145/3152434.3152461

In-Network Aggregation

(e.g., for MapReduce, graph analytics, ML)

[SIGCOMM'17]



SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs

Rui Miao
University of Southern California

Hongyi Zeng
Facebook

Changhoon Kim
Barefoot Networks

Jeongkeun Lee
Barefoot Networks

Minlan Yu
Yale University

ABSTRACT

In this paper, we show that up to hundreds of software load balancer (SLB) servers can be replaced by a single modern switching ASIC, potentially reducing the cost of load balancing by over two orders of magnitude. Today, large data centers typically employ hundreds or thousands of servers to load-balance incoming traffic over application servers. These software load balancers (SLBs) map packets destined to a service (with a virtual IP address, or VIP), to a pool of servers tasked with providing the service (with multiple direct IP addresses, or DIPs). An SLB is stateful, it must always map a connection to the same server, even if the pool of servers changes and/or if the load is spread differently across the pool. This property is called *per-connection consistency* or PCC. The challenge is that the load balancer must keep track of millions of connections simultaneously.

Until recently, it was not possible to implement a load balancer with PCC in a merchant switching ASIC, because high-performance switching ASICs typically can not maintain per-connection states with PCC. Newer switching ASICs provide resources and primitives to enable PCC at a large scale. In this paper, we explore how to use switching ASICs to build much faster load balancers than have been built before. Our system, called SilkRoad, is defined in a 400 line P4 program and when compiled to a state-of-the-art switching ASIC, we show it can load-balance ten million connections simultaneously at line rate.

CCS CONCEPTS

- **Networks** → **Programmable networks**; *Network management*; *Data center networks*;

KEYWORDS

Load balancing; Programmable switches

SIGCOMM '17, August 21–25, 2017, Los Angeles, CA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4653-5/17/08…$15.00
https://doi.org/10.1145/3098822.3098824

ACM Reference format:
Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21–25, 2017,* 14 pages.
https://doi.org/10.1145/3098822.3098824

1 INTRODUCTION

Stateful layer-4 (L4) load balancers scale out services hosted in cloud datacenters by mapping packets destined to a service with a virtual IP address (VIP) to a pool of servers with multiple direct IP addresses (DIPs or DIP pool). L4 load balancing is a critical function for inbound traffic to the cloud and traffic across tenants. A previous study [36] reports that an average of 44% of cloud traffic is VIP traffic and thus needs load balancing function. Building cloud-scale L4 load balancing faces two major challenges:

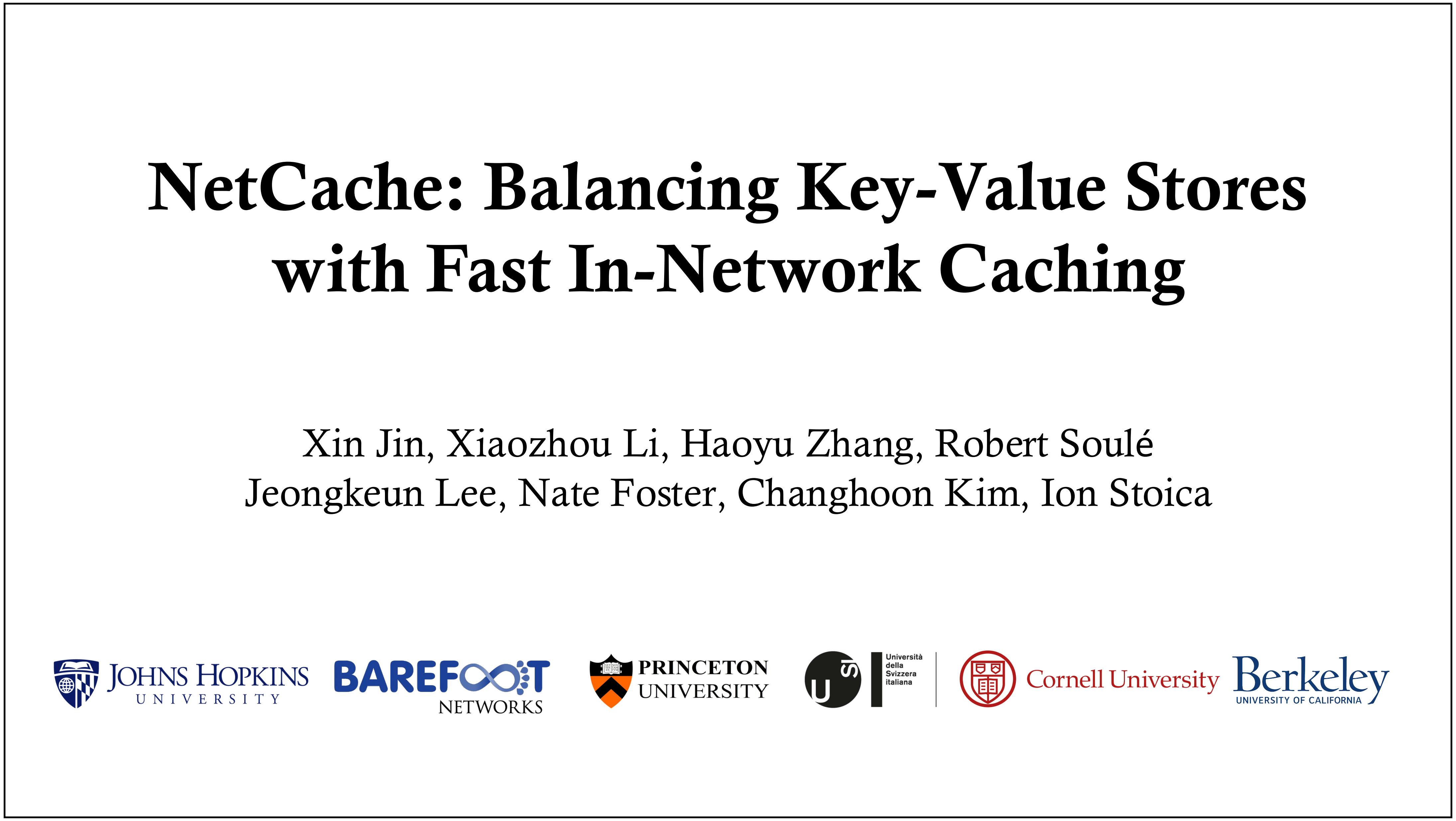
Support full bisection traffic with low latency: Data centers have rapid growth in traffic: doubling every year in Facebook [11] and growing by 50 times in six years in Google [40]. While the community has made efforts to scale out L2/L3 virtual switching to match full bisection bandwidth for intra-datacenter traffic (or full gateway capacity for inbound traffic) [17, 30], one missing piece is *scaling L4 load balancers to match the full bisection bandwidth of the underlying physical network*. Load balancing is also a critical segment for the end-to-end performance of delay-sensitive applications [23] and for low latency data centers (e.g., 2-5 μs RTT with RDMA [42]).

Ensure per connection consistency (PCC) during frequent DIP pool changes: Data center networks are constantly changing to handle failures, deploy new services, upgrade existing services, and react to the traffic increase [24]. Each operational change can result in many DIP pool changes. For example, when we upgrade a service, we need to bring down DIPs and upgrade them one by one to avoid affecting the service capacity. Such frequent DIP pool updates are observed from a large web service provider with about a hundred of data center clusters (§3.1).

During a DIP pool change, it is critical to ensure *per connection consistency (PCC)*, which means all the packets of a connection should be delivered to the same DIP. Sending packets of an ongoing connection to a different DIP breaks the connection. It often takes subseconds to seconds for applications to recover from a broken connection (e.g., one second in Wget), which significantly affects user experience.

Stateful layer-4 load balancers

+ NetCache [SOSP'17], NetChain [NSDI'18]



NetCache: Balancing Key-Value Stores with Fast In-Network Caching

Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé
Jeongkeun Lee, Nate Foster, Changhoon Kim, Ion Stoica

Source: NetCache: Balancing Key-Value Stores with Fast In-Network Caching, Xin Jin, 2017

NetCache solves the problem of load-balancing in key-values stores observing *dynamic*, *skewed* workload

NetCache is a **rack-scale key-value store** that leverages **in-network data plane caching** to achieve

billions QPS throughput & **~10 µs latency**

even under

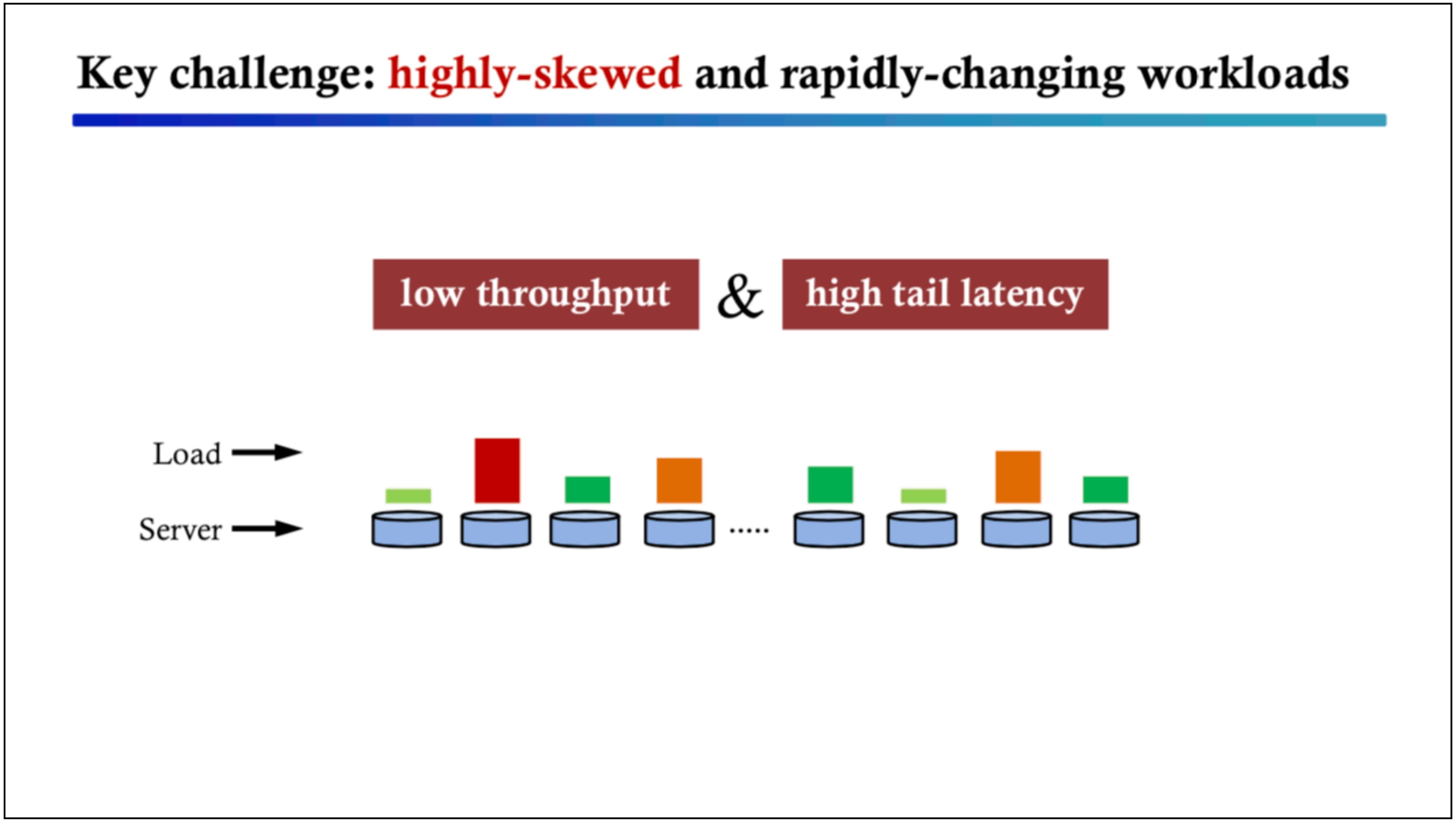
highly-skewed & **rapidly-changing**

workloads.

New generation of systems enabled by programmable switches ☺

Source: NetCache: Balancing Key-Value Stores with Fast In-Network Caching, Xin Jin, 2017

NetCache solves the problem of load-balancing in key-values stores observing *dynamic*, *skewed* workload

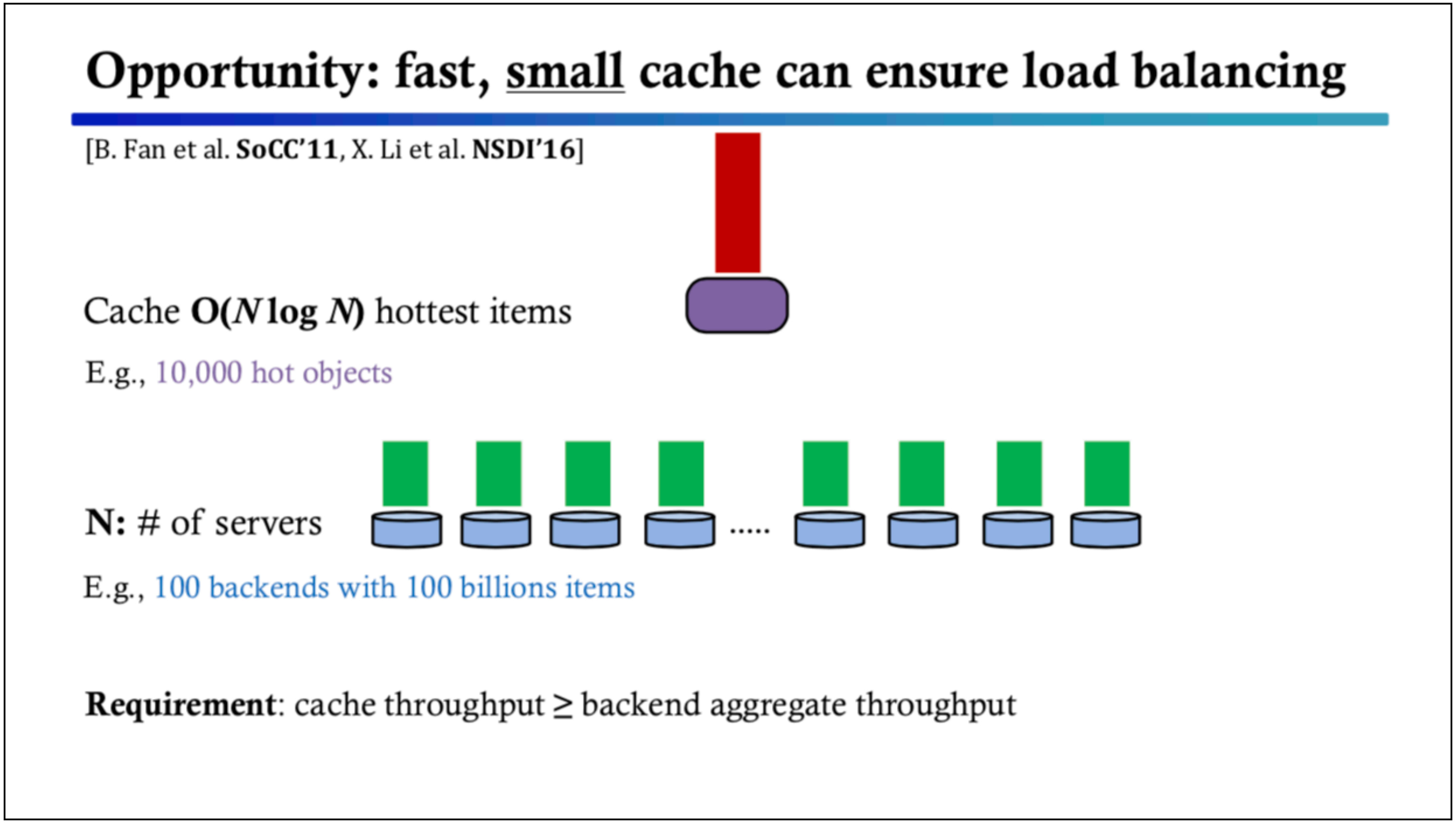


Key challenge: highly-skewed and rapidly-changing workloads

low throughput & high tail latency

Source: NetCache: Balancing Key-Value Stores with Fast In-Network Caching, Xin Jin, 2017

It leverages that a small but very fast cache can provide perfect load-balancing… in theory



Opportunity: fast, small cache can ensure load balancing

[B. Fan et al. **SoCC'11**, X. Li et al. **NSDI'16**]

Cache **O(N log N)** hottest items

E.g., 10,000 hot objects

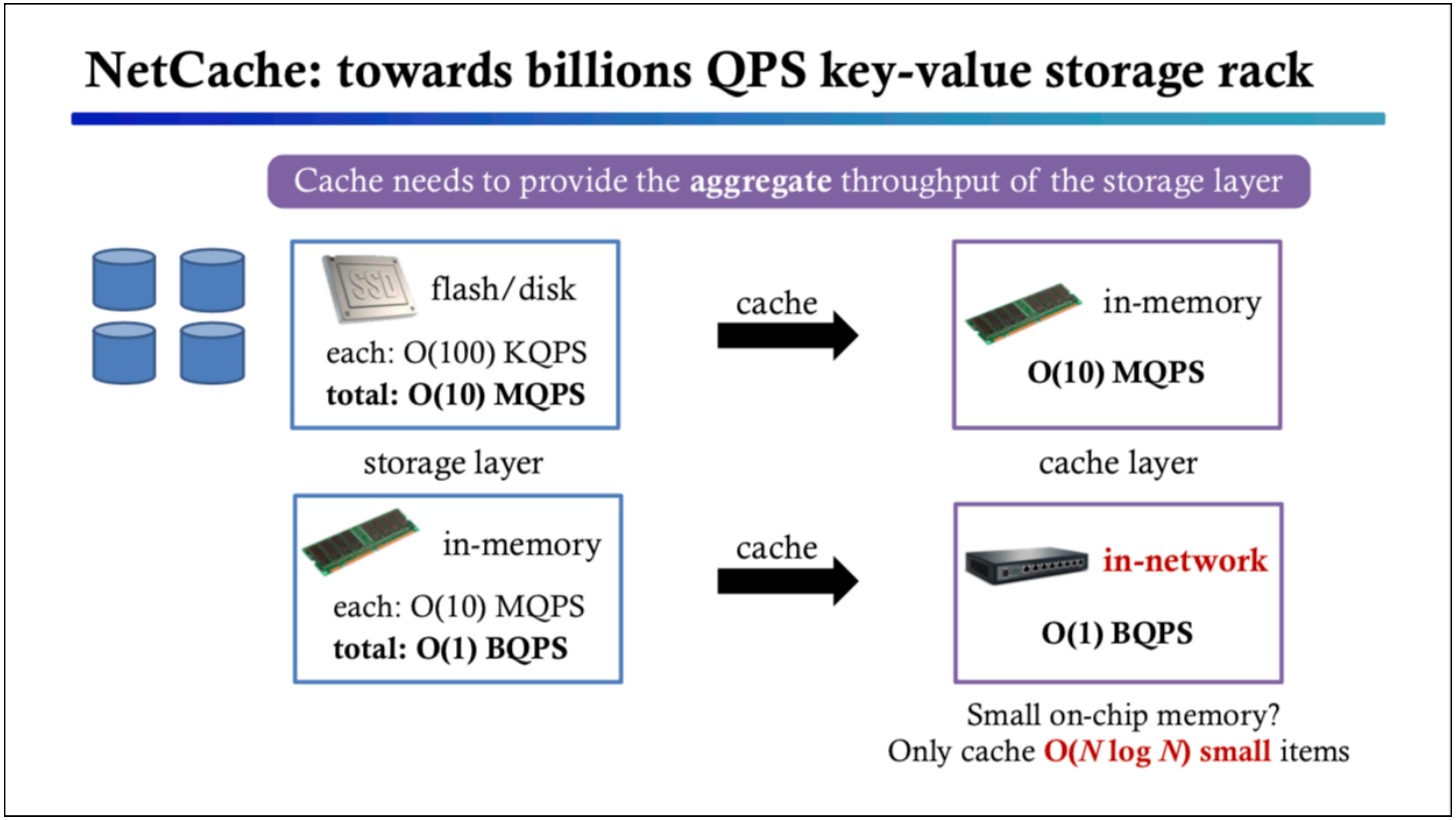
N: # of servers

E.g., 100 backends with 100 billions items

Requirement: cache throughput ≥ backend aggregate throughput

Source: NetCache: Balancing Key-Value Stores with Fast In-Network Caching, Xin Jin, 2017

NetCache relies on the O(billion) throughput of programmable network devices to achieve it in practice



NetCache: towards billions QPS key-value storage rack

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

flash/disk

each: O(100) KQPS

total: O(10) MQPS

storage layer

cache

in-memory

O(10) MQPS

cache layer

in-memory

each: O(10) MQPS

total: O(1) BQPS

cache

in-network

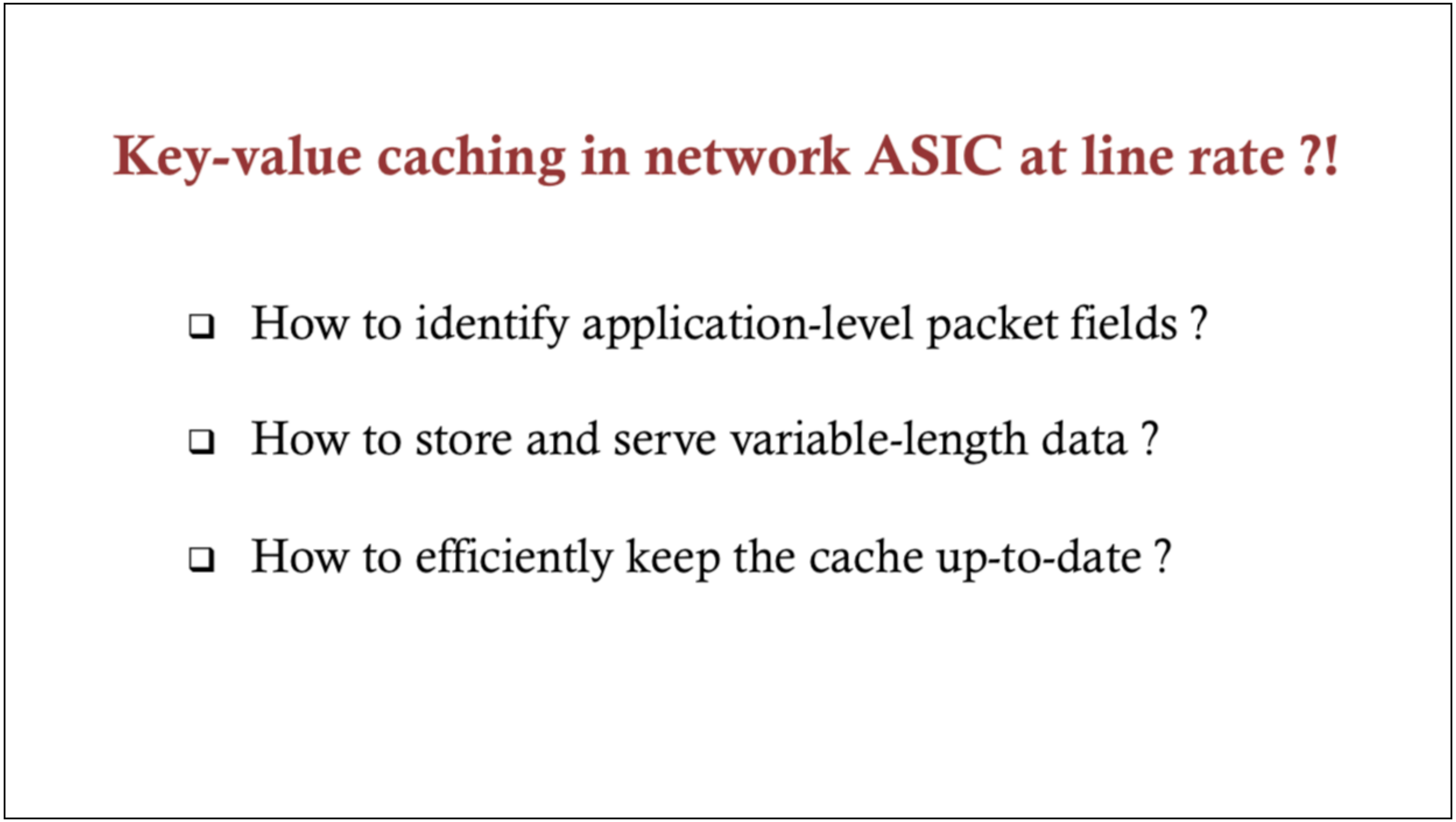
O(1) BQPS

Small on-chip memory?

Only cache **O(*N* log *N*) small** items

Source: NetCache: Balancing Key-Value Stores with Fast In-Network Caching, Xin Jin, 2017

It relies on a tailored UDP-based protocol, an de/encoding scheme for storing variable length values, and sketches



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

❑ How to identify application-level packet fields ?

❑ How to store and serve variable-length data ?

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

Source: NetCache: Balancing Key-Value Stores with Fast In-Network Caching, Xin Jin, 2017

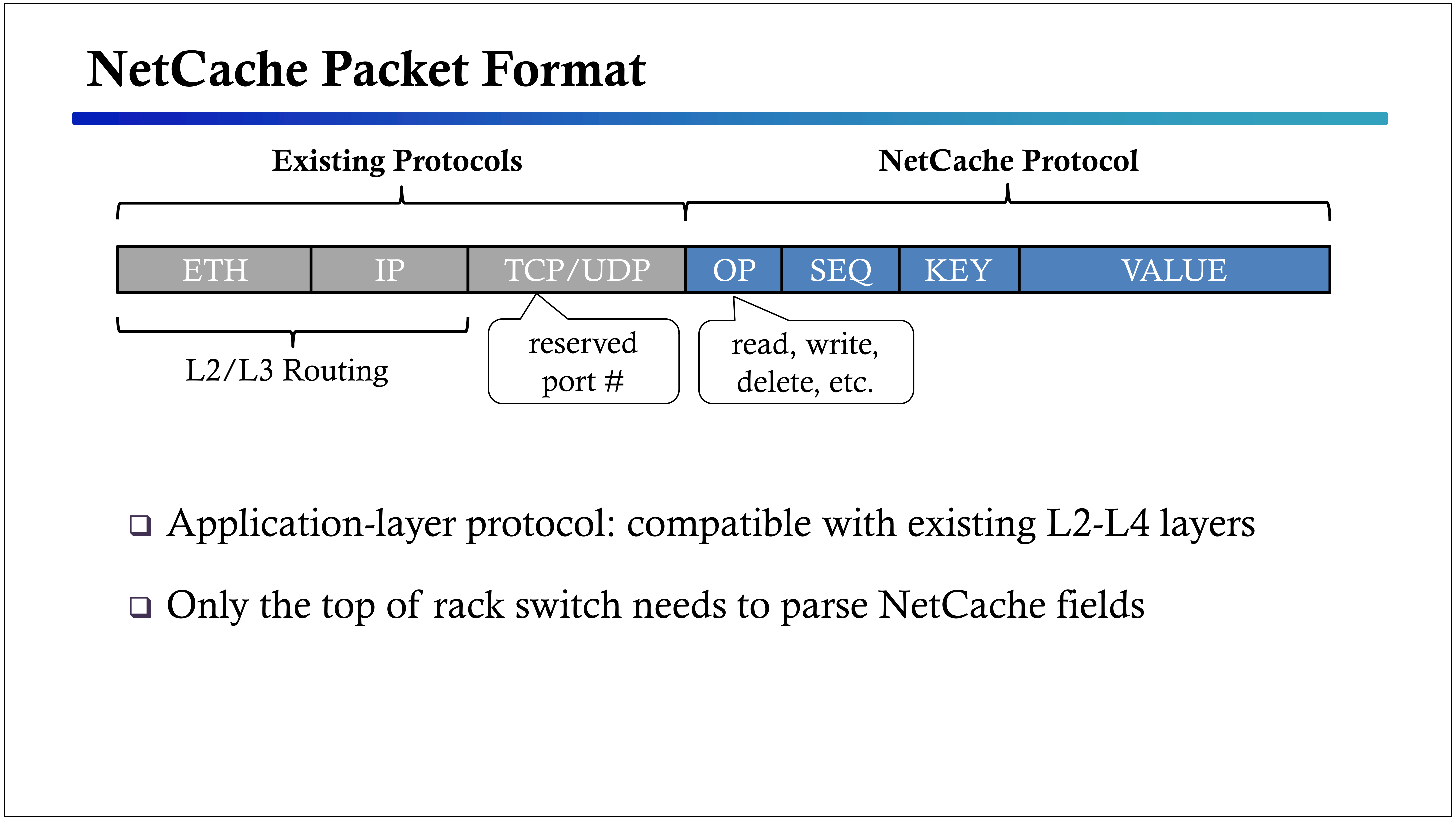
Key-value caching in network ASIC at line rate

→ ❑ How to identify application-level packet fields ?

❑ How to store and serve variable-length data ?

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

Source: NetCache: Balancing Key-Value Stores with Fast In-Network Caching, Xin Jin, 2017



NetCache Packet Format

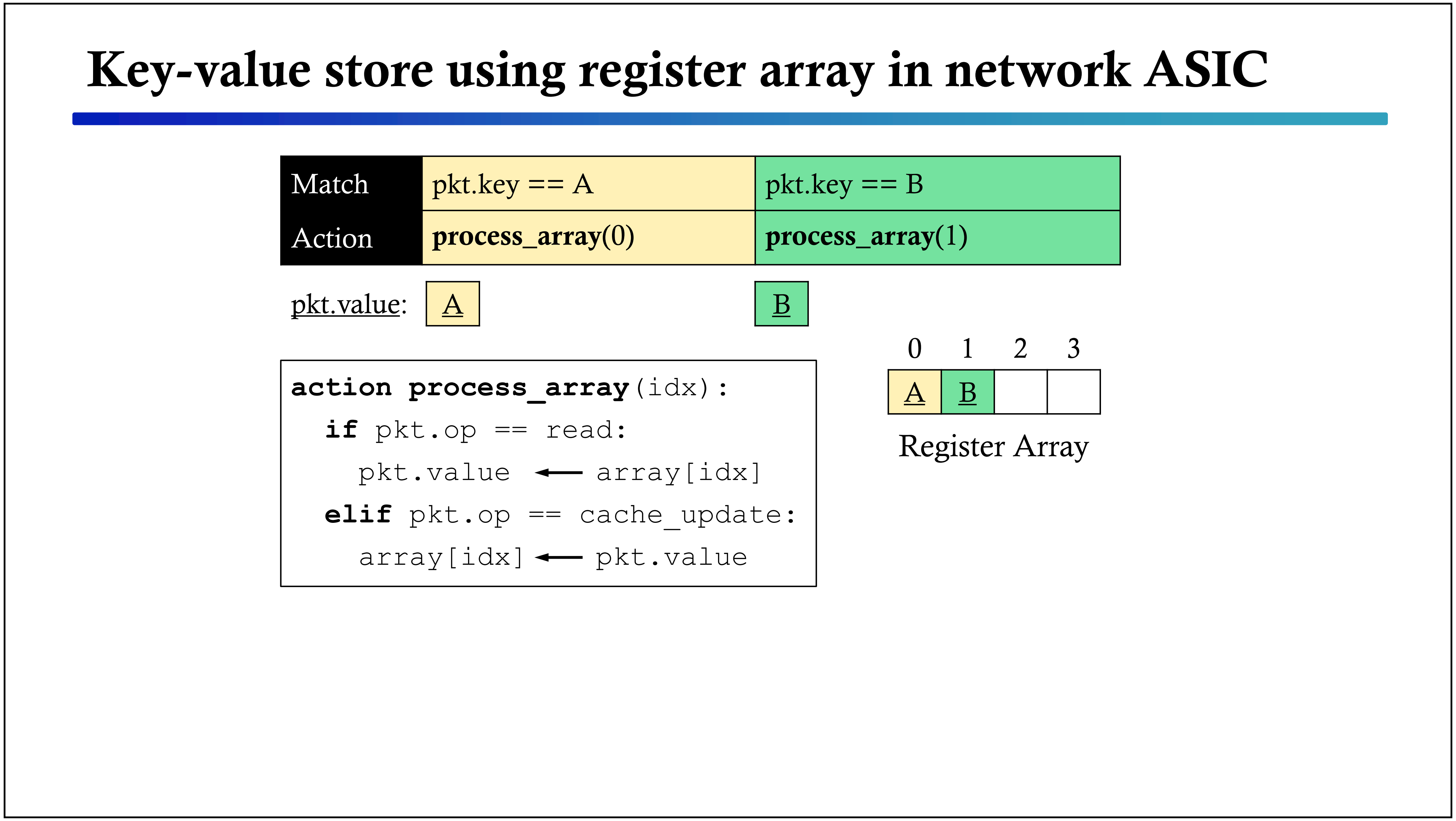
- ❑ Application-layer protocol: compatible with existing L2-L4 layers
- ❑ Only the top of rack switch needs to parse NetCache fields

Source: NetCache: Balancing Key-Value Stores with Fast In-Network Caching, Xin Jin, 2017

Key-value caching in network ASIC at line rate

- ❑ How to identify application-level packet fields ?
- → ❑ How to store and serve variable-length data ?
- ❑ How to efficiently keep the cache up-to-date ?

Source: NetCache: Balancing Key-Value Stores with Fast In-Network Caching, Xin Jin, 2017



Key-value store using register array in network ASIC

Match	pkt.key == A	pkt.key == B
Action	**process_array**(0)	**process_array**(1)

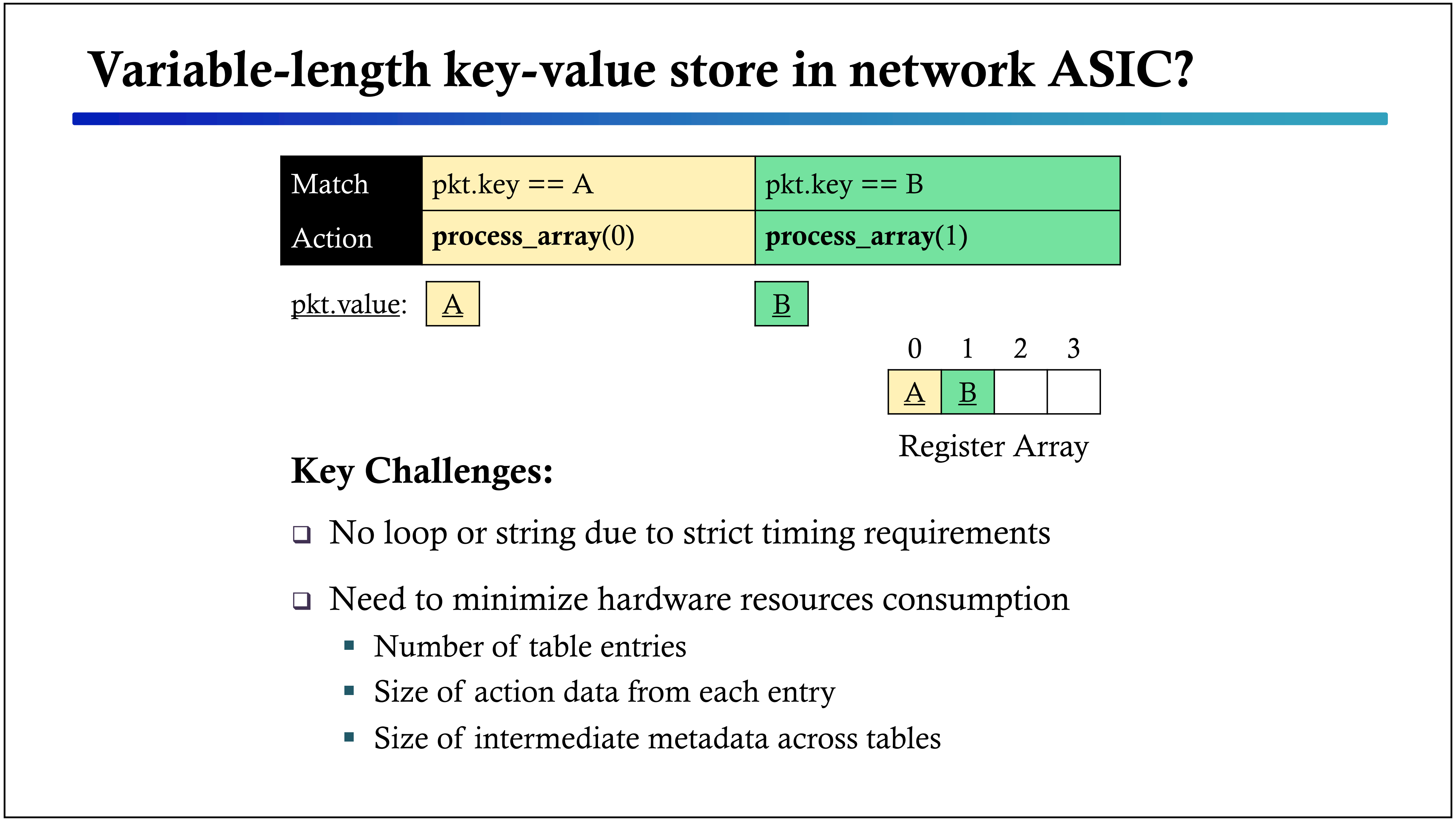
pkt.value: A B

```
action process_array(idx):
   if pkt.op == read:
      pkt.value ⟵ array[idx]
   elif pkt.op == cache_update:
      array[idx] ⟵ pkt.value
```

0	1	2	3
A	B		

Register Array

Source: NetCache: Balancing Key-Value Stores with Fast In-Network Caching, Xin Jin, 2017



Variable-length key-value store in network ASIC?

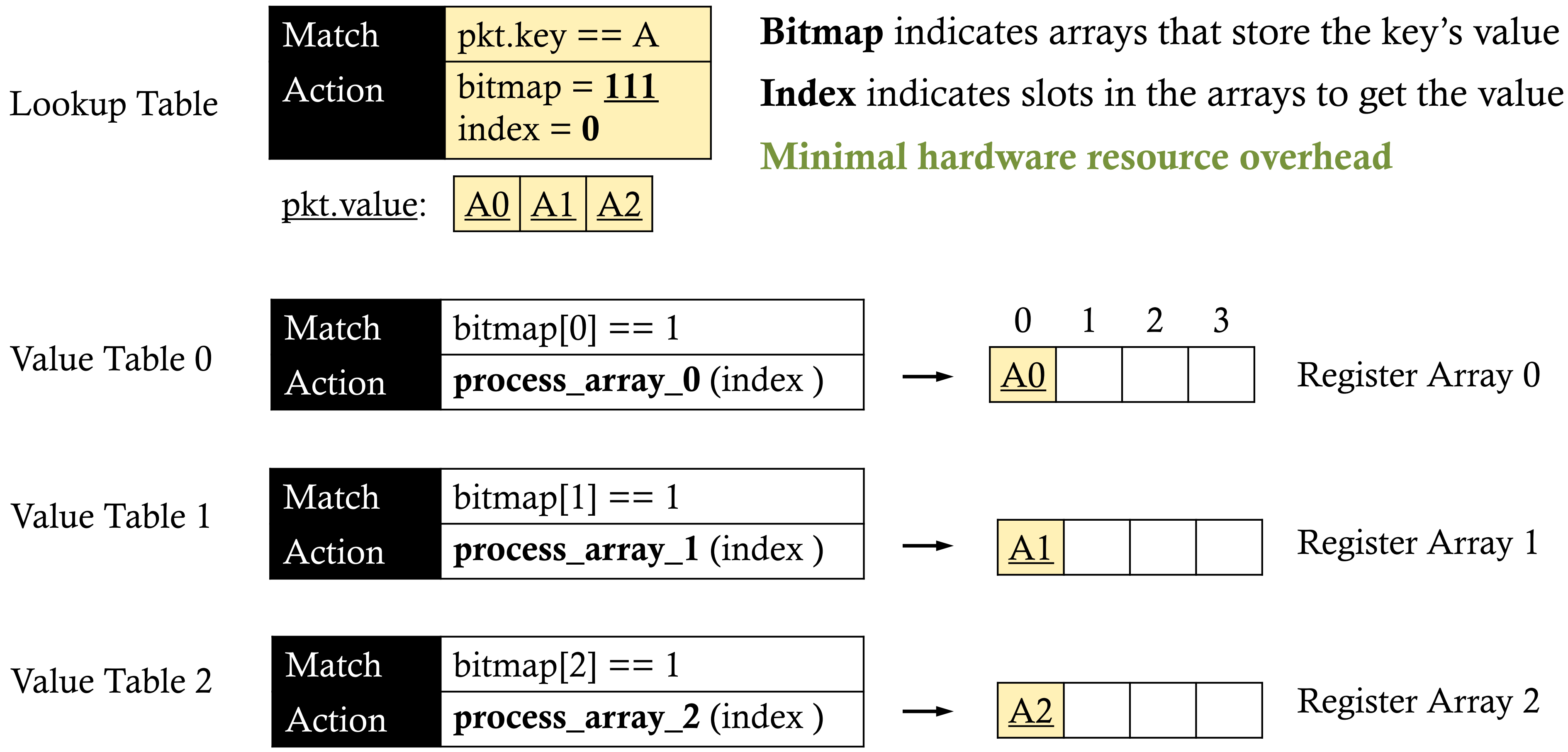
Match	pkt.key == A	pkt.key == B
Action	**process_array**(0)	**process_array**(1)

Key Challenges:

- No loop or string due to strict timing requirements
- Need to minimize hardware resources consumption
  - Number of table entries
  - Size of action data from each entry
  - Size of intermediate metadata across tables

Source: NetCache: Balancing Key-Value Stores with Fast In-Network Caching, Xin Jin, 2017

Combine outputs from multiple arrays



Lookup Table

Match	pkt.key == A

Action	bitmap = **111**
	index = **0**

pkt.value: A0 | A1 | A2

Bitmap indicates arrays that store the key's value

Index indicates slots in the arrays to get the value

Minimal hardware resource overhead

Value Table 0

Match	bitmap[0] == 1
Action	**process_array_0** (index)

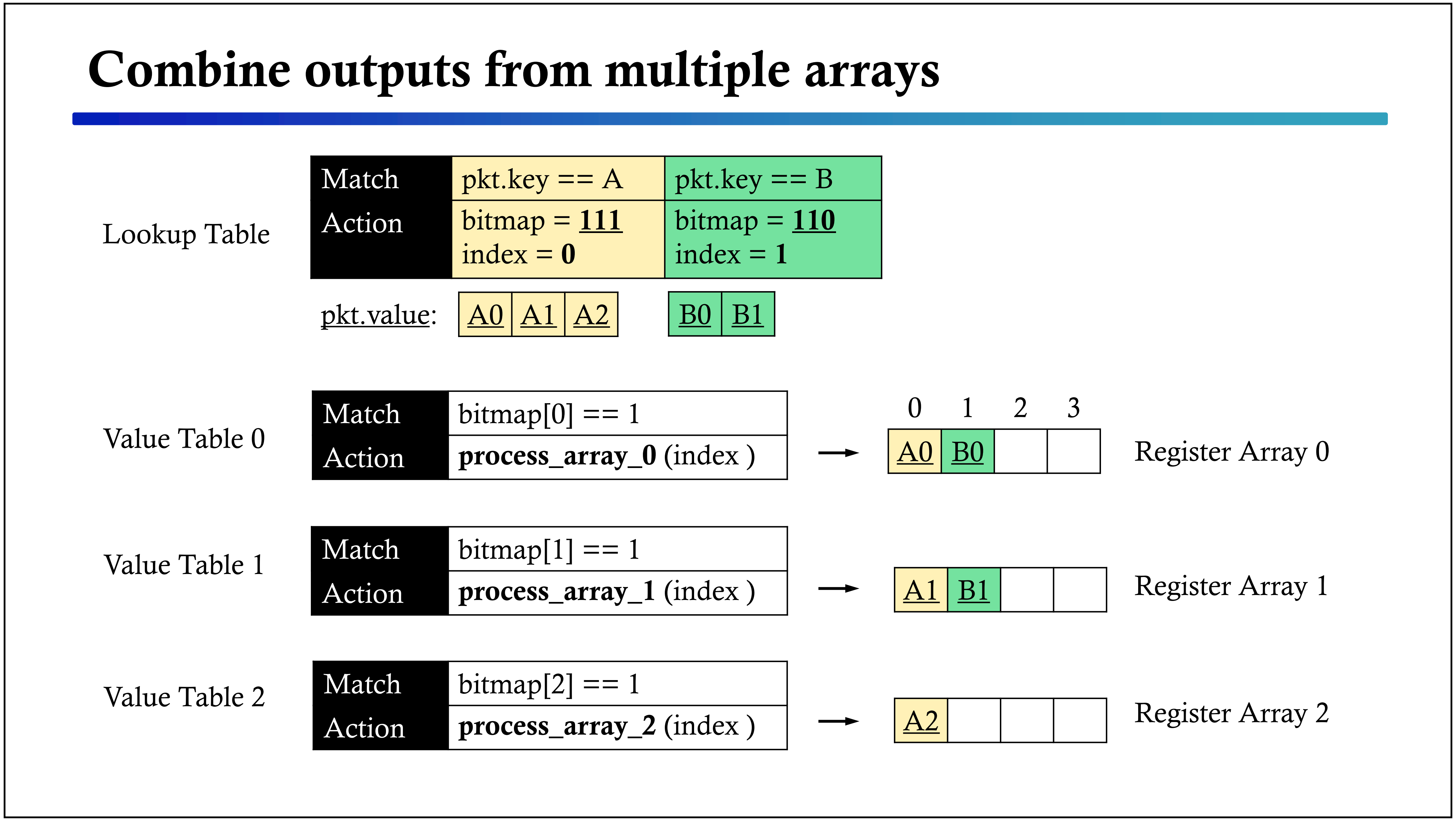
Value Table 1

Match	bitmap[1] == 1
Action	**process_array_1** (index)

Value Table 2

Match	bitmap[2] == 1
Action	**process_array_2** (index)

Source: NetCache: Balancing Key-Value Stores with Fast In-Network Caching, Xin Jin, 2017



Combine outputs from multiple arrays

Lookup Table

Match	pkt.key == A	pkt.key == B

Action	bitmap = **111**
	index = **0**
		bitmap = **110**
		index = **1**

pkt.value: | A0 | A1 | A2 | | B0 | B1 |

Value Table 0

Match	bitmap[0] == 1
Action	**process_array_0** (index)

→

0	1	2	3
A0	B0		

Register Array 0

Value Table 1

Match	bitmap[1] == 1
Action	**process_array_1** (index)

→

A1	B1		

Register Array 1

Value Table 2

Match	bitmap[2] == 1
Action	**process_array_2** (index)

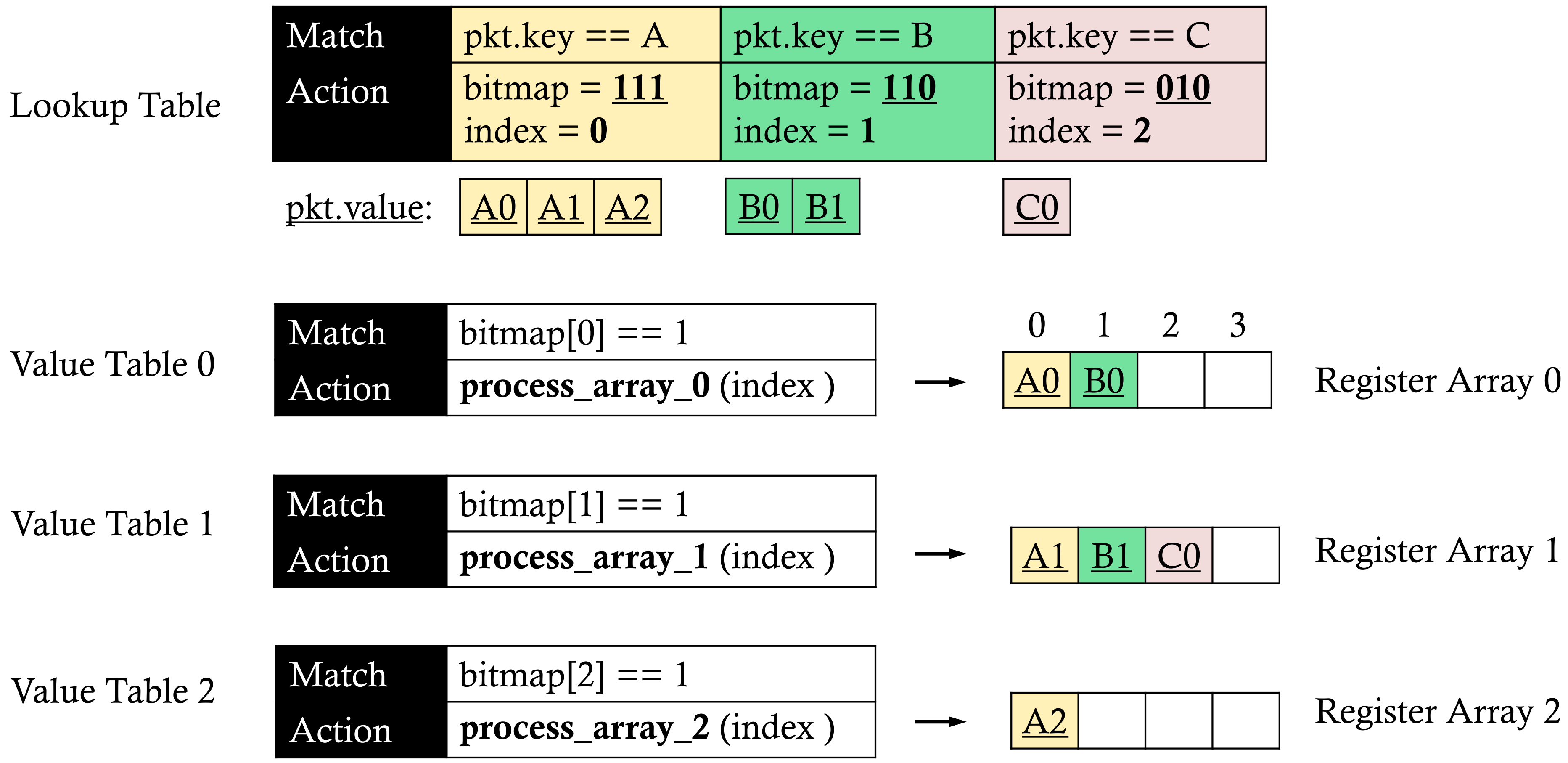
→

A2			

Register Array 2

Source: NetCache: Balancing Key-Value Stores with Fast In-Network Caching, Xin Jin, 2017

Combine outputs from multiple arrays



Lookup Table

Match	pkt.key == A	pkt.key == B	pkt.key == C

Action	bitmap = **111**
	index = **0**
		bitmap = **110**
		index = **1**
			bitmap = **010**
			index = **2**

pkt.value: A0 A1 A2 B0 B1 C0

Value Table 0

Match	bitmap[0] == 1
Action	**process_array_0** (index)

Value Table 1

Match	bitmap[1] == 1
Action	**process_array_1** (index)

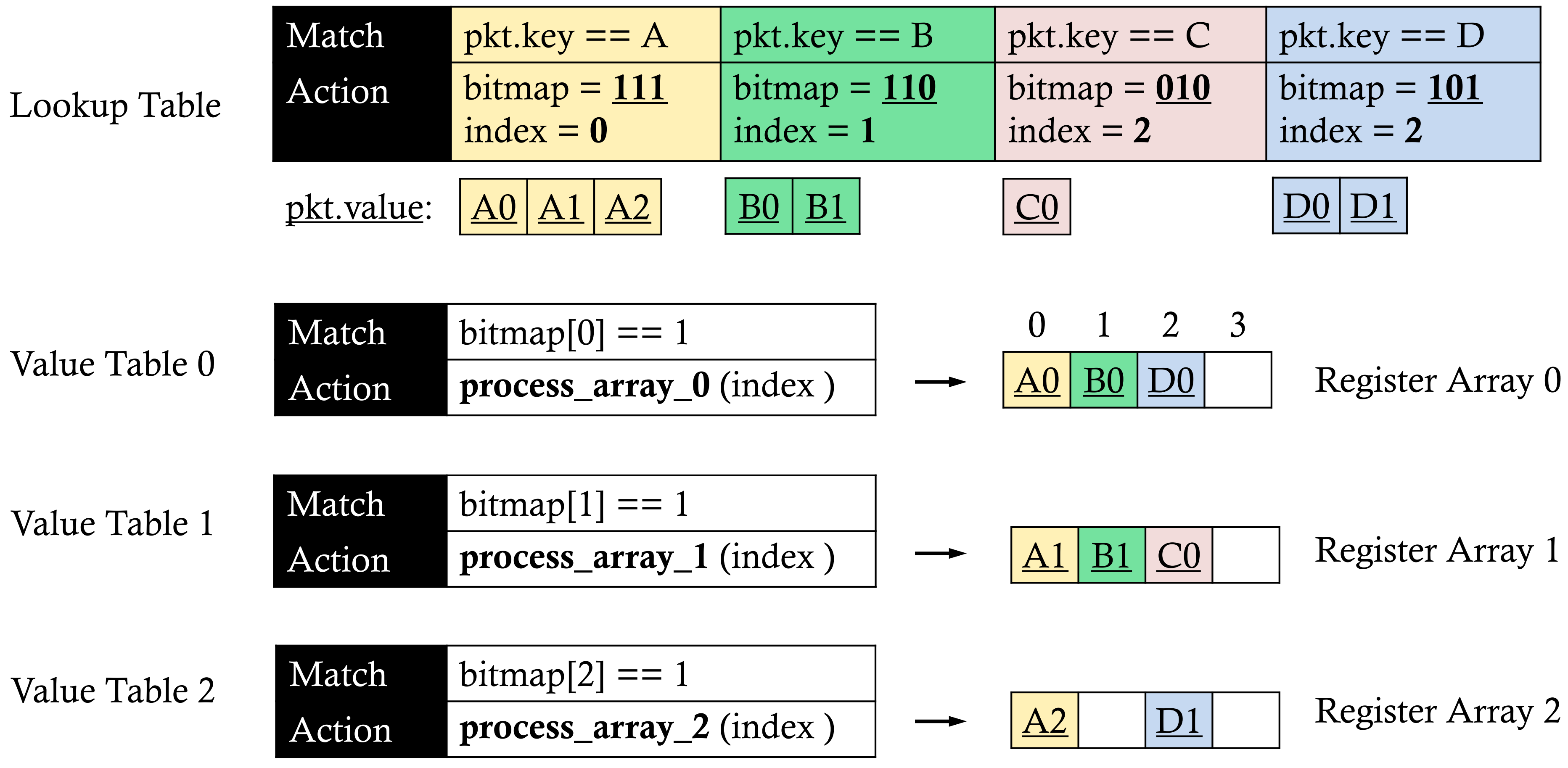
Value Table 2

Match	bitmap[2] == 1
Action	**process_array_2** (index)

	0	1	2	3
Register Array 0	A0	B0		
Register Array 1	A1	B1	C0	
Register Array 2	A2			

Source: NetCache: Balancing Key-Value Stores with Fast In-Network Caching, Xin Jin, 2017

Combine outputs from multiple arrays



Lookup Table

Match	pkt.key == A	pkt.key == B	pkt.key == C	pkt.key == D

Action	bitmap = **111**
	index = **0**
		bitmap = **110**
		index = **1**
			bitmap = **010**
			index = **2**
				bitmap = **101**
				index = **2**

pkt.value:	A0 A1 A2	B0 B1	C0	D0 D1

Value Table 0

Match	bitmap[0] == 1
Action	**process_array_0** (index)

→ Register Array 0

0	1	2	3
A0	B0	D0	

Value Table 1

Match	bitmap[1] == 1
Action	**process_array_1** (index)

→ Register Array 1

A1	B1	C0	

Value Table 2

Match	bitmap[2] == 1
Action	**process_array_2** (index)

→ Register Array 2

A2		D1	

Source: NetCache: Balancing Key-Value Stores with Fast In-Network Caching, Xin Jin, 2017

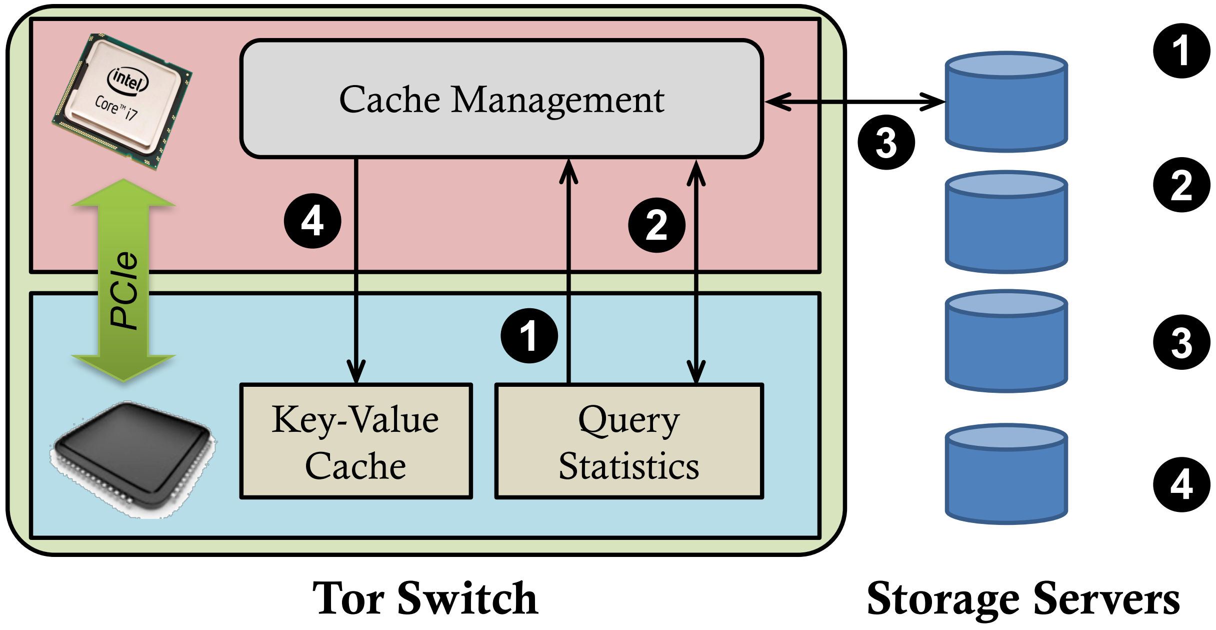
Key-value caching in network ASIC at line rate

- ❑ How to identify application-level packet fields ?
- ❑ How to store and serve variable-length data ?
- → ❑ How to efficiently keep the cache up-to-date ?

Source: NetCache: Balancing Key-Value Stores with Fast In-Network Caching, Xin Jin, 2017

Cache insertion and eviction

- Challenge: cache the hottest O($N \log N$) items with **limited insertion rate**
- Goal: react quickly and effectively to workload changes with **minimal updates**



1 Data plane reports hot keys

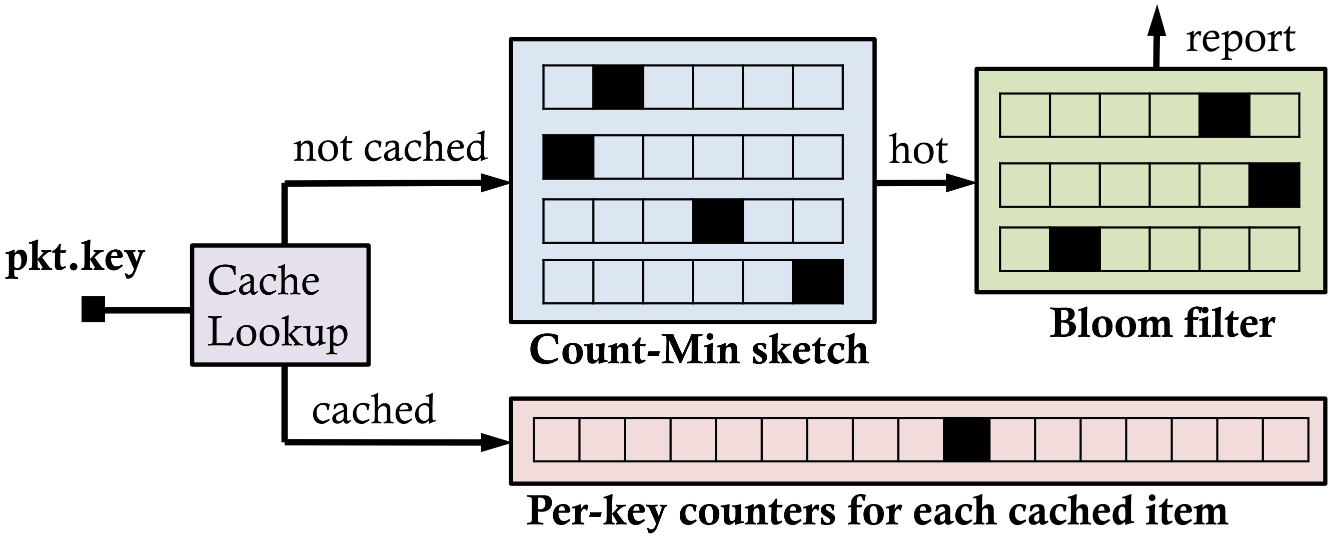
2 Control plane compares loads of new hot and sampled cached keys

3 Control plane fetches values for keys to be inserted to the cache

4 Control plane inserts and evicts keys

Source: NetCache: Balancing Key-Value Stores with Fast In-Network Caching, Xin Jin, 2017

Query statistics in the data plane



- ❑ Cached key: per-key counter array
- ❑ Uncached key
  - Count-Min sketch: report new hot keys
  - Bloom filter: remove duplicated hot key reports

Source: NetCache: Balancing Key-Value Stores with Fast In-Network Caching, Xin Jin, 2017

Data plane programmability

for

Performance

Monitoring

Applications offloading

Platforms

Correctness

Management

for

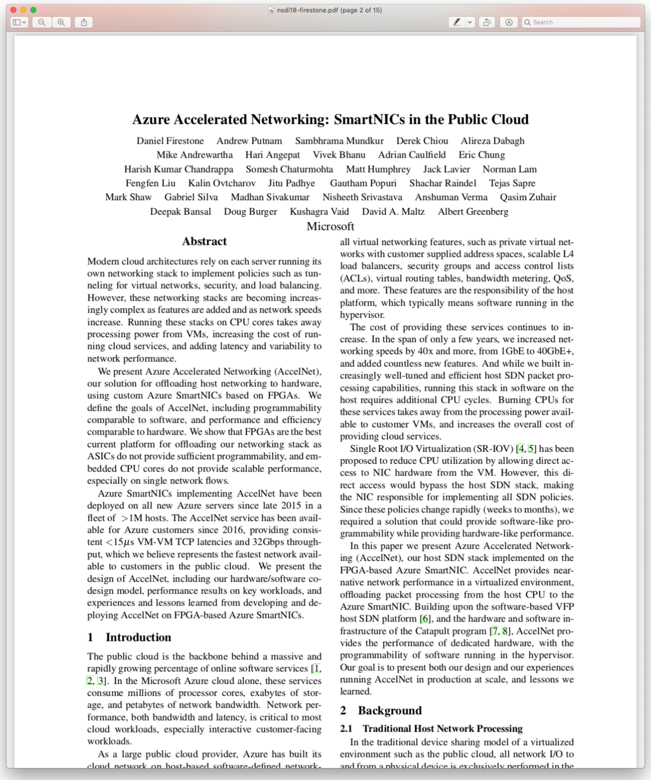
Data plane programmability

"Data-plane" programmability goes beyond switch programmability (or P4 for that matter)

Offloading…

… to FPGA-based SmartNICS

host networking



Azure Accelerated Networking: SmartNICs in the Public Cloud

Daniel Firestone Andrew Putnam Sambhrama Mundkur Derek Chiou Alireza Dabagh
Mike Andrewartha Hari Angepat Vivek Bhanu Adrian Caulfield Eric Chung
Harish Kumar Chandrappa Somesh Chaturmohta Matt Humphrey Jack Lavier Norman Lam
Fengfen Liu Kalin Ovtcharov Jitu Padhye Gautham Popuri Shachar Raindel Tejas Sapre
Mark Shaw Gabriel Silva Madhan Sivakumar Nisheeth Srivastava Anshuman Verma Qasim Zuhair
Deepak Bansal Doug Burger Kushagra Vaid David A. Maltz Albert Greenberg
Microsoft

Abstract

Modern cloud architectures rely on each server running its own networking stack to implement policies such as tunneling for virtual networks, security, and load balancing. However, these networking stacks are becoming increasingly complex as features are added and as network speeds increase. Running these stacks on CPU cores takes away processing power from VMs, increasing the cost of running cloud services, and adding latency and variability to network performance.

We present Azure Accelerated Networking (AccelNet), our solution for offloading host networking to hardware, using custom Azure SmartNICs based on FPGAs. We define the goals of AccelNet, including programmability comparable to software, and performance and efficiency comparable to hardware. We show that FPGAs are the best current platform for offloading our networking stack as ASICs do not provide sufficient programmability, and embedded CPU cores do not provide scalable performance, especially on single network flows.

Azure SmartNICs implementing AccelNet have been deployed on all new Azure servers since late 2015 in a fleet of >1M hosts. The AccelNet service has been available for Azure customers since 2016, providing consistent <15μs VM-VM TCP latencies and 32Gbps throughput, which we believe represents the fastest network available to customers in the public cloud. We present the design of AccelNet, including our hardware/software co-design model, performance results on key workloads, and experiences and lessons learned from developing and deploying AccelNet on FPGA-based Azure SmartNICs.

1 Introduction

The public cloud is the backbone behind a massive and rapidly growing percentage of online software services [1, 2, 3]. In the Microsoft Azure cloud alone, these services consume millions of processor cores, exabytes of storage, and petabytes of network bandwidth. Network performance, both bandwidth and latency, is critical to most cloud workloads, especially interactive customer-facing workloads.

As a large public cloud provider, Azure has built its cloud network on host-based software-defined network- all virtual networking features, such as private virtual networks with customer supplied address spaces, scalable L4 load balancers, security groups and access control lists (ACLs), virtual routing tables, bandwidth metering, QoS, and more. These features are the responsibility of the host platform, which typically means software running in the hypervisor.

The cost of providing these services continues to increase. In the span of only a few years, we increased networking speeds by 40x and more, from 1GbE to 40GbE+, and added countless new features. And while we built increasingly well-tuned and efficient host SDN packet processing capabilities, running this stack in software on the host requires additional CPU cycles. Burning CPUs for these services takes away from the processing power available to customer VMs, and increases the overall cost of providing cloud services.

Single Root I/O Virtualization (SR-IOV) [4, 5] has been proposed to reduce CPU utilization by allowing direct access to NIC hardware from the VM. However, this direct access would bypass the host SDN stack, making the NIC responsible for implementing all SDN policies. Since these policies change rapidly (weeks to months), we required a solution that could provide software-like programmability while providing hardware-like performance.

In this paper we present Azure Accelerated Networking (AccelNet), our host SDN stack implemented on the FPGA-based Azure SmartNIC. AccelNet provides near-native network performance in a virtualized environment, offloading packet processing from the host CPU to the Azure SmartNIC. Building upon the software-based VFP host SDN platform [6], and the hardware and software infrastructure of the Catapult program [7, 8], AccelNet provides the performance of dedicated hardware, with the programmability of software running in the hypervisor. Our goal is to present both our design and our experiences running AccelNet in production at scale, and lessons we learned.

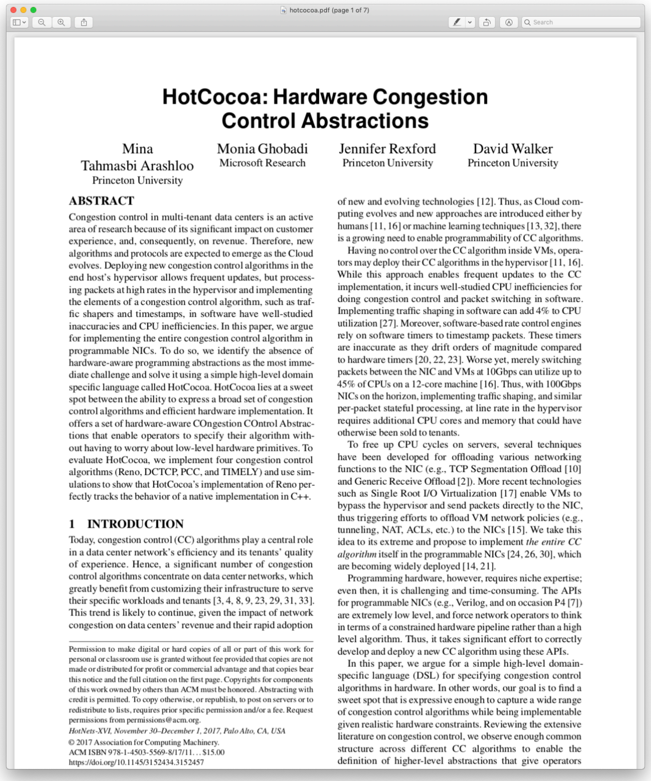
2 Background

2.1 Traditional Host Network Processing

In the traditional device sharing model of a virtualized environment such as the public cloud, all network I/O to and from a physical device is exclusively performed in the

[NSDI'18]

congestion control



HotCocoa: Hardware Congestion Control Abstractions

Mina Tahmasbi Arashloo, Princeton University
Monia Ghobadi, Microsoft Research
Jennifer Rexford, Princeton University
David Walker, Princeton University

ABSTRACT

Congestion control in multi-tenant data centers is an active area of research because of its significant impact on customer experience, and, consequently, on revenue. Therefore, new algorithms and protocols are expected to emerge as the Cloud evolves. Deploying new congestion control algorithms in the end host's hypervisor allows frequent updates, but processing packets at high rates in the hypervisor and implementing the elements of a congestion control algorithm, such as traffic shapers and timestamps, in software have well-studied inaccuracies and CPU inefficiencies. In this paper, we argue for implementing the entire congestion control algorithm in programmable NICs. To do so, we identify the absence of hardware-aware programming abstractions as the most immediate challenge and solve it using a simple high-level domain specific language called HotCocoa. HotCocoa lies at a sweet spot between the ability to express a broad set of congestion control algorithms and efficient hardware implementation. It offers a set of hardware-aware COngestion COntrol Abstractions that enable operators to specify their algorithm without having to worry about low-level hardware primitives. To evaluate HotCocoa, we implement four congestion control algorithms (Reno, DCTCP, PCC, and TIMELY) and use simulations to show that HotCocoa's implementation of Reno perfectly tracks the behavior of a native implementation in C++.

1 INTRODUCTION

Today, congestion control (CC) algorithms play a central role in a data center network's efficiency and its tenants' quality of experience. Hence, a significant number of congestion control algorithms concentrate on data center networks, which greatly benefit from customizing their infrastructure to serve their specific workloads and tenants [3, 4, 8, 9, 23, 29, 31, 33]. This trend is likely to continue, given the impact of network congestion on data centers' revenue and their rapid adoption of new and evolving technologies [12]. Thus, as Cloud computing evolves and new approaches are introduced either by humans [11, 16] or machine learning techniques [13, 32], there is a growing need to enable programmability of CC algorithms.

Having no control over the CC algorithm inside VMs, operators may deploy their CC algorithms in the hypervisor [11, 16]. While this approach enables frequent updates to the CC implementation, it incurs well-studied CPU inefficiencies for doing congestion control and packet switching in software. Implementing traffic shaping in software can add 4% to CPU utilization [27]. Moreover, software-based rate control engines rely on software timers to timestamp packets. These timers are inaccurate as they drift orders of magnitude compared to hardware timers [20, 22, 23]. Worse yet, merely switching packets between the NIC and VMs at 10Gbps can utilize up to 45% of CPUs on a 12-core machine [16]. Thus, with 100Gbps NICs on the horizon, implementing traffic shaping, and similar per-packet stateful processing, at line rate in the hypervisor requires additional CPU cores and memory that could have otherwise been sold to tenants.

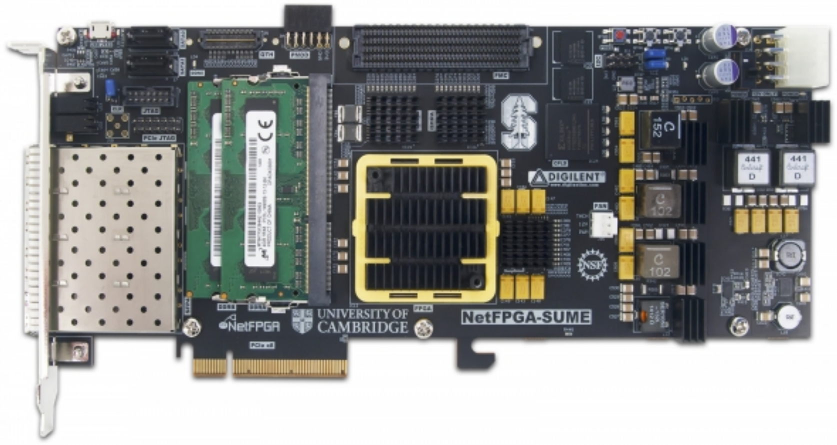
To free up CPU cycles on servers, several techniques have been developed for offloading various networking functions to the NIC (e.g., TCP Segmentation Offload [10] and Generic Receive Offload [2]). More recent technologies such as Single Root I/O Virtualization [17] enable VMs to bypass the hypervisor and send packets directly to the NIC, thus triggering efforts to offload VM network policies (e.g., tunneling, NAT, ACLs, etc.) to the NICs [15]. We take this idea to its extreme and propose to implement *the entire CC algorithm* itself in the programmable NICs [24, 26, 30], which are becoming widely deployed [14, 21].

Programming hardware, however, requires niche expertise; even then, it is challenging and time-consuming. The APIs for programmable NICs (e.g., Verilog, and on occasion P4 [7]) are extremely low level, and force network operators to think in terms of a constrained hardware pipeline rather than a high level algorithm. Thus, it takes significant effort to correctly develop and deploy a new CC algorithm using these APIs.

In this paper, we argue for a simple high-level domain-specific language (DSL) for specifying congestion control algorithms in hardware. In other words, our goal is to find a sweet spot that is expressive enough to capture a wide range of congestion control algorithms while being implementable given realistic hardware constraints. Reviewing the extensive literature on congestion control, we observe enough common structure across different CC algorithms to enable the definition of higher-level abstractions that give operators

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HotNets-XVI, November 30–December 1, 2017, Palo Alto, CA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5569-8/17/11. . . $15.00
https://doi.org/10.1145/3152434.3152457

[HotNets'17]

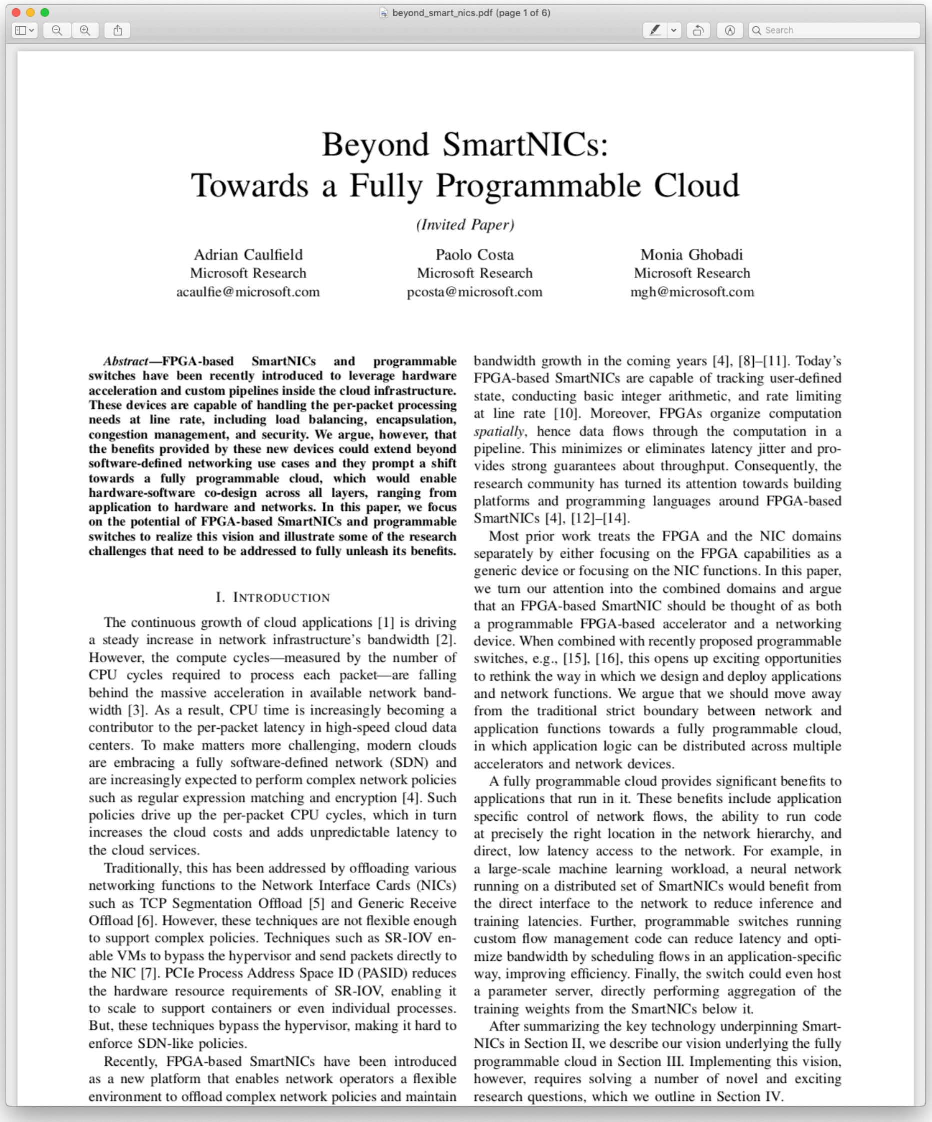


NetFPGA SUME board

Host-based programmability + SmartNICs + programmable switches = fully programmable platforms

Big question is

How to combine them best?



Beyond SmartNICs: Towards a Fully Programmable Cloud

(Invited Paper)

Adrian Caulfield
Microsoft Research
acaulfie@microsoft.com

Paolo Costa
Microsoft Research
pcosta@microsoft.com

Monia Ghobadi
Microsoft Research
mgh@microsoft.com

***Abstract*—FPGA-based SmartNICs and programmable switches have been recently introduced to leverage hardware acceleration and custom pipelines inside the cloud infrastructure. These devices are capable of handling the per-packet processing needs at line rate, including load balancing, encapsulation, congestion management, and security. We argue, however, that the benefits provided by these new devices could extend beyond software-defined networking use cases and they prompt a shift towards a fully programmable cloud, which would enable hardware-software co-design across all layers, ranging from application to hardware and networks. In this paper, we focus on the potential of FPGA-based SmartNICs and programmable switches to realize this vision and illustrate some of the research challenges that need to be addressed to fully unleash its benefits.**

I. Introduction

The continuous growth of cloud applications [1] is driving a steady increase in network infrastructure's bandwidth [2]. However, the compute cycles—measured by the number of CPU cycles required to process each packet—are falling behind the massive acceleration in available network bandwidth [3]. As a result, CPU time is increasingly becoming a contributor to the per-packet latency in high-speed cloud data centers. To make matters more challenging, modern clouds are embracing a fully software-defined network (SDN) and are increasingly expected to perform complex network policies such as regular expression matching and encryption [4]. Such policies drive up the per-packet CPU cycles, which in turn increases the cloud costs and adds unpredictable latency to the cloud services.

Traditionally, this has been addressed by offloading various networking functions to the Network Interface Cards (NICs) such as TCP Segmentation Offload [5] and Generic Receive Offload [6]. However, these techniques are not flexible enough to support complex policies. Techniques such as SR-IOV enable VMs to bypass the hypervisor and send packets directly to the NIC [7]. PCIe Process Address Space ID (PASID) reduces the hardware resource requirements of SR-IOV, enabling it to scale to support containers or even individual processes. But, these techniques bypass the hypervisor, making it hard to enforce SDN-like policies.

Recently, FPGA-based SmartNICs have been introduced as a new platform that enables network operators a flexible environment to offload complex network policies and maintain bandwidth growth in the coming years [4], [8]–[11]. Today's FPGA-based SmartNICs are capable of tracking user-defined state, conducting basic integer arithmetic, and rate limiting at line rate [10]. Moreover, FPGAs organize computation *spatially*, hence data flows through the computation in a pipeline. This minimizes or eliminates latency jitter and provides strong guarantees about throughput. Consequently, the research community has turned its attention towards building platforms and programming languages around FPGA-based SmartNICs [4], [12]–[14].

Most prior work treats the FPGA and the NIC domains separately by either focusing on the FPGA capabilities as a generic device or focusing on the NIC functions. In this paper, we turn our attention into the combined domains and argue that an FPGA-based SmartNIC should be thought of as both a programmable FPGA-based accelerator and a networking device. When combined with recently proposed programmable switches, e.g., [15], [16], this opens up exciting opportunities to rethink the way in which we design and deploy applications and network functions. We argue that we should move away from the traditional strict boundary between network and application functions towards a fully programmable cloud, in which application logic can be distributed across multiple accelerators and network devices.

A fully programmable cloud provides significant benefits to applications that run in it. These benefits include application specific control of network flows, the ability to run code at precisely the right location in the network hierarchy, and direct, low latency access to the network. For example, in a large-scale machine learning workload, a neural network running on a distributed set of SmartNICs would benefit from the direct interface to the network to reduce inference and training latencies. Further, programmable switches running custom flow management code can reduce latency and optimize bandwidth by scheduling flows in an application-specific way, improving efficiency. Finally, the switch could even host a parameter server, directly performing aggregation of the training weights from the SmartNICs below it.

After summarizing the key technology underpinning SmartNICs in Section II, we describe our vision underlying the fully programmable cloud in Section III. Implementing this vision, however, requires solving a number of novel and exciting research questions, which we outline in Section IV.

IEEE International Conference on
High Performance Switching and Routing, 2018

Data plane programmability for

Performance

Monitoring

Applications offloading

Platforms

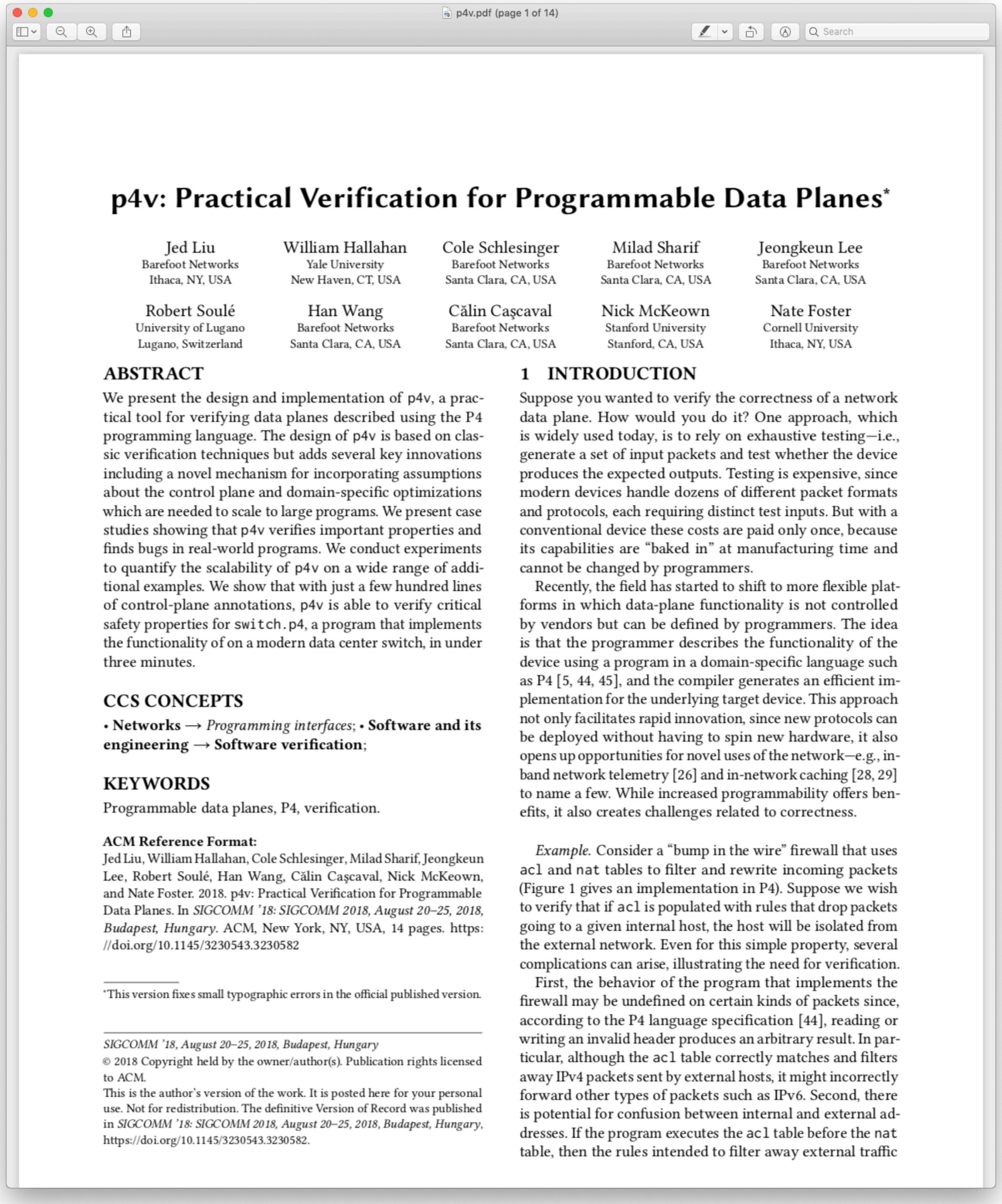
Correctness

Management

for Data plane programmability

So you've a programmable networks…

How do you make sure that it works as it should?!



p4v: Practical Verification for Programmable Data Planes*

Jed Liu, Barefoot Networks, Ithaca, NY, USA
William Hallahan, Yale University, New Haven, CT, USA
Cole Schlesinger, Barefoot Networks, Santa Clara, CA, USA
Milad Sharif, Barefoot Networks, Santa Clara, CA, USA
Jeongkeun Lee, Barefoot Networks, Santa Clara, CA, USA
Robert Soulé, University of Lugano, Lugano, Switzerland
Han Wang, Barefoot Networks, Santa Clara, CA, USA
Călin Caşcaval, Barefoot Networks, Santa Clara, CA, USA
Nick McKeown, Stanford University, Stanford, CA, USA
Nate Foster, Cornell University, Ithaca, NY, USA

ABSTRACT

We present the design and implementation of p4v, a practical tool for verifying data planes described using the P4 programming language. The design of p4v is based on classic verification techniques but adds several key innovations including a novel mechanism for incorporating assumptions about the control plane and domain-specific optimizations which are needed to scale to large programs. We present case studies showing that p4v verifies important properties and finds bugs in real-world programs. We conduct experiments to quantify the scalability of p4v on a wide range of additional examples. We show that with just a few hundred lines of control-plane annotations, p4v is able to verify critical safety properties for switch.p4, a program that implements the functionality of on a modern data center switch, in under three minutes.

CCS CONCEPTS

• **Networks** → *Programming interfaces*; • **Software and its engineering** → **Software verification**;

KEYWORDS

Programmable data planes, P4, verification.

ACM Reference Format:
Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. p4v: Practical Verification for Programmable Data Planes. In *SIGCOMM '18: SIGCOMM 2018, August 20–25, 2018, Budapest, Hungary*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3230543.3230582

1 INTRODUCTION

Suppose you wanted to verify the correctness of a network data plane. How would you do it? One approach, which is widely used today, is to rely on exhaustive testing—i.e., generate a set of input packets and test whether the device produces the expected outputs. Testing is expensive, since modern devices handle dozens of different packet formats and protocols, each requiring distinct test inputs. But with a conventional device these costs are paid only once, because its capabilities are "baked in" at manufacturing time and cannot be changed by programmers.

Recently, the field has started to shift to more flexible platforms in which data-plane functionality is not controlled by vendors but can be defined by programmers. The idea is that the programmer describes the functionality of the device using a program in a domain-specific language such as P4 [5, 44, 45], and the compiler generates an efficient implementation for the underlying target device. This approach not only facilitates rapid innovation, since new protocols can be deployed without having to spin new hardware, it also opens up opportunities for novel uses of the network—e.g., in-band network telemetry [26] and in-network caching [28, 29] to name a few. While increased programmability offers benefits, it also creates challenges related to correctness.

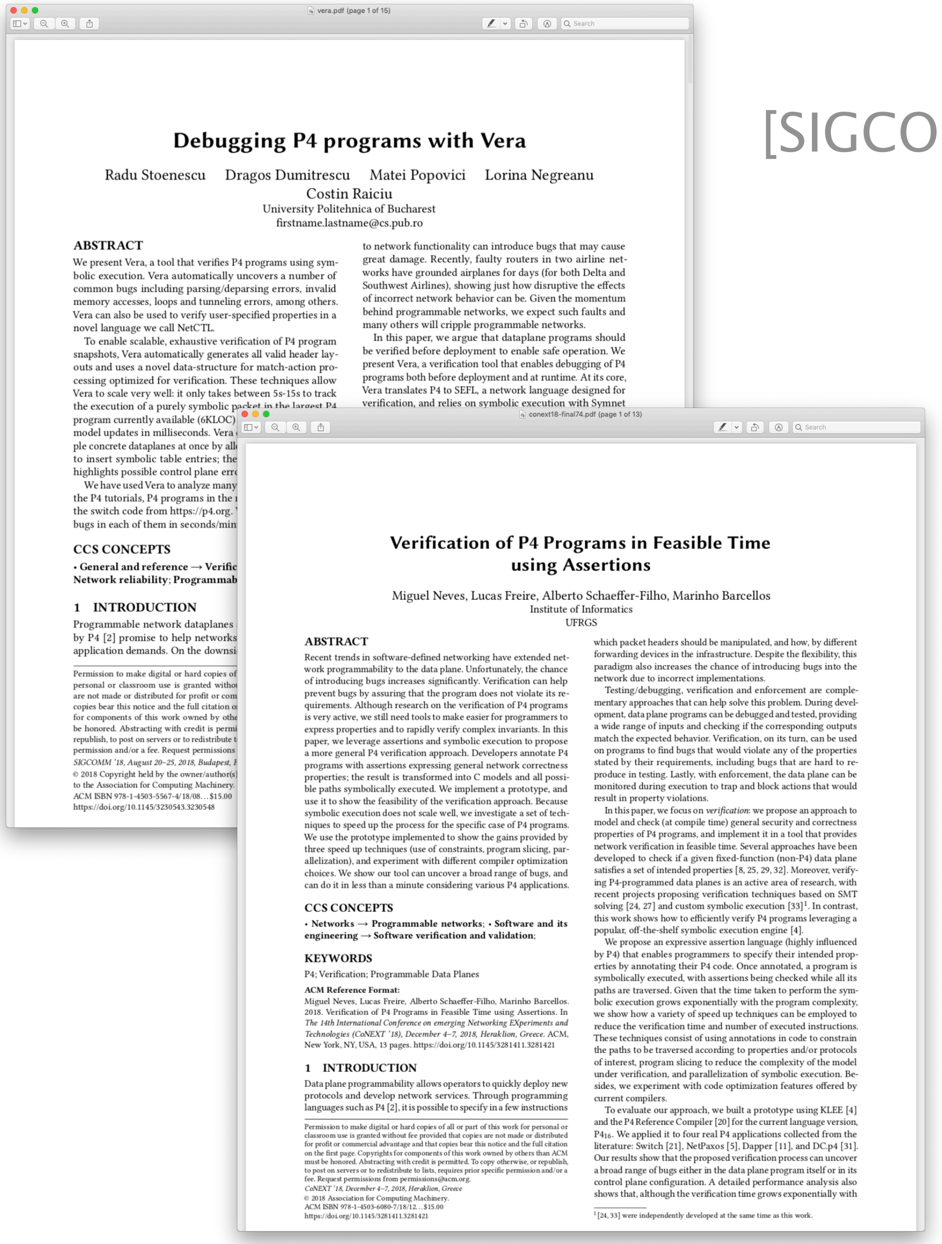
Example. Consider a "bump in the wire" firewall that uses acl and nat tables to filter and rewrite incoming packets (Figure 1 gives an implementation in P4). Suppose we wish to verify that if acl is populated with rules that drop packets going to a given internal host, the host will be isolated from the external network. Even for this simple property, several complications can arise, illustrating the need for verification.

First, the behavior of the program that implements the firewall may be undefined on certain kinds of packets since, according to the P4 language specification [44], reading or writing an invalid header produces an arbitrary result. In particular, although the acl table correctly matches and filters away IPv4 packets sent by external hosts, it might incorrectly forward other types of packets such as IPv6. Second, there is potential for confusion between internal and external addresses. If the program executes the acl table before the nat table, then the rules intended to filter away external traffic

*This version fixes small typographic errors in the official published version.

SIGCOMM '18, August 20–25, 2018, Budapest, Hungary

[SIGCOMM'18]



Debugging P4 programs with Vera

Radu Stoenescu Dragos Dumitrescu Matei Popovici Lorina Negreanu
Costin Raiciu
University Politehnica of Bucharest
firstname.lastname@cs.pub.ro

ABSTRACT

We present Vera, a tool that verifies P4 programs using symbolic execution. Vera automatically uncovers a number of common bugs including parsing/deparsing errors, invalid memory accesses, loops and tunneling errors, among others. Vera can also be used to verify user-specified properties in a novel language we call NetCTL.

To enable scalable, exhaustive verification of P4 program snapshots, Vera automatically generates all valid header layouts and uses a novel data-structure for match-action processing optimized for verification. These techniques allow Vera to scale very well: it only takes between 5s-15s to track the execution of a purely symbolic packet in the largest P4
program currently available (6KLOC)
model updates in milliseconds. Vera
ple concrete dataplanes at once by all
to insert symbolic table entries; the
highlights possible control plane err

We have used Vera to analyze many
the P4 tutorials, P4 programs in the
the switch code from https://p4.org.
bugs in each of them in seconds/min

CCS CONCEPTS

• **General and reference** → **Verific
Network reliability**; **Programmab**

1 INTRODUCTION

Programmable network dataplanes
by P4 [2] promise to help networks
application demands. On the downsi

SIGCOMM '18, August 20–25, 2018, Budapest, H

ACM ISBN 978-1-4503-5567-4/18/08…$15.00
https://doi.org/10.1145/3230543.3230548

to network functionality can introduce bugs that may cause great damage. Recently, faulty routers in two airline networks have grounded airplanes for days (for both Delta and Southwest Airlines), showing just how disruptive the effects of incorrect network behavior can be. Given the momentum behind programmable networks, we expect such faults and many others will cripple programmable networks.

In this paper, we argue that dataplane programs should be verified before deployment to enable safe operation. We present Vera, a verification tool that enables debugging of P4 programs both before deployment and at runtime. At its core, Vera translates P4 to SEFL, a network language designed for verification, and relies on symbolic execution with Symnet

[SIGCOMM'18]

Verification of P4 Programs in Feasible Time using Assertions

Miguel Neves, Lucas Freire, Alberto Schaeffer-Filho, Marinho Barcellos
Institute of Informatics
UFRGS

ABSTRACT

Recent trends in software-defined networking have extended network programmability to the data plane. Unfortunately, the chance of introducing bugs increases significantly. Verification can help prevent bugs by assuring that the program does not violate its requirements. Although research on the verification of P4 programs is very active, we still need tools to make easier for programmers to express properties and to rapidly verify complex invariants. In this paper, we leverage assertions and symbolic execution to propose a more general P4 verification approach. Developers annotate P4 programs with assertions expressing general network correctness properties; the result is transformed into C models and all possible paths symbolically executed. We implement a prototype, and use it to show the feasibility of the verification approach. Because symbolic execution does not scale well, we investigate a set of techniques to speed up the process for the specific case of P4 programs. We use the prototype implemented to show the gains provided by three speed up techniques (use of constraints, program slicing, parallelization), and experiment with different compiler optimization choices. We show our tool can uncover a broad range of bugs, and can do it in less than a minute considering various P4 applications.

CCS CONCEPTS

• **Networks** → **Programmable networks**; • **Software and its engineering** → **Software verification and validation**;

KEYWORDS

P4; Verification; Programmable Data Planes

ACM Reference Format:
Miguel Neves, Lucas Freire, Alberto Schaeffer-Filho, Marinho Barcellos. 2018. Verification of P4 Programs in Feasible Time using Assertions. In *The 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '18), December 4–7, 2018, Heraklion, Greece*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3281411.3281421

1 INTRODUCTION

Data plane programmability allows operators to quickly deploy new protocols and develop network services. Through programming languages such as P4 [2], it is possible to specify in a few instructions which packet headers should be manipulated, and how, by different forwarding devices in the infrastructure. Despite the flexibility, this paradigm also increases the chance of introducing bugs into the network due to incorrect implementations.

Testing/debugging, verification and enforcement are complementary approaches that can help solve this problem. During development, data plane programs can be debugged and tested, providing a wide range of inputs and checking if the corresponding outputs match the expected behavior. Verification, on its turn, can be used on programs to find bugs that would violate any of the properties stated by their requirements, including bugs that are hard to reproduce in testing. Lastly, with enforcement, the data plane can be monitored during execution to trap and block actions that would result in property violations.

In this paper, we focus on *verification*: we propose an approach to model and check (at compile time) general security and correctness properties of P4 programs, and implement it in a tool that provides network verification in feasible time. Several approaches have been developed to check if a given fixed-function (non-P4) data plane satisfies a set of intended properties [8, 25, 29, 32]. Moreover, verifying P4-programmed data planes is an active area of research, with recent projects proposing verification techniques based on SMT solving [24, 27] and custom symbolic execution [33][1]. In contrast, this work shows how to efficiently verify P4 programs leveraging a popular, off-the-shelf symbolic execution engine [4].

We propose an expressive assertion language (highly influenced by P4) that enables programmers to specify their intended properties by annotating their P4 code. Once annotated, a program is symbolically executed, with assertions being checked while all its paths are traversed. Given that the time taken to perform the symbolic execution grows exponentially with the program complexity, we show how a variety of speed up techniques can be employed to reduce the verification time and number of executed instructions. These techniques consist of using annotations in code to constrain the paths to be traversed according to properties and/or protocols of interest, program slicing to reduce the complexity of the model under verification, and parallelization of symbolic execution. Besides, we experiment with code optimization features offered by current compilers.

To evaluate our approach, we built a prototype using KLEE [4] and the P4 Reference Compiler [20] for the current language version, $P4_{16}$. We applied it to four real P4 applications collected from the literature: Switch [21], NetPaxos [5], Dapper [11], and DC.p4 [31]. Our results show that the proposed verification process can uncover a broad range of bugs either in the data plane program itself or in its control plane configuration. A detailed performance analysis also shows that, although the verification time grows exponentially with

CoNEXT '18, December 4–7, 2018, Heraklion, Greece

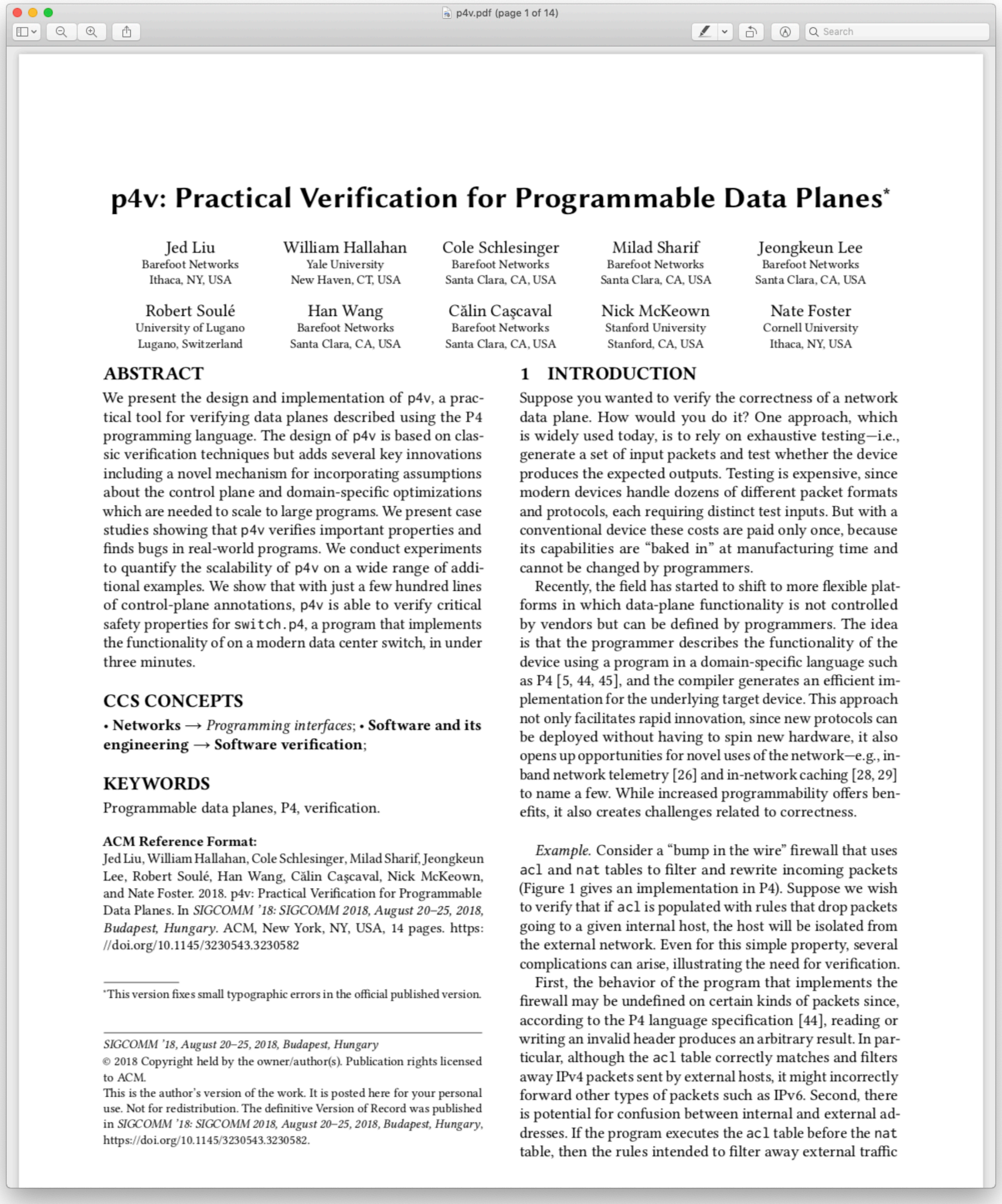
ACM ISBN 978-1-4503-6080-7/18/12…$15.00
https://doi.org/10.1145/3281411.3281421

[1] [24, 33] were independently developed at the same time as this work.

[CoNEXT'18]

So you've a programmable networks…

How do you make sure that it works as it should?!



p4v: Practical Verification for Programmable Data Planes*

Jed Liu, Barefoot Networks, Ithaca, NY, USA
William Hallahan, Yale University, New Haven, CT, USA
Cole Schlesinger, Barefoot Networks, Santa Clara, CA, USA
Milad Sharif, Barefoot Networks, Santa Clara, CA, USA
Jeongkeun Lee, Barefoot Networks, Santa Clara, CA, USA
Robert Soulé, University of Lugano, Lugano, Switzerland
Han Wang, Barefoot Networks, Santa Clara, CA, USA
Călin Caşcaval, Barefoot Networks, Santa Clara, CA, USA
Nick McKeown, Stanford University, Stanford, CA, USA
Nate Foster, Cornell University, Ithaca, NY, USA

ABSTRACT

We present the design and implementation of p4v, a practical tool for verifying data planes described using the P4 programming language. The design of p4v is based on classic verification techniques but adds several key innovations including a novel mechanism for incorporating assumptions about the control plane and domain-specific optimizations which are needed to scale to large programs. We present case studies showing that p4v verifies important properties and finds bugs in real-world programs. We conduct experiments to quantify the scalability of p4v on a wide range of additional examples. We show that with just a few hundred lines of control-plane annotations, p4v is able to verify critical safety properties for switch.p4, a program that implements the functionality of on a modern data center switch, in under three minutes.

CCS CONCEPTS

• **Networks** → *Programming interfaces*; • **Software and its engineering** → **Software verification**;

KEYWORDS

Programmable data planes, P4, verification.

ACM Reference Format:
Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. p4v: Practical Verification for Programmable Data Planes. In *SIGCOMM '18: SIGCOMM 2018, August 20–25, 2018, Budapest, Hungary.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3230543.3230582

1 INTRODUCTION

Suppose you wanted to verify the correctness of a network data plane. How would you do it? One approach, which is widely used today, is to rely on exhaustive testing—i.e., generate a set of input packets and test whether the device produces the expected outputs. Testing is expensive, since modern devices handle dozens of different packet formats and protocols, each requiring distinct test inputs. But with a conventional device these costs are paid only once, because its capabilities are "baked in" at manufacturing time and cannot be changed by programmers.

Recently, the field has started to shift to more flexible platforms in which data-plane functionality is not controlled by vendors but can be defined by programmers. The idea is that the programmer describes the functionality of the device using a program in a domain-specific language such as P4 [5, 44, 45], and the compiler generates an efficient implementation for the underlying target device. This approach not only facilitates rapid innovation, since new protocols can be deployed without having to spin new hardware, it also opens up opportunities for novel uses of the network—e.g., in-band network telemetry [26] and in-network caching [28, 29] to name a few. While increased programmability offers benefits, it also creates challenges related to correctness.

Example. Consider a "bump in the wire" firewall that uses acl and nat tables to filter and rewrite incoming packets (Figure 1 gives an implementation in P4). Suppose we wish to verify that if acl is populated with rules that drop packets going to a given internal host, the host will be isolated from the external network. Even for this simple property, several complications can arise, illustrating the need for verification.

First, the behavior of the program that implements the firewall may be undefined on certain kinds of packets since, according to the P4 language specification [44], reading or writing an invalid header produces an arbitrary result. In particular, although the acl table correctly matches and filters away IPv4 packets sent by external hosts, it might incorrectly forward other types of packets such as IPv6. Second, there is potential for confusion between internal and external addresses. If the program executes the acl table before the nat table, then the rules intended to filter away external traffic

*This version fixes small typographic errors in the official published version.

SIGCOMM '18, August 20–25, 2018, Budapest, Hungary
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *SIGCOMM '18: SIGCOMM 2018, August 20–25, 2018, Budapest, Hungary*, https://doi.org/10.1145/3230543.3230582.

[SIGCOMM'18]

Debugging P4 programs with Vera

Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, Costin Raiciu
University Politehnica of Bucharest
firstname.lastname@cs.pub.ro

ABSTRACT

We present Vera, a tool that verifies P4 programs using symbolic execution. Vera automatically uncovers a number of common bugs including parsing/deparsing errors, invalid memory accesses, loops and tunneling errors, among others. Vera can also be used to verify user-specified properties in a novel language we call NetCTL.

To enable scalable, exhaustive verification of P4 program snapshots, Vera automatically generates all valid header layouts and uses a novel data-structure for match-action processing optimized for verification. These techniques allow Vera to scale very well: it only takes between 5s-15s to track the execution of a purely symbolic packet in the largest P4 program currently available (6KLOC)

model updates in milliseconds. Vera

ple concrete dataplanes at once by all

to insert symbolic table entries; the

highlights possible control plane err

We have used Vera to analyze many

the P4 tutorials, P4 programs in the

the switch code from https://p4.org.

bugs in each of them in seconds/min

CCS CONCEPTS

• **General and reference** → **Verific**

Network reliability; **Programmab**

1 INTRODUCTION

Programmable network dataplanes

by P4 [2] promise to help networks

application demands. On the downsi

Permission to make digital or hard copies of

personal or classroom use is granted witho

are not made or distributed for profit or com

copies bear this notice and the full citation o

for components of this work owned by othe

be honored. Abstracting with credit is permi

republish, to post on servers or to redistribute t

permission and/or a fee. Request permissions

SIGCOMM '18, August 20–25, 2018, Budapest, H

© 2018 Copyright held by the owner/author(s)

to the Association for Computing Machinery.

ACM ISBN 978-1-4503-5567-4/18/08…$15.00

https://doi.org/10.1145/3230543.3230548

to network functionality can introduce bugs that may cause great damage. Recently, faulty routers in two airline networks have grounded airplanes for days (for both Delta and Southwest Airlines), showing just how disruptive the effects of incorrect network behavior can be. Given the momentum behind programmable networks, we expect such faults and many others will cripple programmable networks.

In this paper, we argue that dataplane programs should be verified before deployment to enable safe operation. We present Vera, a verification tool that enables debugging of P4 programs both before deployment and at runtime. At its core, Vera translates P4 to SEFL, a network language designed for verification, and relies on symbolic execution with Symnet

[SIGCOMM'18]

Verification of P4 Programs in Feasible Time using Assertions

Miguel Neves, Lucas Freire, Alberto Schaeffer-Filho, Marinho Barcellos
Institute of Informatics
UFRGS

ABSTRACT

Recent trends in software-defined networking have extended network programmability to the data plane. Unfortunately, the chance of introducing bugs increases significantly. Verification can help prevent bugs by assuring that the program does not violate its requirements. Although research on the verification of P4 programs is very active, we still need tools to make easier for programmers to express properties and to rapidly verify complex invariants. In this paper, we leverage assertions and symbolic execution to propose a more general P4 verification approach. Developers annotate P4 programs with assertions expressing general network correctness properties; the result is transformed into C models and all possible paths symbolically executed. We implement a prototype, and use it to show the feasibility of the verification approach. Because symbolic execution does not scale well, we investigate a set of techniques to speed up the process for the specific case of P4 programs. We use the prototype implemented to show the gains provided by three speed up techniques (use of constraints, program slicing, parallelization), and experiment with different compiler optimization choices. We show our tool can uncover a broad range of bugs, and can do it in less than a minute considering various P4 applications.

CCS CONCEPTS

• **Networks** → **Programmable networks**; • **Software and its engineering** → **Software verification and validation**;

KEYWORDS

P4; Verification; Programmable Data Planes

ACM Reference Format:
Miguel Neves, Lucas Freire, Alberto Schaeffer-Filho, Marinho Barcellos. 2018. Verification of P4 Programs in Feasible Time using Assertions. In *The 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '18), December 4–7, 2018, Heraklion, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3281411.3281421

1 INTRODUCTION

Data plane programmability allows operators to quickly deploy new protocols and develop network services. Through programming languages such as P4 [2], it is possible to specify in a few instructions which packet headers should be manipulated, and how, by different forwarding devices in the infrastructure. Despite the flexibility, this paradigm also increases the chance of introducing bugs into the network due to incorrect implementations.

Testing/debugging, verification and enforcement are complementary approaches that can help solve this problem. During development, data plane programs can be debugged and tested, providing a wide range of inputs and checking if the corresponding outputs match the expected behavior. Verification, on its turn, can be used on programs to find bugs that would violate any of the properties stated by their requirements, including bugs that are hard to reproduce in testing. Lastly, with enforcement, the data plane can be monitored during execution to trap and block actions that would result in property violations.

In this paper, we focus on *verification*: we propose an approach to model and check (at compile time) general security and correctness properties of P4 programs, and implement it in a tool that provides network verification in feasible time. Several approaches have been developed to check if a given fixed-function (non-P4) data plane satisfies a set of intended properties [8, 25, 29, 32]. Moreover, verifying P4-programmed data planes is an active area of research, with recent projects proposing verification techniques based on SMT solving [24, 27] and custom symbolic execution [33][1]. In contrast, this work shows how to efficiently verify P4 programs leveraging a popular, off-the-shelf symbolic execution engine [4].

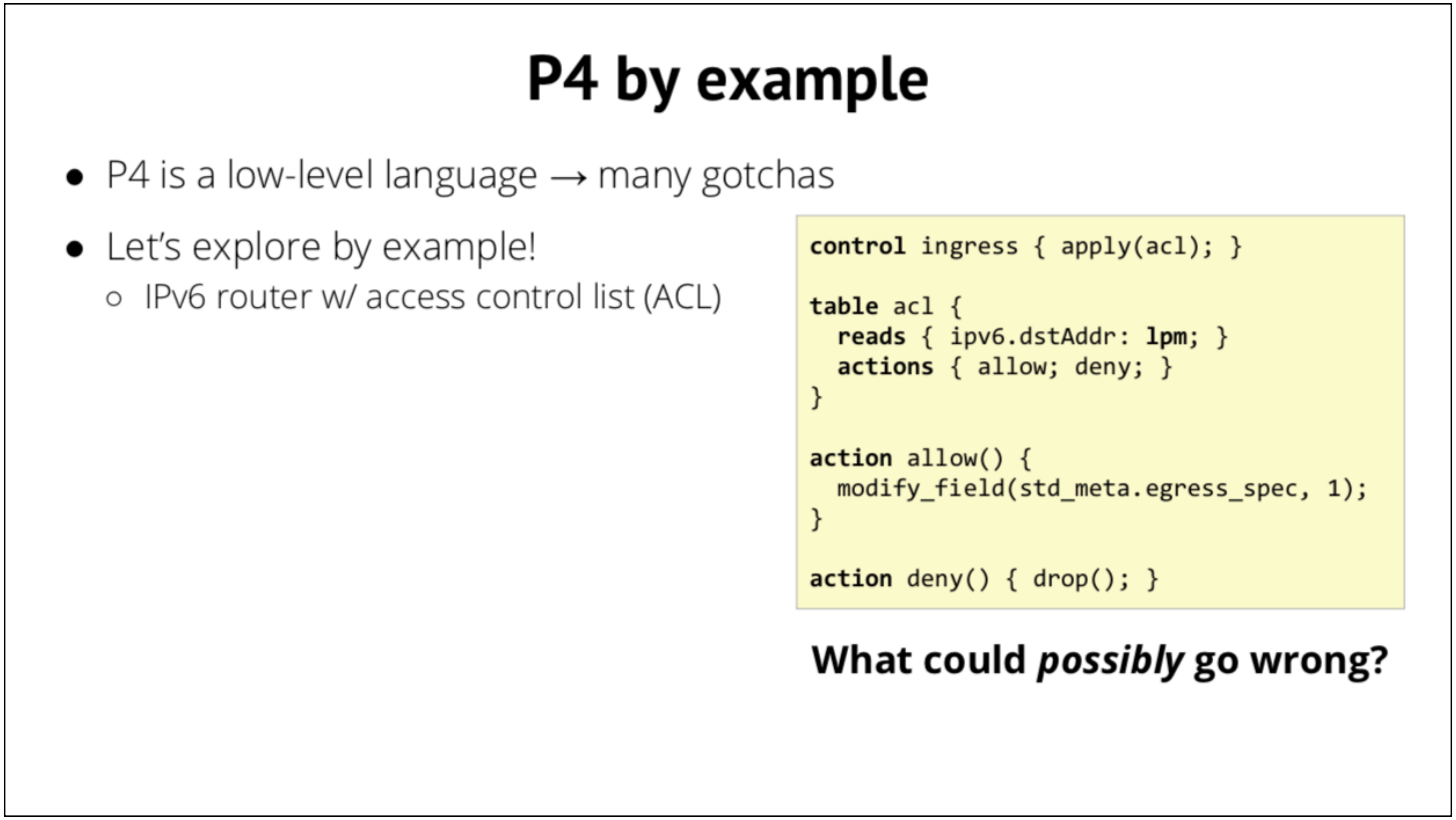
We propose an expressive assertion language (highly influenced by P4) that enables programmers to specify their intended properties by annotating their P4 code. Once annotated, a program is symbolically executed, with assertions being checked while all its paths are traversed. Given that the time taken to perform the symbolic execution grows exponentially with the program complexity, we show how a variety of speed up techniques can be employed to reduce the verification time and number of executed instructions. These techniques consist of using annotations in code to constrain the paths to be traversed according to properties and/or protocols of interest, program slicing to reduce the complexity of the model under verification, and parallelization of symbolic execution. Besides, we experiment with code optimization features offered by current compilers.

To evaluate our approach, we built a prototype using KLEE [4] and the P4 Reference Compiler [20] for the current language version, $P4_{16}$. We applied it to four real P4 applications collected from the literature: Switch [21], NetPaxos [5], Dapper [11], and DC.p4 [31]. Our results show that the proposed verification process can uncover a broad range of bugs either in the data plane program itself or in its control plane configuration. A detailed performance analysis also shows that, although the verification time grows exponentially with

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CoNEXT '18, December 4–7, 2018, Heraklion, Greece
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6080-7/18/12…$15.00
https://doi.org/10.1145/3281411.3281421

[1] [24, 33] were independently developed at the same time as this work.

[CoNEXT'18]



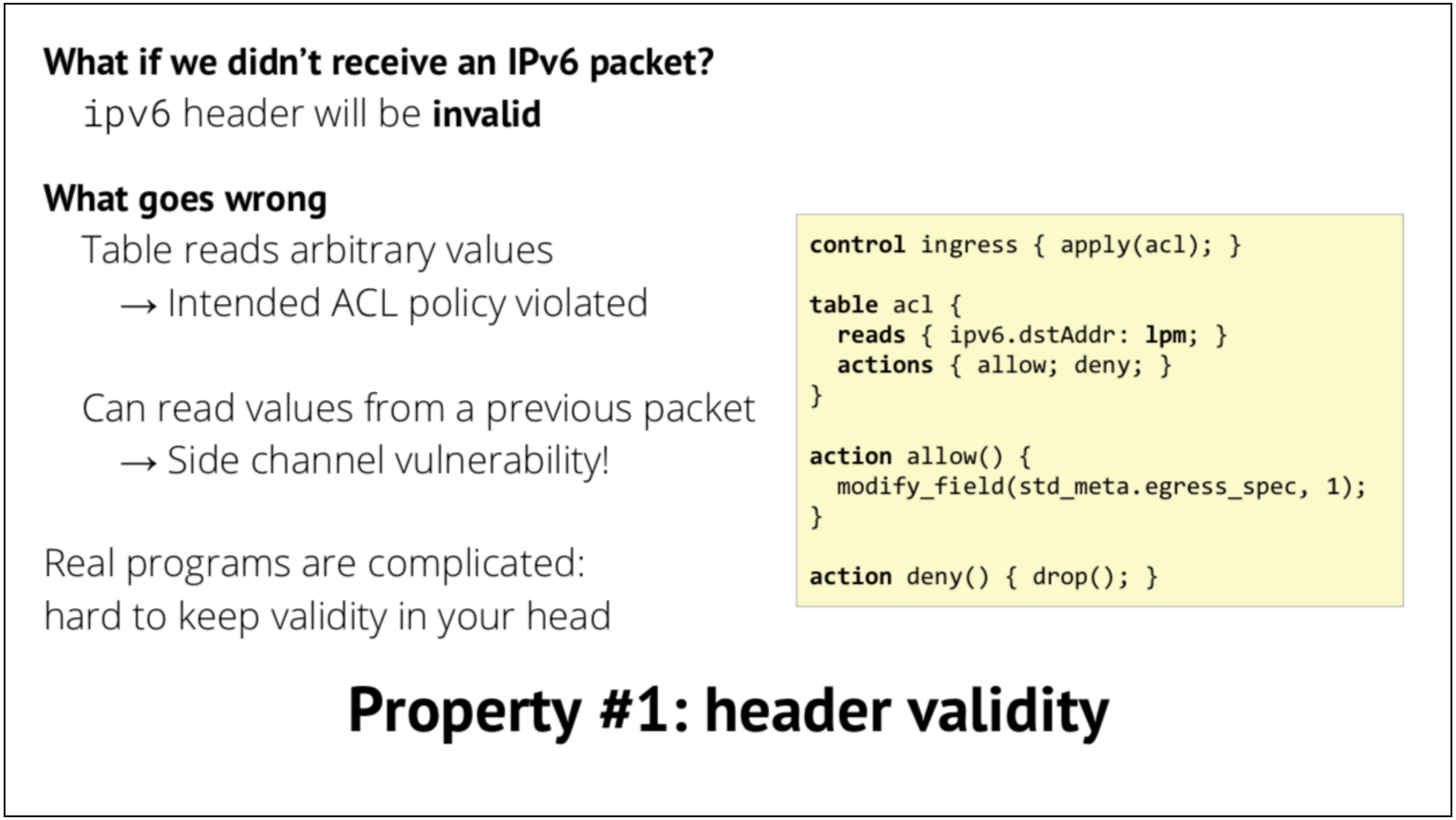
P4 by example

- P4 is a low-level language → many gotchas
- Let's explore by example!
  - IPv6 router w/ access control list (ACL)

```
control ingress { apply(acl); }

table acl {
  reads { ipv6.dstAddr: lpm; }
  actions { allow; deny; }
}

action allow() {
  modify_field(std_meta.egress_spec, 1);
}

action deny() { drop(); }
```

What could *possibly* go wrong?

Source: p4v, Practical Verification for Programmable Data Planes, Liu et al., 2018



What if we didn't receive an IPv6 packet?

`ipv6` header will be **invalid**

What goes wrong

Table reads arbitrary values

→ Intended ACL policy violated

Can read values from a previous packet

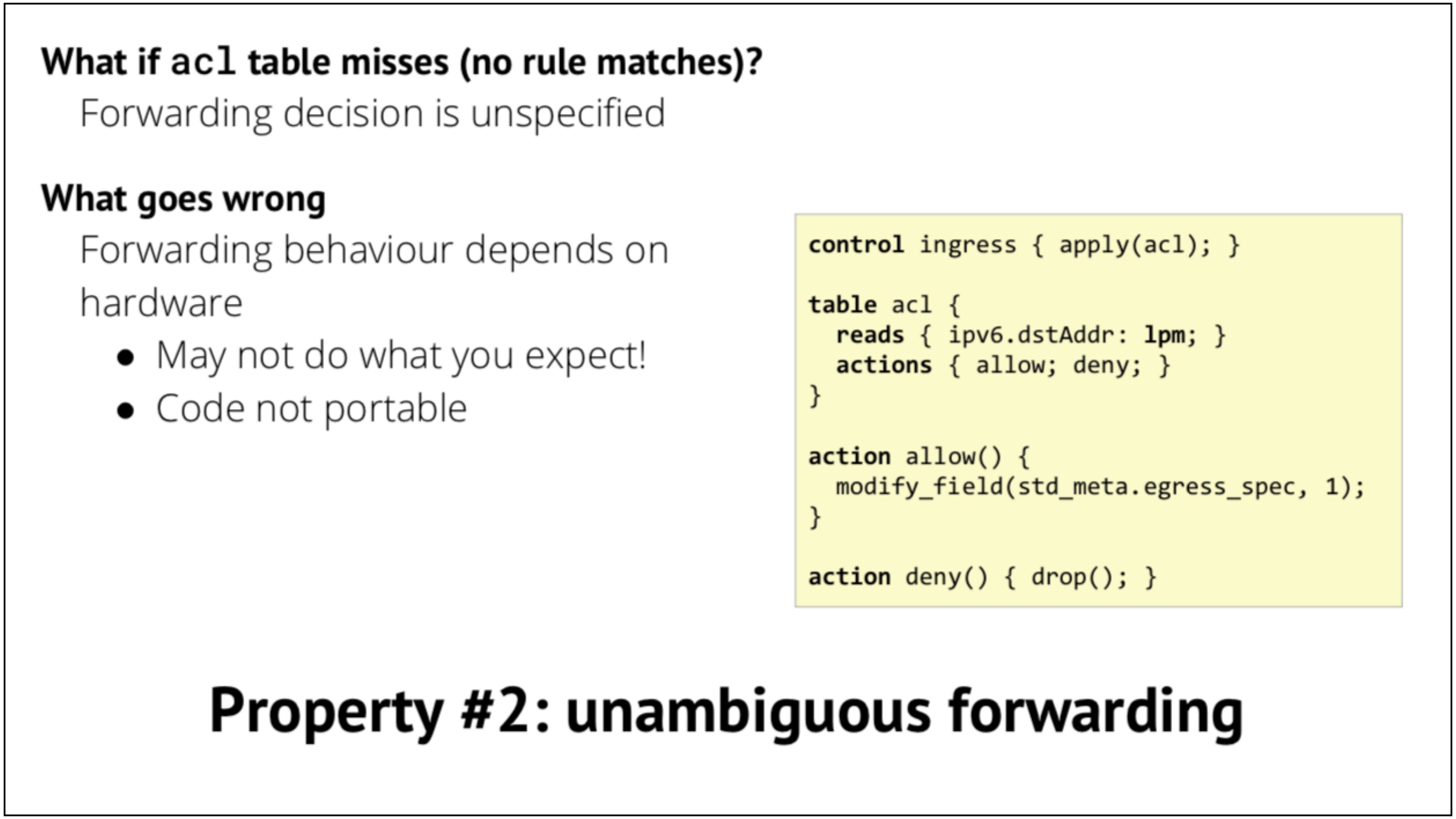
→ Side channel vulnerability!

Real programs are complicated:
hard to keep validity in your head

```
control ingress { apply(acl); }

table acl {
  reads { ipv6.dstAddr: lpm; }
  actions { allow; deny; }
}

action allow() {
  modify_field(std_meta.egress_spec, 1);
}

action deny() { drop(); }
```

Property #1: header validity

Source: p4v, Practical Verification for Programmable Data Planes, Liu et al., 2018



What if `acl` table misses (no rule matches)?

Forwarding decision is unspecified

What goes wrong

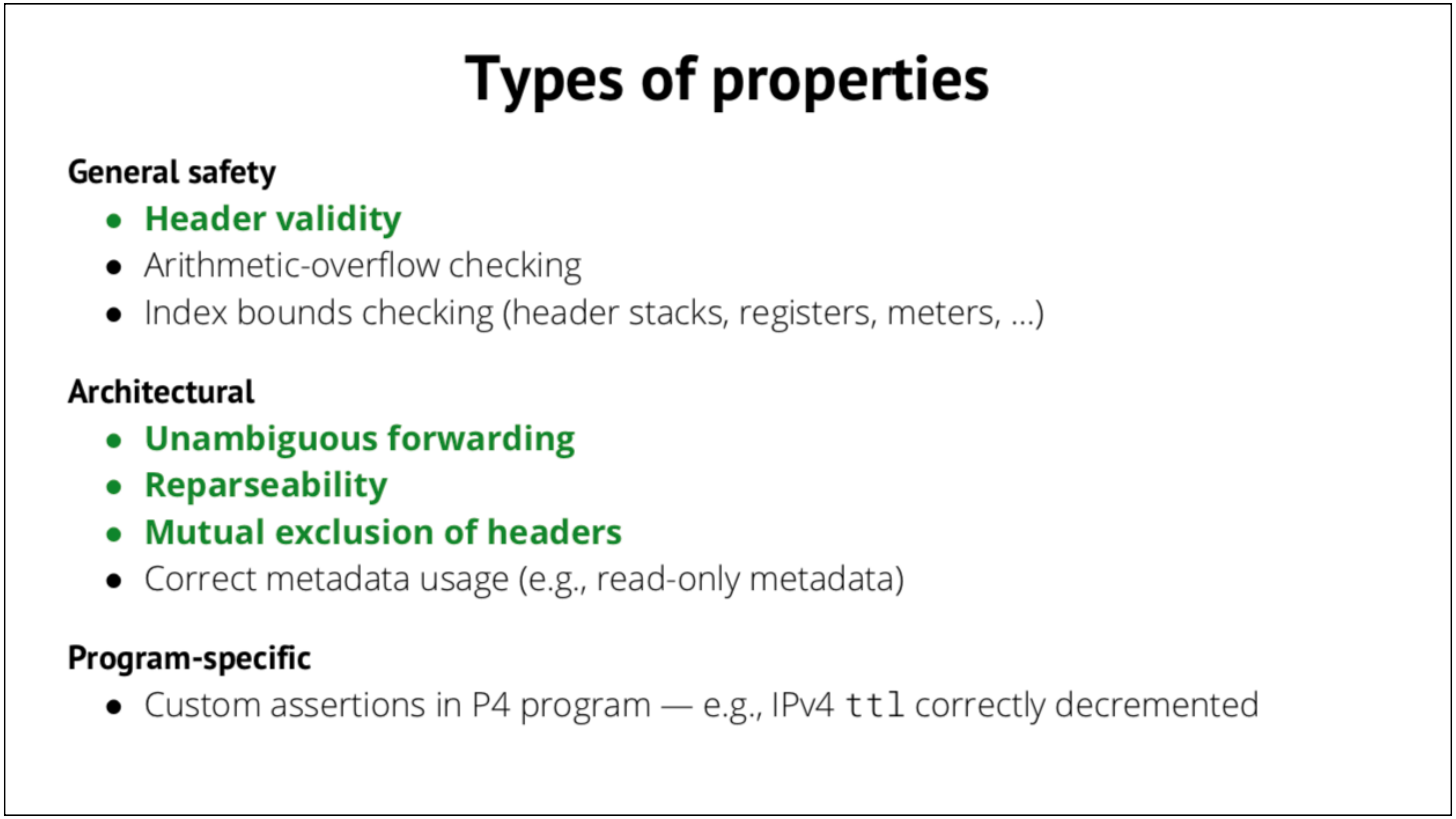
Forwarding behaviour depends on hardware

- May not do what you expect!
- Code not portable

```
control ingress { apply(acl); }

table acl {
  reads { ipv6.dstAddr: lpm; }
  actions { allow; deny; }
}

action allow() {
  modify_field(std_meta.egress_spec, 1);
}

action deny() { drop(); }
```

Property #2: unambiguous forwarding

Source: p4v, Practical Verification for Programmable Data Planes, Liu et al., 2018



Types of properties

General safety

- **Header validity**
- Arithmetic-overflow checking
- Index bounds checking (header stacks, registers, meters, …)

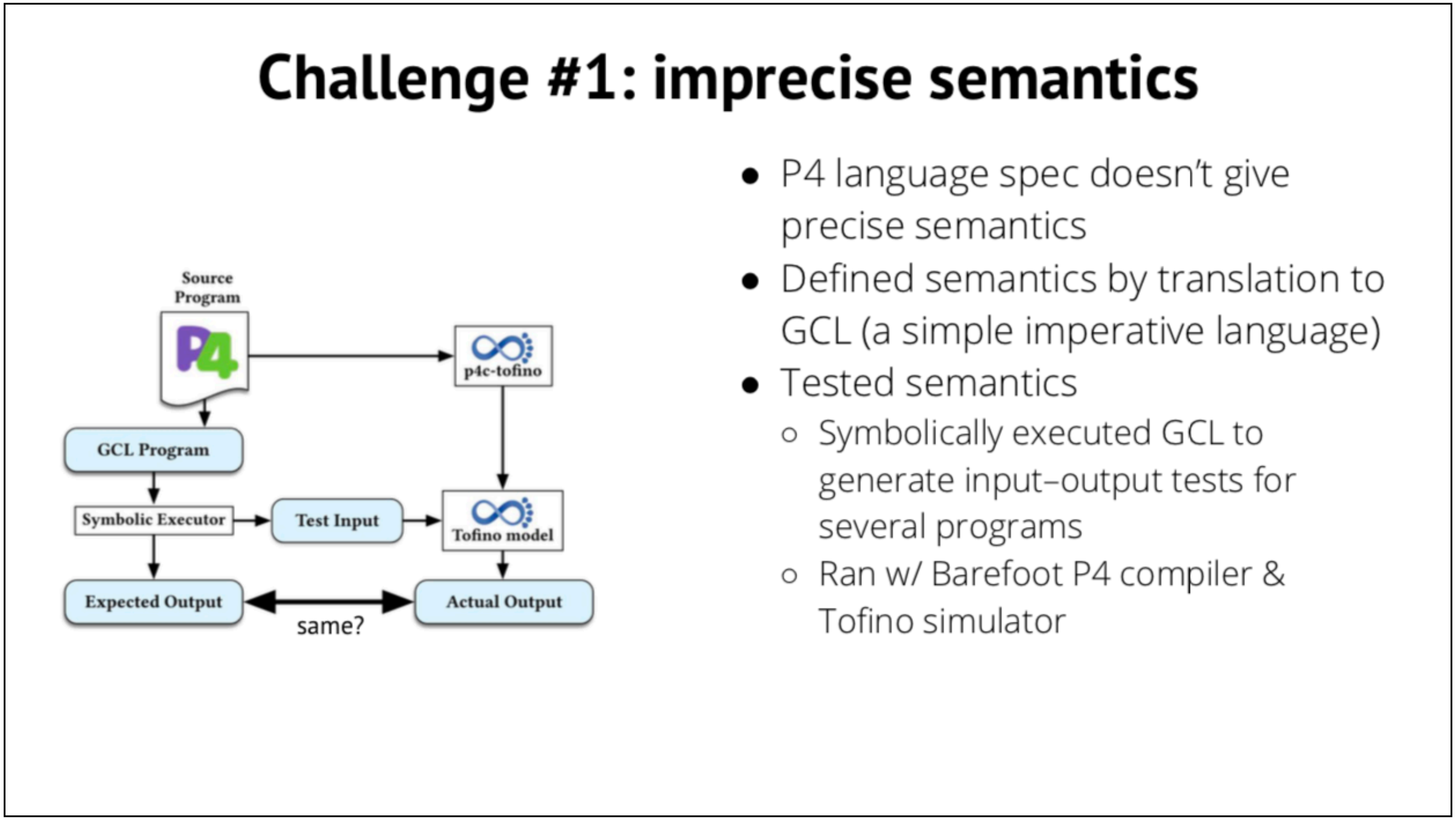
Architectural

- **Unambiguous forwarding**
- **Reparseability**
- **Mutual exclusion of headers**
- Correct metadata usage (e.g., read-only metadata)

Program-specific

- Custom assertions in P4 program — e.g., IPv4 `ttl` correctly decremented

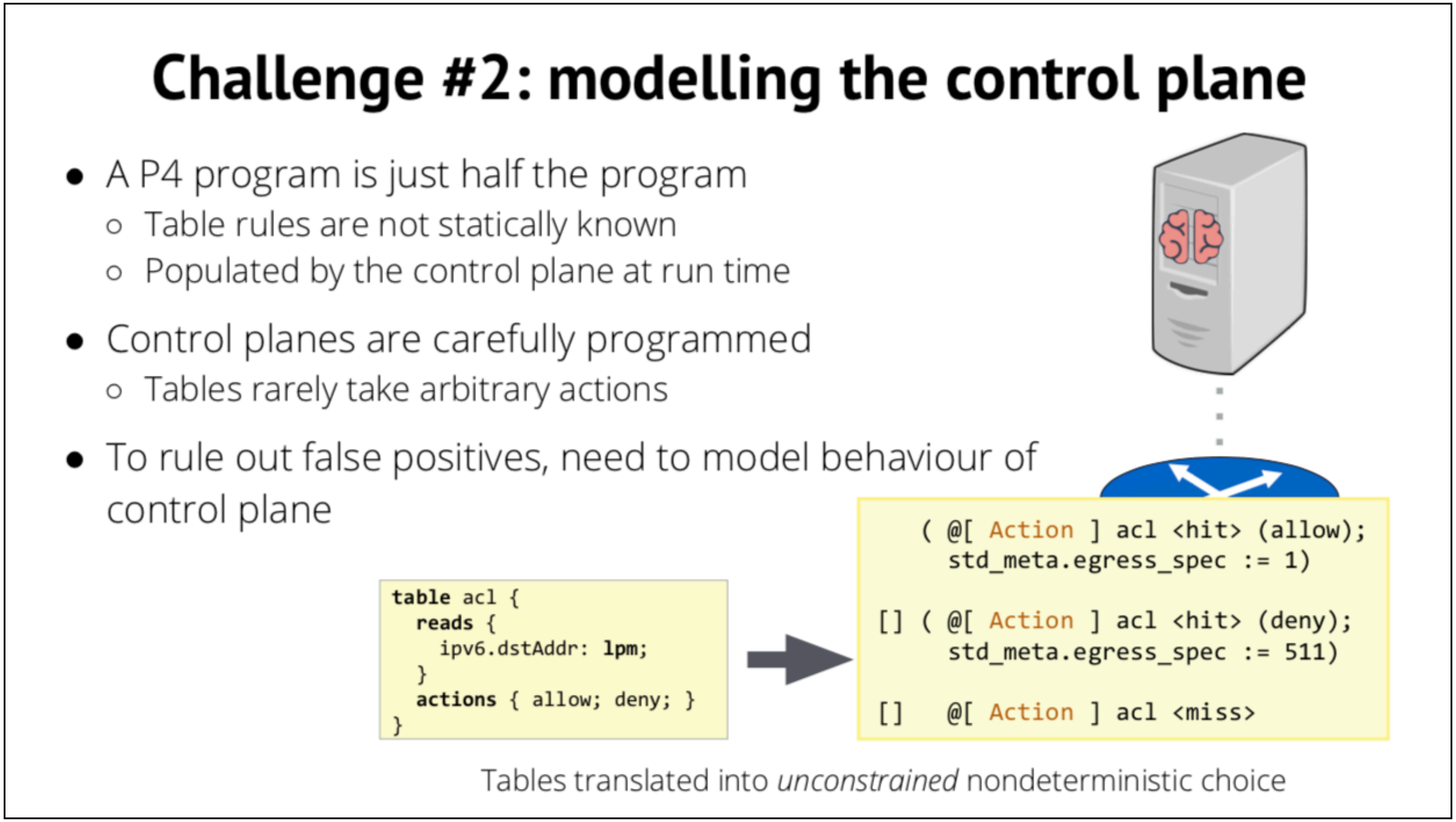
Source: p4v, Practical Verification for Programmable Data Planes, Liu et al., 2018



Challenge #1: imprecise semantics

- P4 language spec doesn't give precise semantics
- Defined semantics by translation to GCL (a simple imperative language)
- Tested semantics
  - Symbolically executed GCL to generate input–output tests for several programs
  - Ran w/ Barefoot P4 compiler & Tofino simulator

Source: p4v, Practical Verification for Programmable Data Planes, Liu et al., 2018



Challenge #2: modelling the control plane

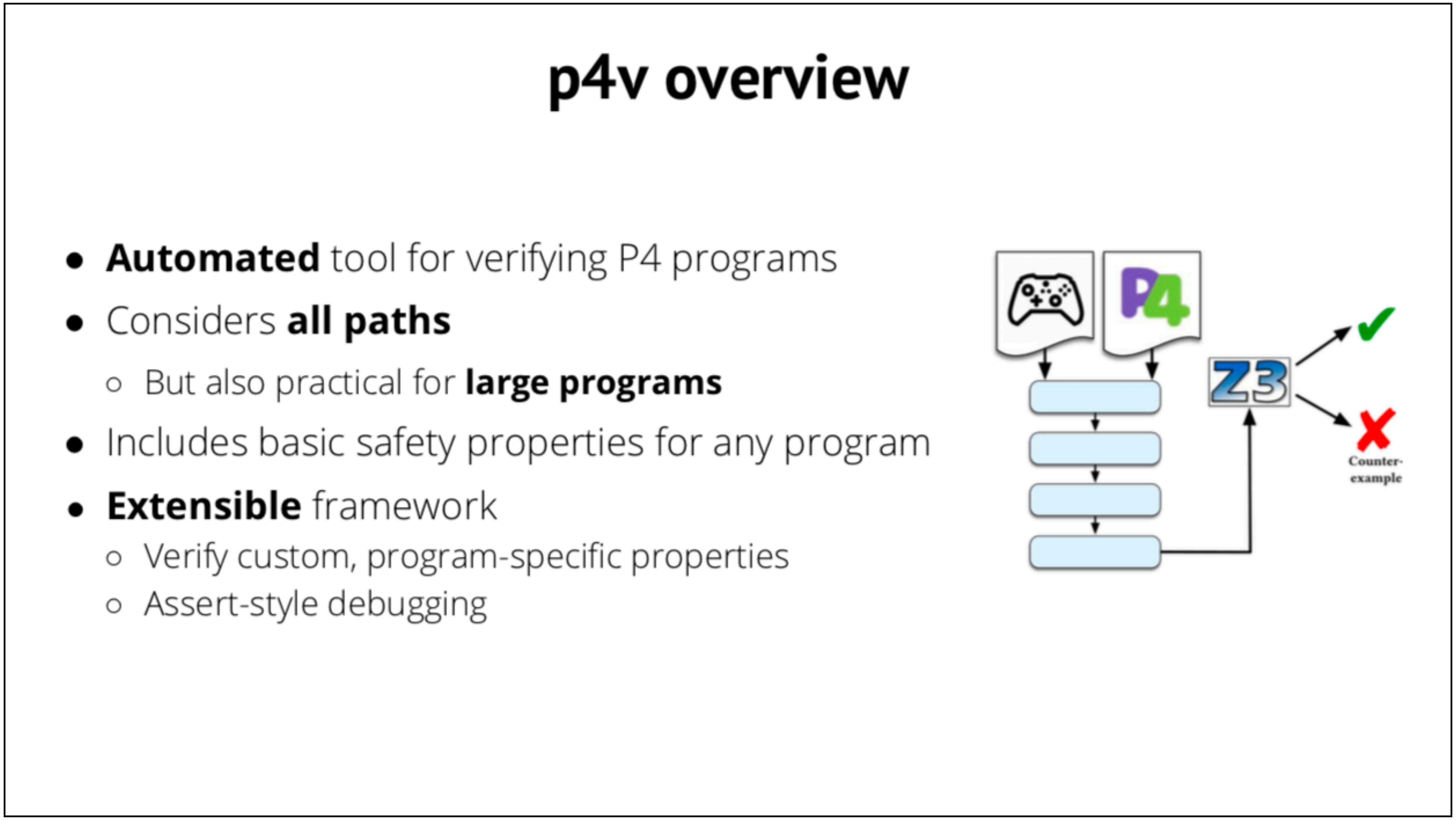
- A P4 program is just half the program
  - Table rules are not statically known
  - Populated by the control plane at run time
- Control planes are carefully programmed
  - Tables rarely take arbitrary actions
- To rule out false positives, need to model behaviour of control plane

```
table acl {
  reads {
    ipv6.dstAddr: lpm;
  }
  actions { allow; deny; }
}
```

```
   ( @[ Action ] acl <hit> (allow);
    std_meta.egress_spec := 1)

[] ( @[ Action ] acl <hit> (deny);
    std_meta.egress_spec := 511)

[]   @[ Action ] acl <miss>
```

Tables translated into *unconstrained* nondeterministic choice

Source: p4v, Practical Verification for Programmable Data Planes, Liu et al., 2018



p4v overview

- **Automated** tool for verifying P4 programs
- Considers **all paths**
  - But also practical for **large programs**
- Includes basic safety properties for any program
- **Extensible** framework
  - Verify custom, program-specific properties
  - Assert-style debugging

P4
Z3
✓
✗
Counter-example

Source: p4v, Practical Verification for Programmable Data Planes, Liu et al., 2018

Data plane programmability for

Performance
Monitoring
Applications offloading

Platforms
Correctness
Management

for Data plane programmability

So you've a *verified* programmable networks…

How do you manage it?!

How do you perform planned maintenance?

now that you've state in your switches…

How do you run multiple applications in your switches?

monitoring, forwarding, load-balancing, etc.

How do you share resources amongst applications?

especially memory and # packet operations

We need an Operating System for the data plane

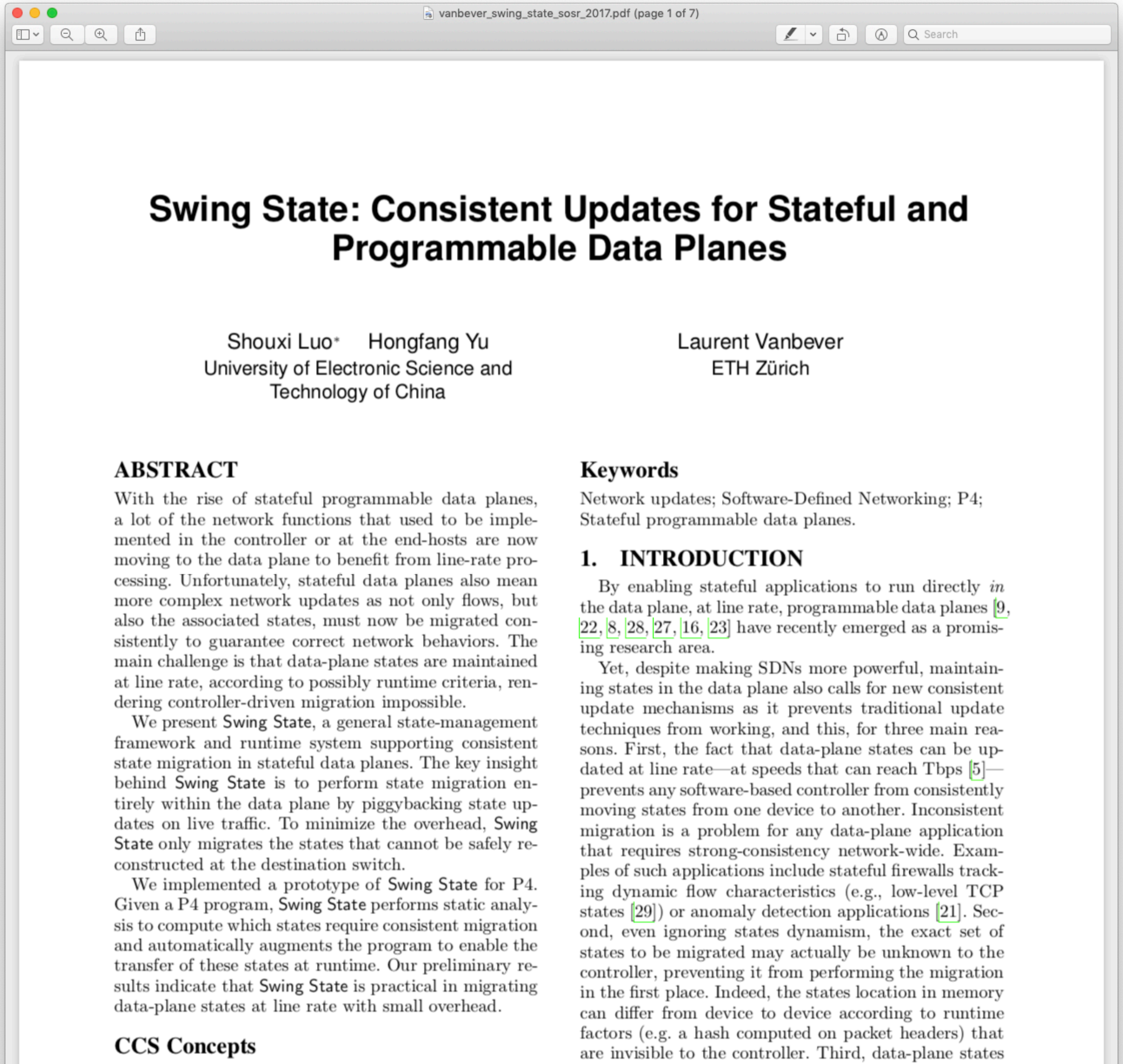
Definition
Wikipedia

An operating system is a system software that manages computer hardware and software resources and provides common services for computer programs.

Do we have that? Nope. Not yet at least.

We're working on it...

[SOSR'17]



Swing State: Consistent Updates for Stateful and Programmable Data Planes

Shouxi Luo* Hongfang Yu
University of Electronic Science and Technology of China

Laurent Vanbever
ETH Zürich

ABSTRACT

With the rise of stateful programmable data planes, a lot of the network functions that used to be implemented in the controller or at the end-hosts are now moving to the data plane to benefit from line-rate processing. Unfortunately, stateful data planes also mean more complex network updates as not only flows, but also the associated states, must now be migrated consistently to guarantee correct network behaviors. The main challenge is that data-plane states are maintained at line rate, according to possibly runtime criteria, rendering controller-driven migration impossible.

We present Swing State, a general state-management framework and runtime system supporting consistent state migration in stateful data planes. The key insight behind Swing State is to perform state migration entirely within the data plane by piggybacking state updates on live traffic. To minimize the overhead, Swing State only migrates the states that cannot be safely reconstructed at the destination switch.

We implemented a prototype of Swing State for P4. Given a P4 program, Swing State performs static analysis to compute which states require consistent migration and automatically augments the program to enable the transfer of these states at runtime. Our preliminary results indicate that Swing State is practical in migrating data-plane states at line rate with small overhead.

CCS Concepts

Keywords

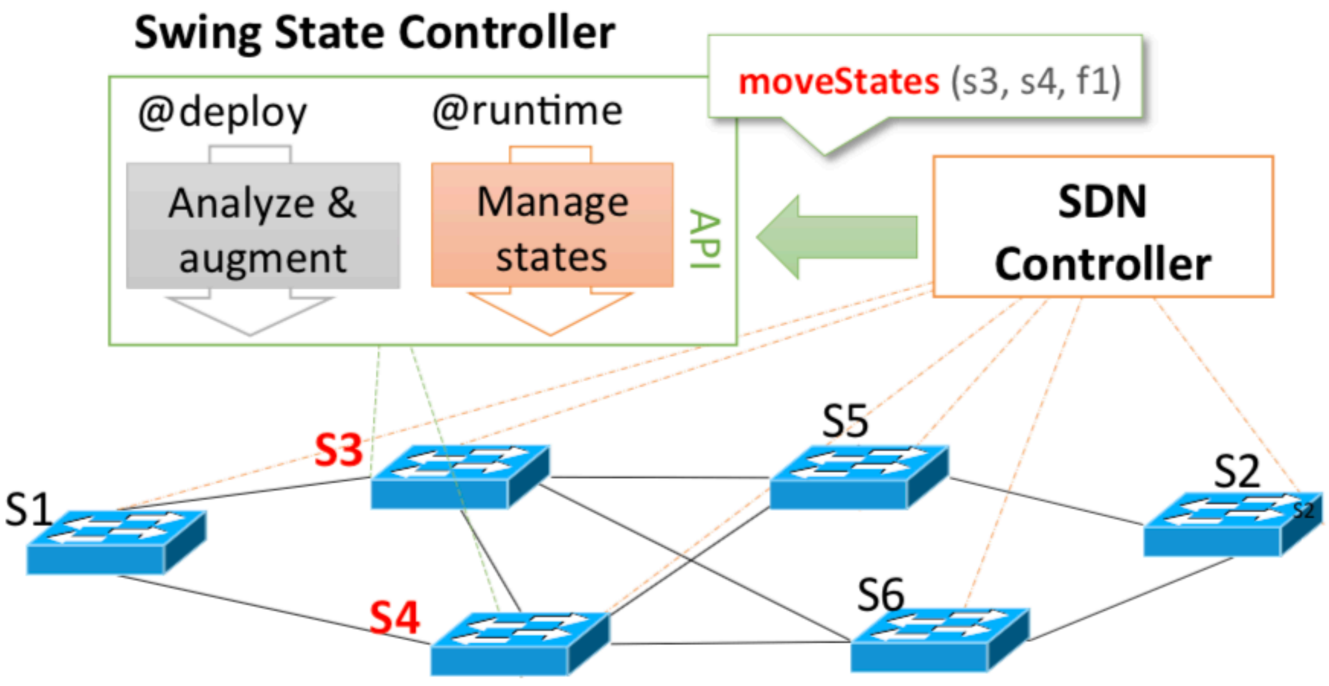
Network updates; Software-Defined Networking; P4; Stateful programmable data planes.

1. INTRODUCTION

By enabling stateful applications to run directly *in* the data plane, at line rate, programmable data planes [9, 22, 8, 28, 27, 16, 23] have recently emerged as a promising research area.

Yet, despite making SDNs more powerful, maintaining states in the data plane also calls for new consistent update mechanisms as it prevents traditional update techniques from working, and this, for three main reasons. First, the fact that data-plane states can be updated at line rate—at speeds that can reach Tbps [5]—prevents any software-based controller from consistently moving states from one device to another. Inconsistent migration is a problem for any data-plane application that requires strong-consistency network-wide. Examples of such applications include stateful firewalls tracking dynamic flow characteristics (e.g., low-level TCP states [29]) or anomaly detection applications [21]. Second, even ignoring states dynamism, the exact set of states to be migrated may actually be unknown to the controller, preventing it from performing the migration in the first place. Indeed, the states location in memory can differ from device to device according to runtime factors (e.g. a hash computed on packet headers) that are invisible to the controller. Third, data-plane states

Swing State is a state management framework with 1 primitive: moveStates



Source: Swing State: Consistent Updates for Stateful and Programmable Data Planes Luo et al., SOSR 2017

Advanced Topics in Communication Networks

COMPLETED

Lectures/Exercices

~7 weeks

how to program in P4

Group project

>= 7 weeks

in teams of 3

Advanced Topics in Communication Networks

COMPLETED

Lectures/Exercices

~7 weeks

how to program in P4

Group project

>= 7 weeks

in teams of 3

The group project starts next week

It accounts for 50% of your final grade

The evaluation of your project will depend on your implementation, report, and presentation

The evaluation of your project will depend on your implementation, report, and presentation

implementation

70%

achieves the basic goals

is properly documented

runs + results can be reproduced

The evaluation of your project will depend on your implementation, report, and presentation

implementation

70%

achieves the basic goals

is properly documented

runs + results can be reproduced

You'll have to write
a detailed README (in Markdown)
We'll provide you with a template

The evaluation of your project will depend on your implementation, report, and presentation

implementation 70%	achieves the basic goals is properly documented runs + results can be reproduced
report 15%, 10 pages max	describes the main building blocks evaluates the solution describes what each group member did

The evaluation of your project will depend on your implementation, report, and presentation

implementation	achieves the basic goals
70%	is properly documented
	runs + results can be reproduced
report	describes the main building blocks
15%, 10 pages max	evaluates the solution
	describes what each group member did
presentation	summarizes the problem and the solution
15%, 10 min. +questions	contains a *live* demo
	involves all group members

The final deadline for the project is

Wed Dec 16 at 23.59pm

This week	Select a proposal from the list (adv-net.ethz.ch) or send us your own proposal by email
Every week	Meet with the responsible assistant schedule a recurring slot in [10.15am; noon]
Mon Dec 16 11.59pm	Send us an archive with report, code, slides
Tue Dec 17 1.15pm—	Groups presentation + course/exam debrief attendance is mandatory

The project has to be done in groups of 3 students

"Matching" process for incomplete groups via Slack

Project grade is shared by each group member provided that each collaborated (roughly equally)

- Let us know in advance if that's *not* the case
- Briefly describe in the report the contribution of each group member
- Each group member should be involved in the presentation and be able to answer questions

If you want to propose your own project,
send us an email describing it by Thu Oct 31 11.59am

lvanbever@ethz.ch, cedgar@ethz.ch

Quick overview of the proposals

Albert

Thomas

Roland

Alexander

Maria

Edgar

Quick overview of the proposals

Albert

Thomas

Roland

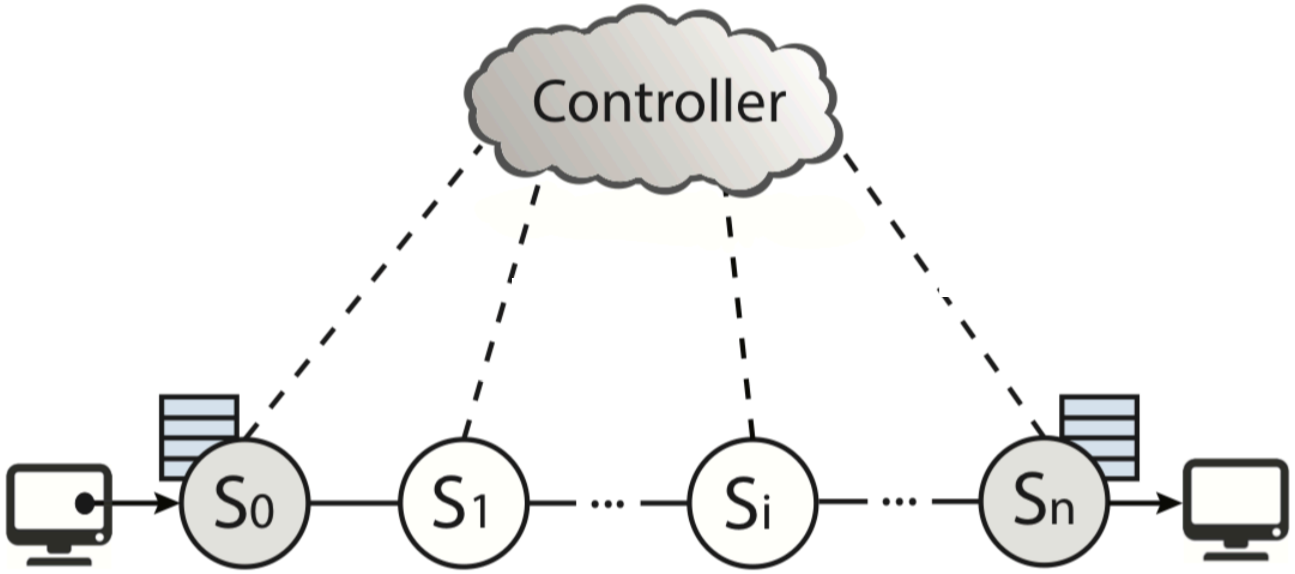
Alexander

Maria

Edgar

Proposal #1

SDNSec: Forwarding Accountability for SDN Data Plane



Current data plane lacks accountability:

✗ Enforcing forwarding policies

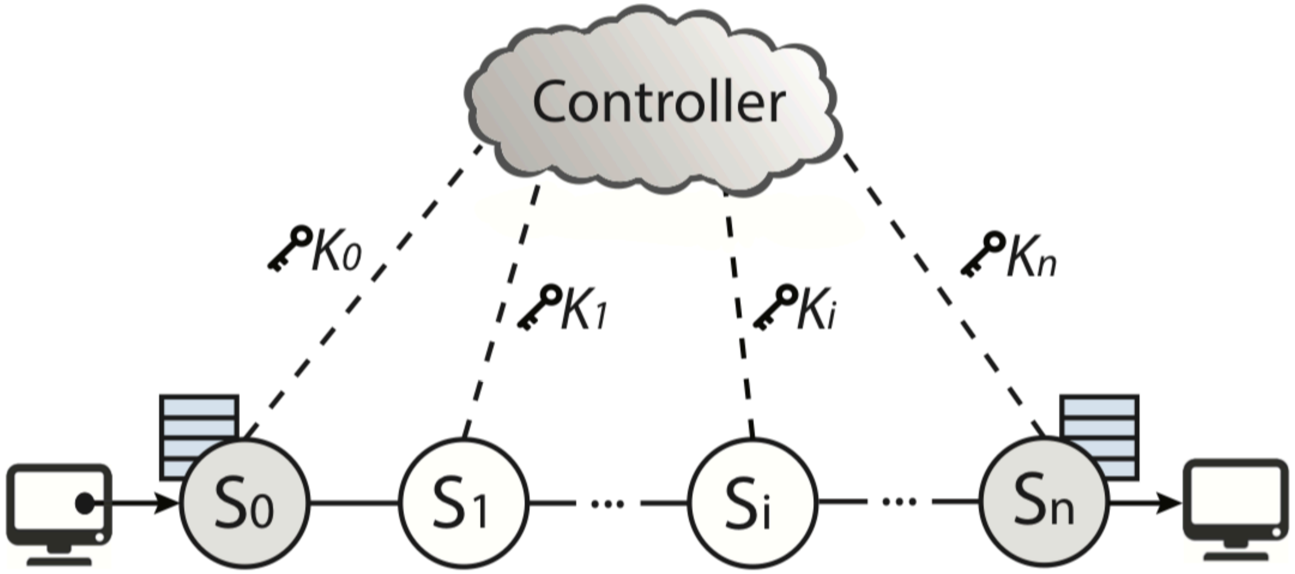
✗ Validating that policies have not been violated

✗ Consistency guarantees under reconfiguration

[ICCCN 2016] NEC Corporation Japan, ETH Zurich (Perrig et. al.)

Proposal #1

SDNSec: Forwarding Accountability for SDN Data Plane



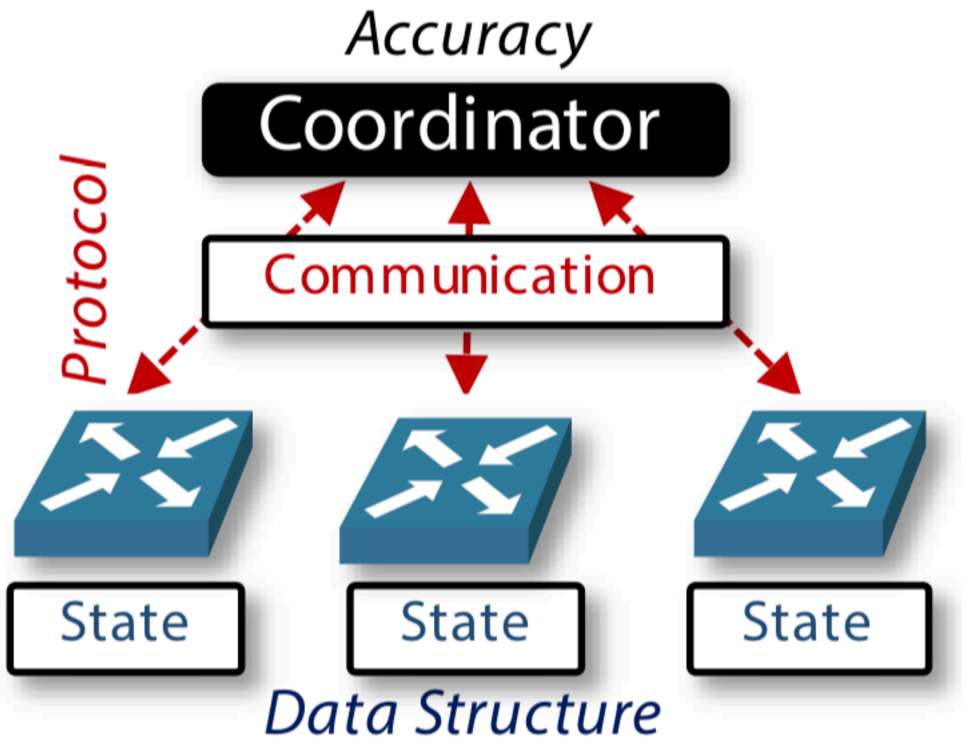
With SDNSec:

- ✓ Ingress-switch adds path in header
- ✓ Core-switches extract header, decrypt and forward
- ✓ Controller verifies policy

[ICCCN 2016] NEC Corporation Japan, ETH Zurich (Perrig et. al.)

Proposal #2

Herding the Elephants: Detecting Network-Wide Heavy Hitters with Limited Resources



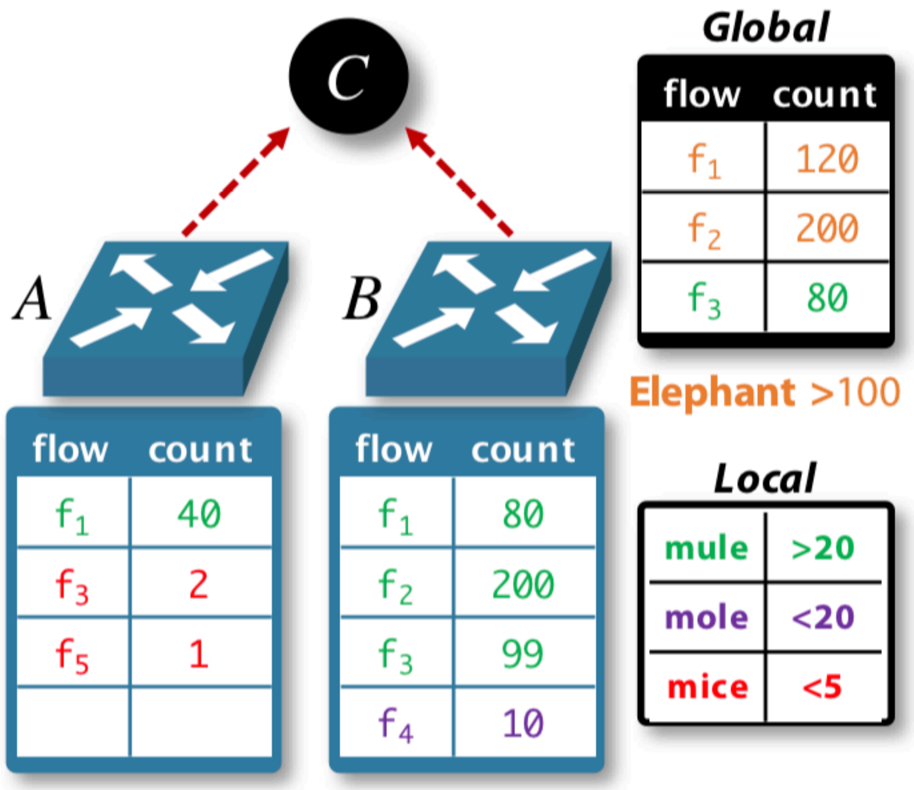
Separating elephant from mice is key in network management:

- ✗ Sampling is not accurate and results are delayed
- ✗ App-specific sketches limit network visibility

[Semantic scholar] Princeton, Walter Robert J. Harrison (Rexford et. al.)

Proposal #2

Herding the Elephants: Detecting Network-Wide Heavy Hitters with Limited Resources



flow	count
f_1	40
f_3	2
f_5	1

flow	count
f_1	80
f_2	200
f_3	99
f_4	10

Global

flow	count
f_1	120
f_2	200
f_3	80

Elephant >100

Local

mule	>20
mole	<20
mice	<5

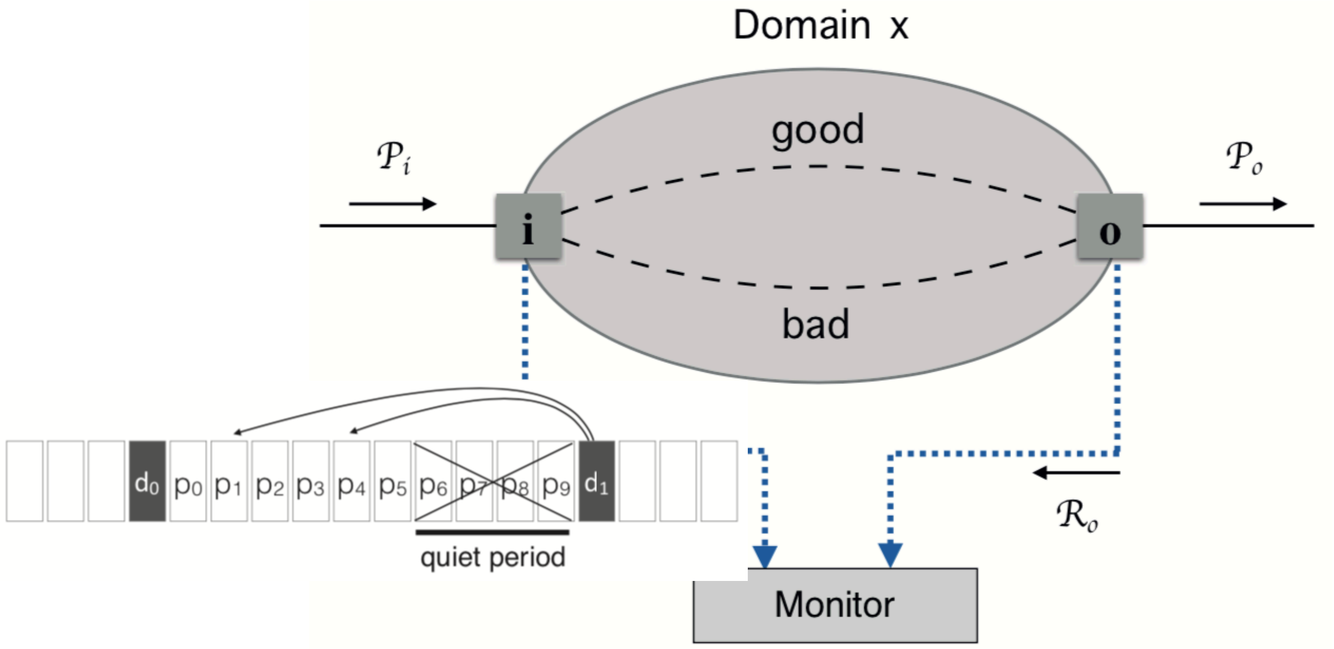
- ✓ Herd provides accuracy network wide
- ✓ Switches allocate resources based on flow type
- ✓ Switches notify controller when local heavy hitter
- ✓ Controller finds global heavy hitters

Extension: Network-Wide Heavy Hitter Detection with Commodity Switches

[Semantic scholar] Princeton, Walter Robert J. Harrison (Rexford et. al.)

Proposal #3

Retroactive Packet Sampling for Traffic Receipts



- ✗ Network nodes could cheat in monitoring
- ✗ Performing better for selected samples
- ✓ Delayed disclosure mechanism prevents it
- ✓ Estimates loss-rate and delay from controller

Extension: SQR: In-Network Packet Loss Recovery

[SIGMETRICS 2019] EPFL Lausanne, ETH Zurich (Perrig et. al.)

Quick overview of the proposals

Albert

Thomas

Roland

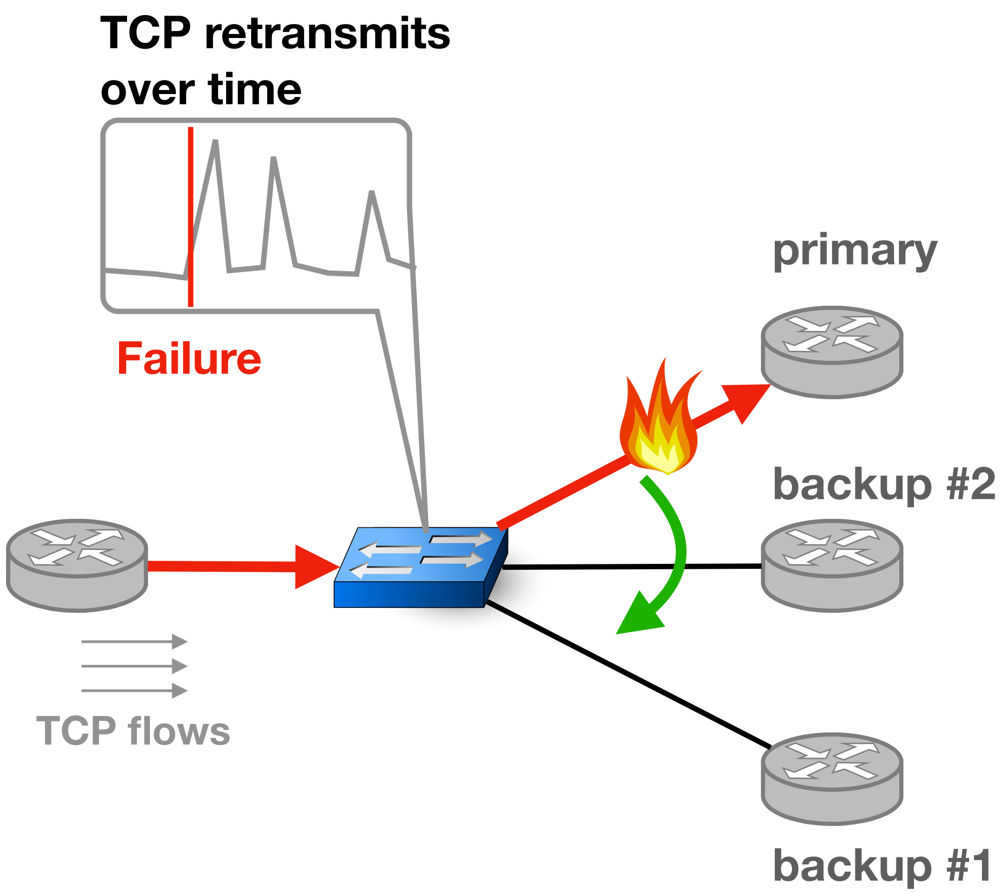
Alexander

Maria

Edgar

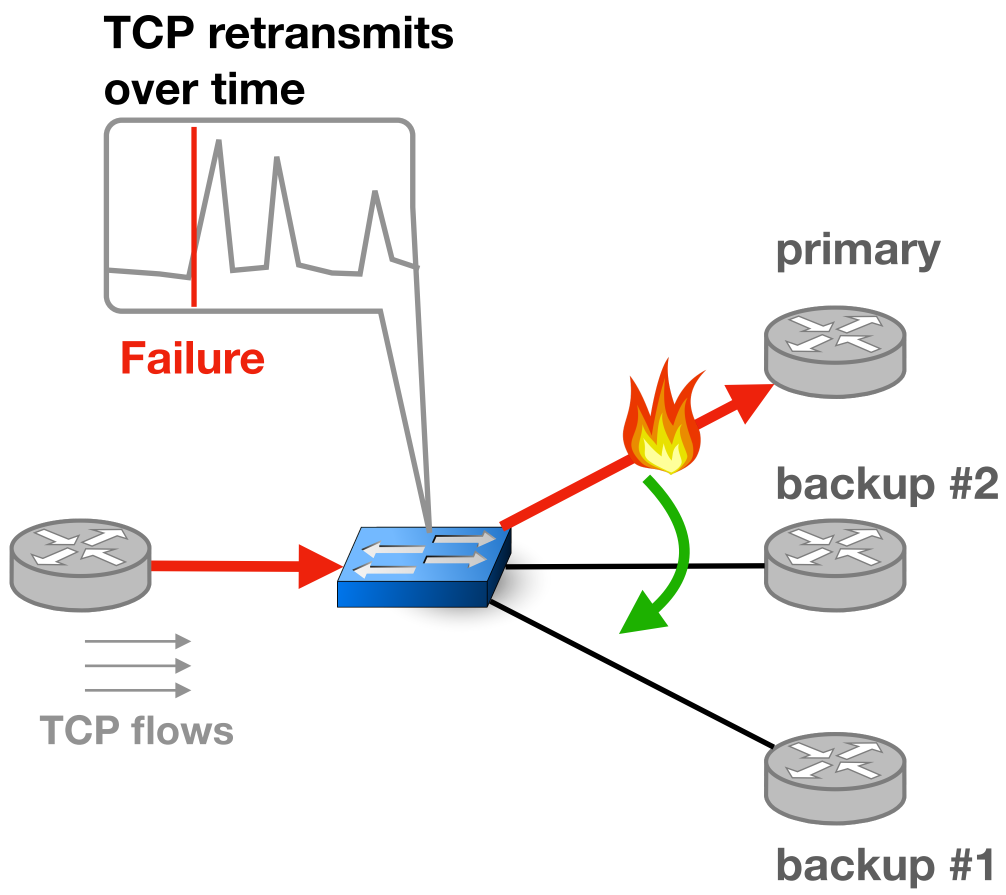
Blink: Fast Connectivity Recovery Entirely in the Data Plane

NSDI'19



Blink: Fast Connectivity Recovery Entirely in the Data Plane

NSDI'19



Goal: improving blink

1. Selecting flows with low RTTs

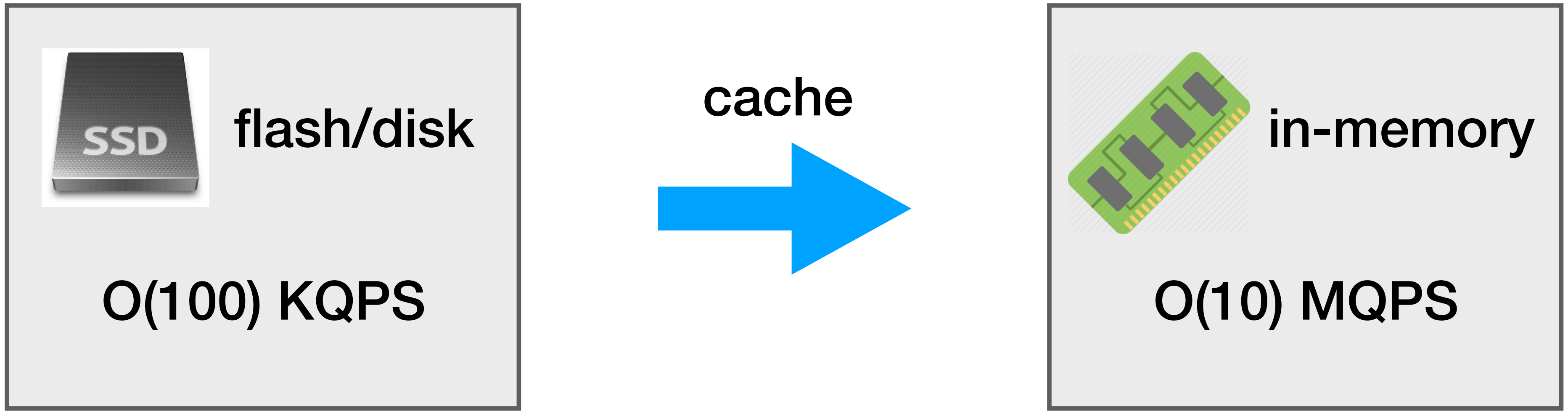
2. Monitoring backup next-hops continuously to reroute faster

3. Monitoring the throughput to improve accuracy

NetCache: Balancing Key-Value Stores with Fast In-Network Caching

SOSP'17 (for 2 students only)

Traditional way to implement a key value store:



flash/disk

O(100) KQPS

cache

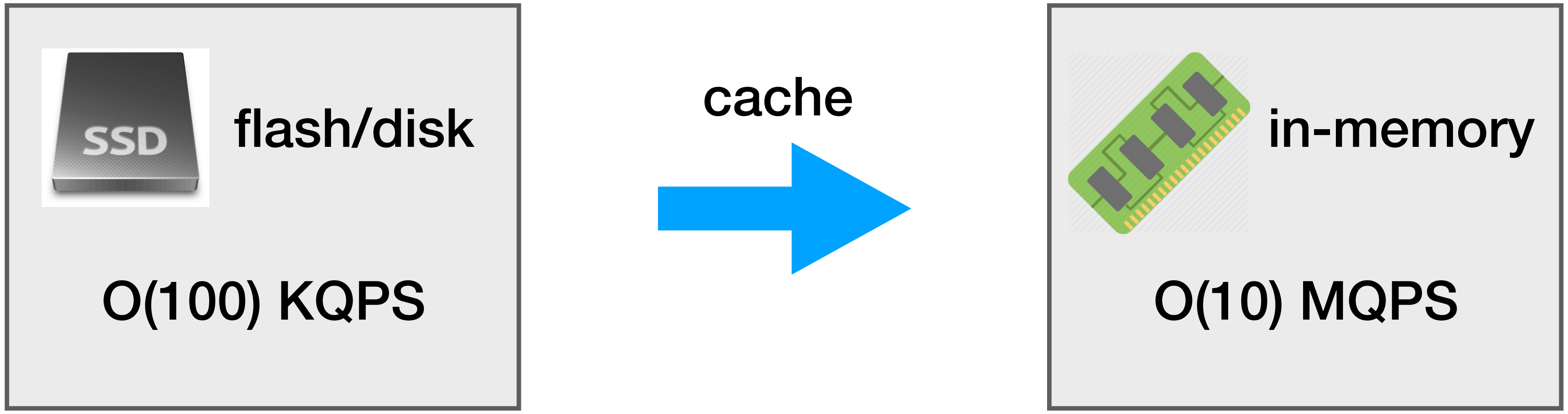
in-memory

O(10) MQPS

NetCache: Balancing Key-Value Stores with Fast In-Network Caching

SOSP'17 (for 2 students only)

Traditional way to implement a key value store:



flash/disk

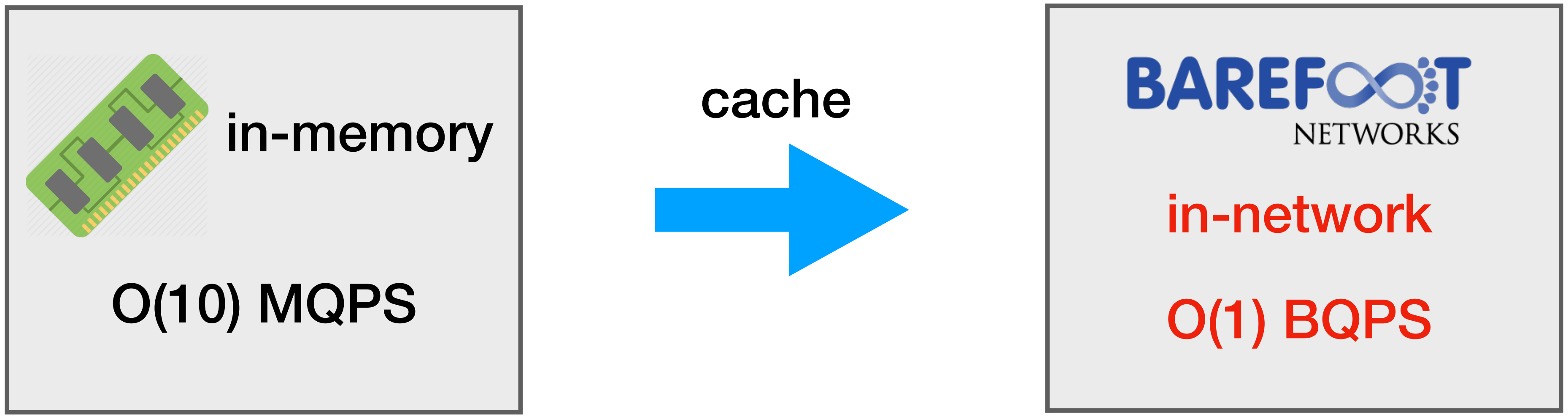
O(100) KQPS

cache

in-memory

O(10) MQPS

NetCache:



in-memory

O(10) MQPS

cache

BAREFOOT NETWORKS

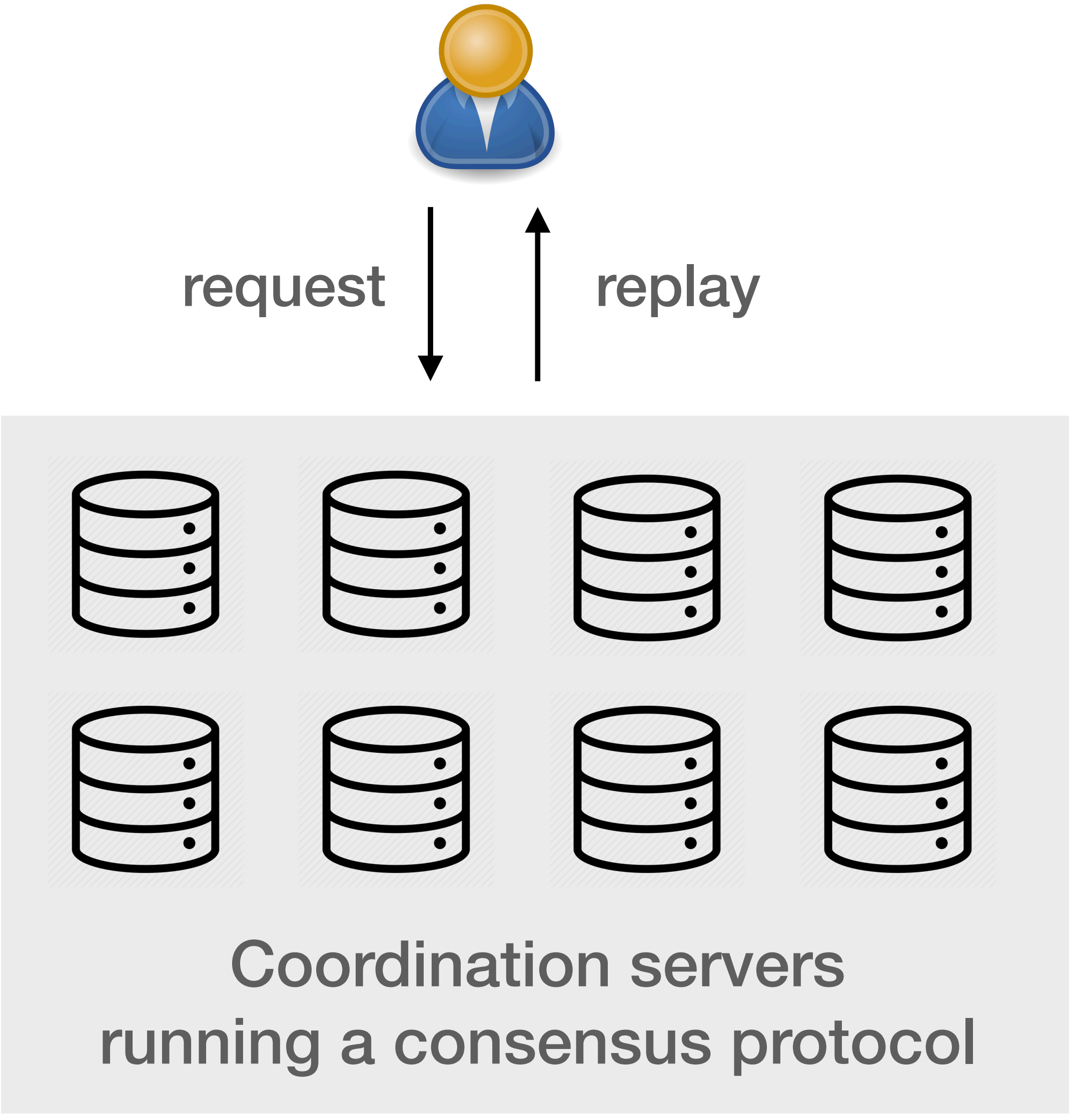
in-network

O(1) BQPS

NetChain: Scale-Free sub-RTT Coordination

NSDI'18 (for 2 students only)

Traditionally, key-value stores are replicated for *fault-tolerance*



request

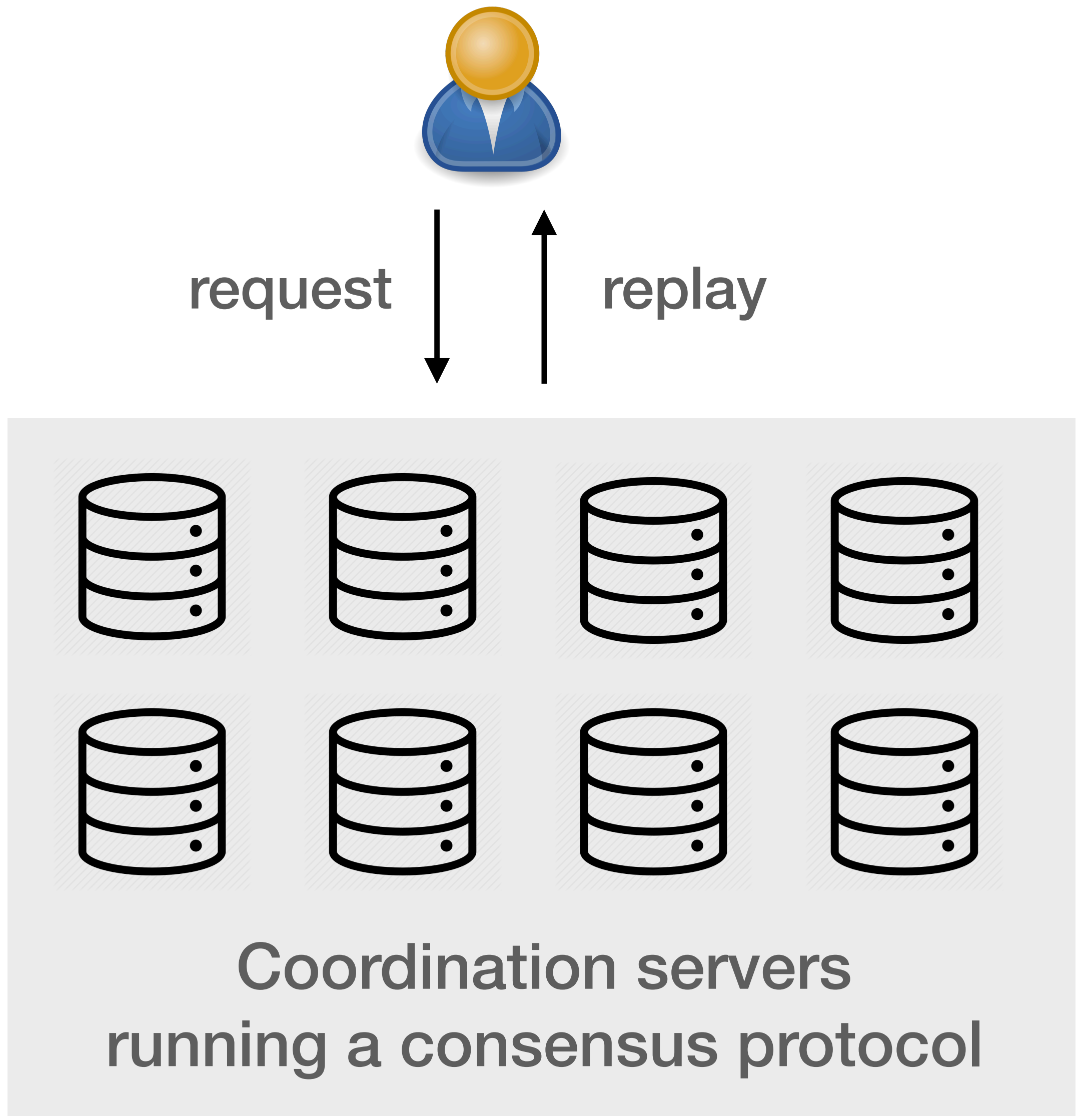
replay

Coordination servers running a consensus protocol

NetChain: Scale-Free sub-RTT Coordination

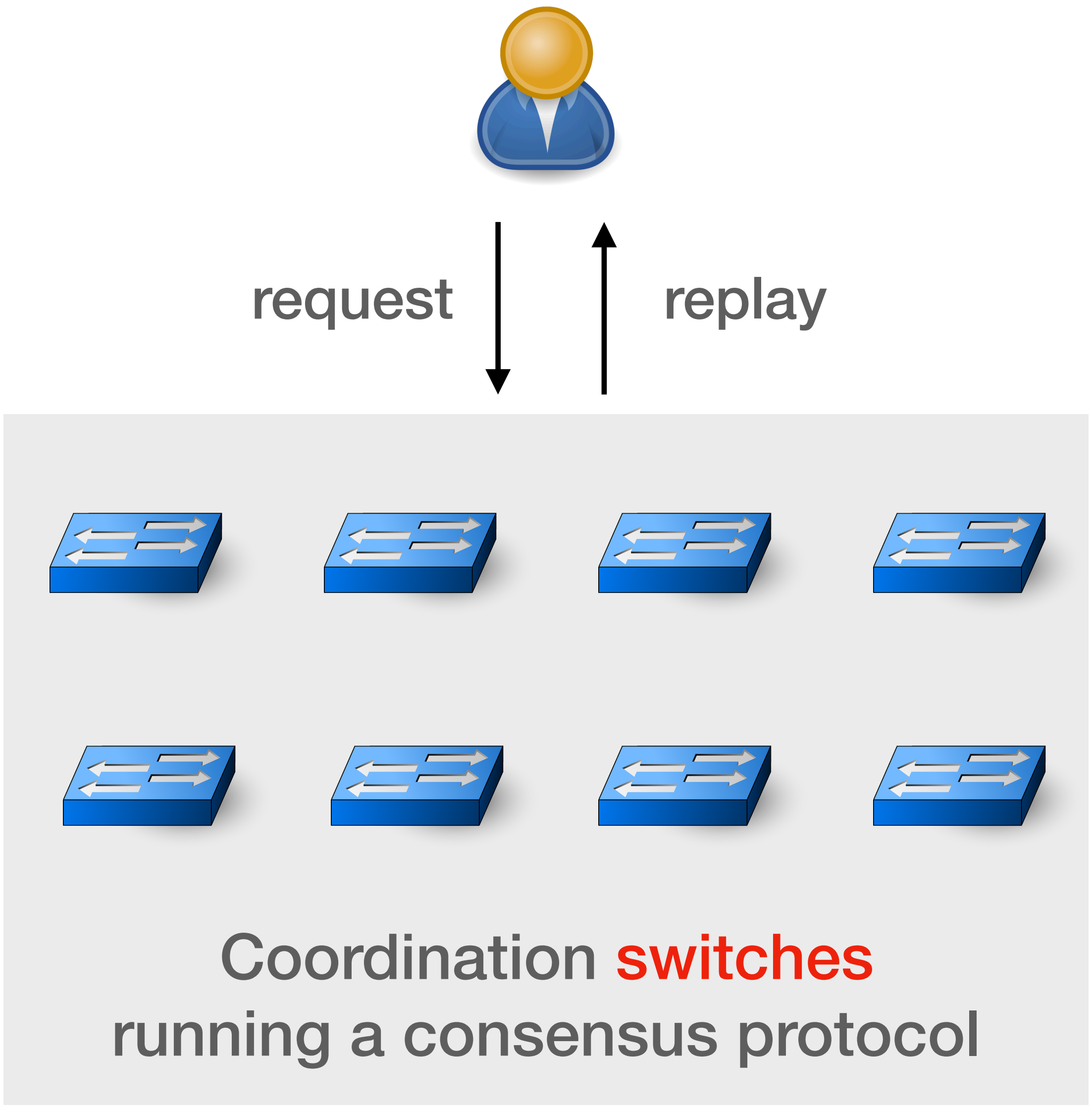
NSDI'18 (for 2 students only)

Traditionally, key-value stores are replicated for *fault-tolerance*



Coordination servers running a consensus protocol

NetChain



Coordination switches running a consensus protocol

Quick overview of the proposals

Albert

Thomas

Roland

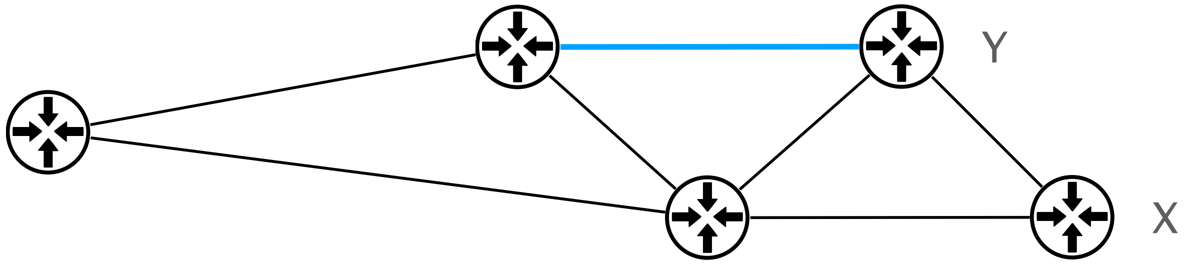
Alexander

Maria

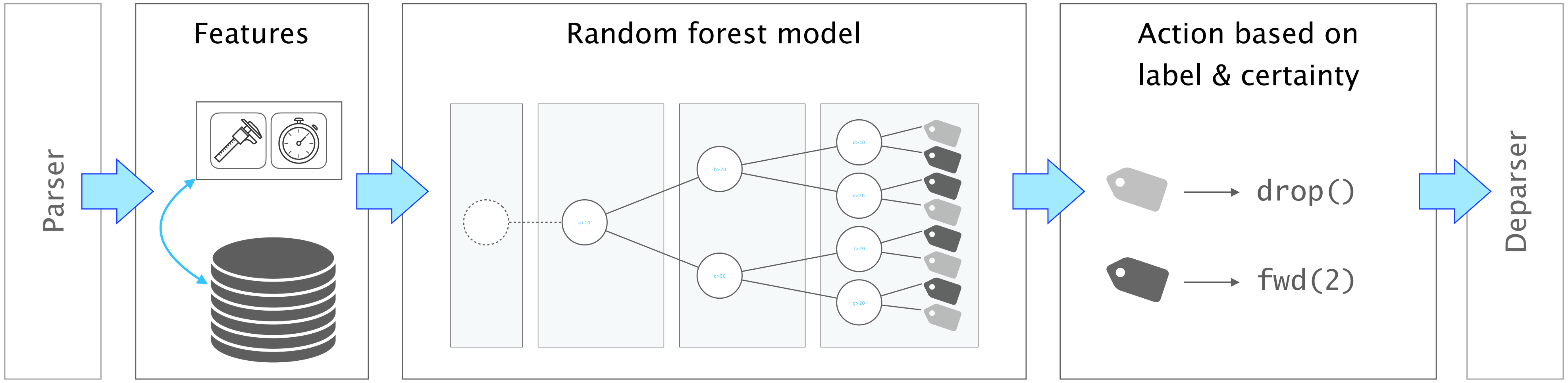
Edgar

NetHide: Secure and Practical Network Topology Obfuscation

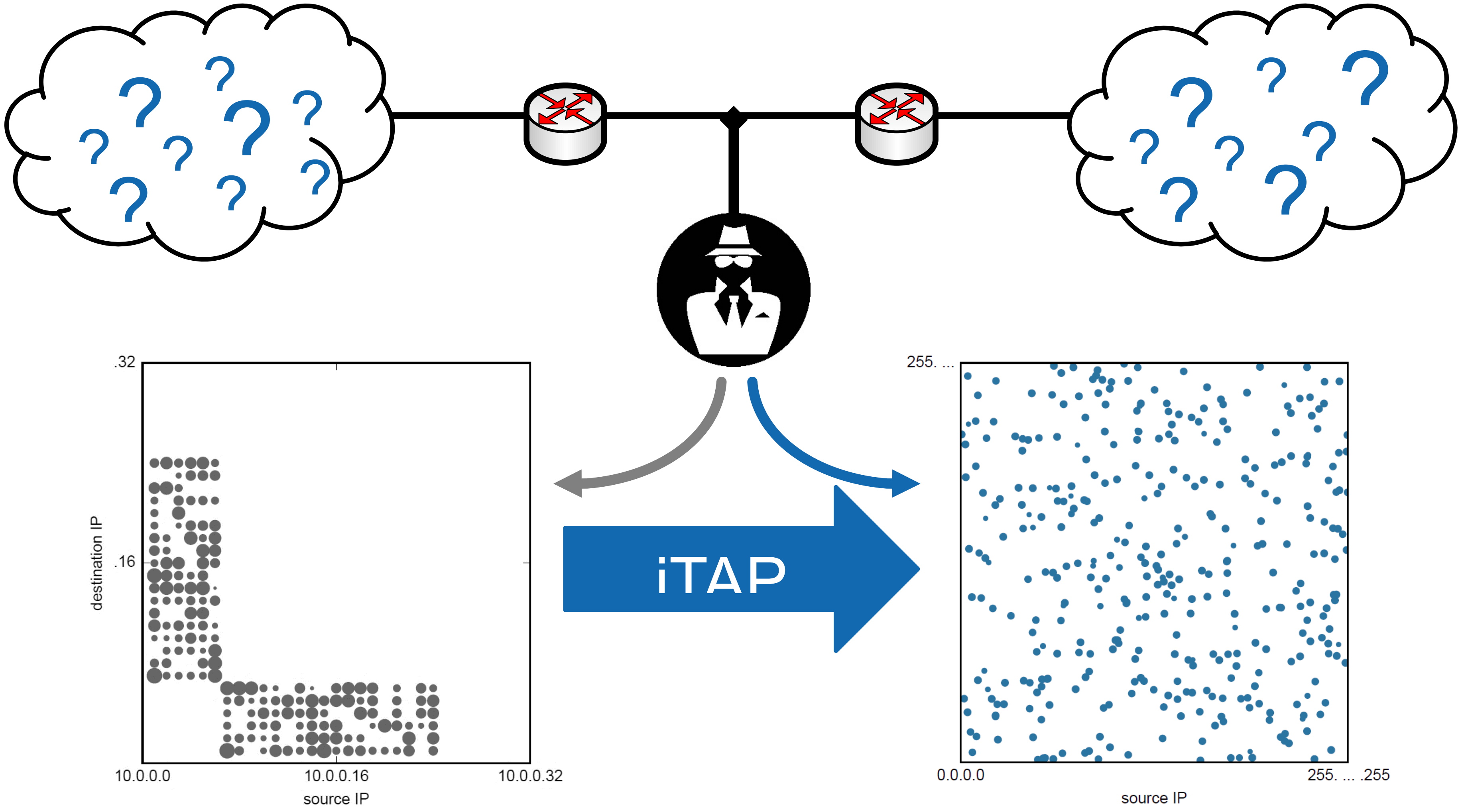
If I receive a packet to X with TTL = i,
I should send it to Y with TTL = j



pForest: In-Network Inference with Random Forests



iTAP: In-Network Traffic Analysis Prevention Using Software-Defined Networks



Quick overview of the proposals

Albert

Thomas

Roland

Alexander

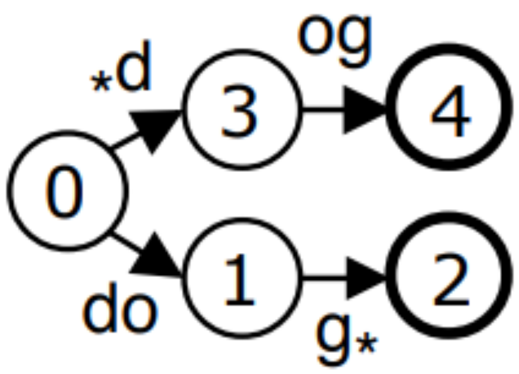
Maria

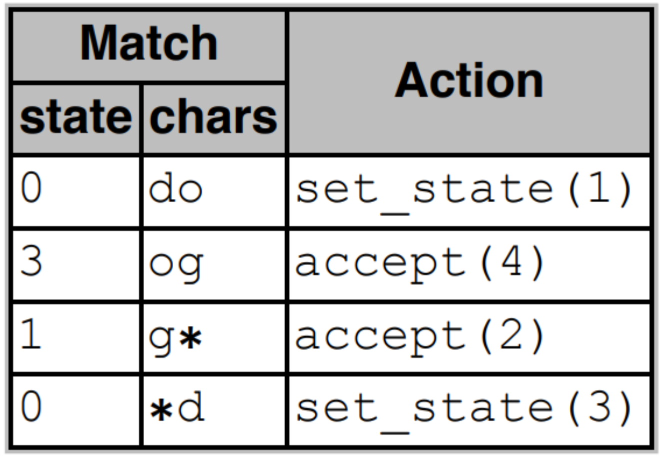
Edgar

Proposal #4

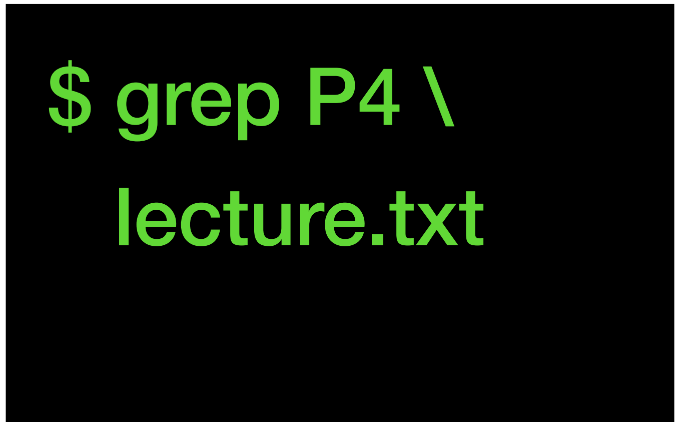
Fast String Searching on PISA

P4 is very limited, e.g. it cannot work with strings.
Or can it? It can even handle regular expressions!





Match		Action
state	chars	
0	do	set_state(1)
3	og	accept(4)
1	g*	accept(2)
0	*d	set_state(3)



```
$ grep P4 \
  lecture.txt
```

In the control-plane:
Translate regex
to automaton.

In the data-plane:
Execute automaton
using recirculation.

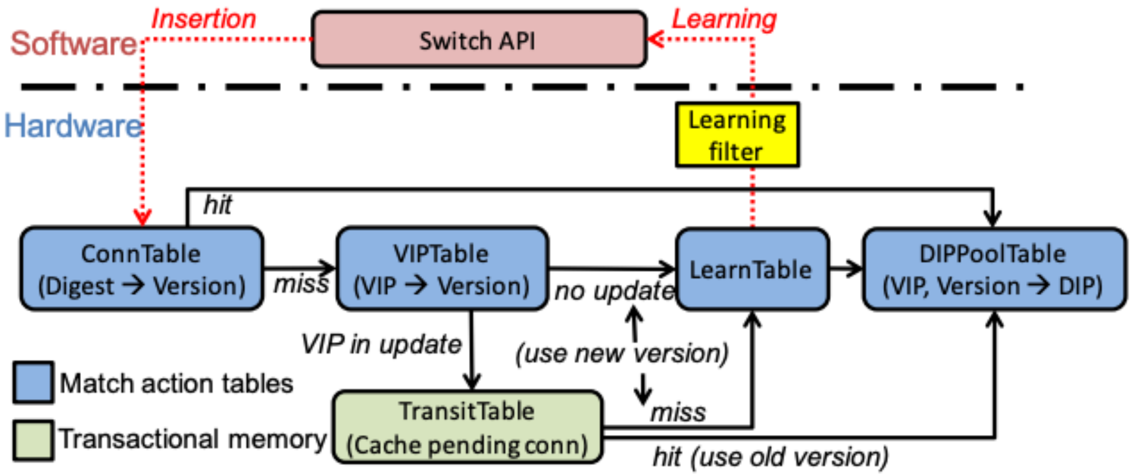
Evaluation:
Compare to grep
and co.

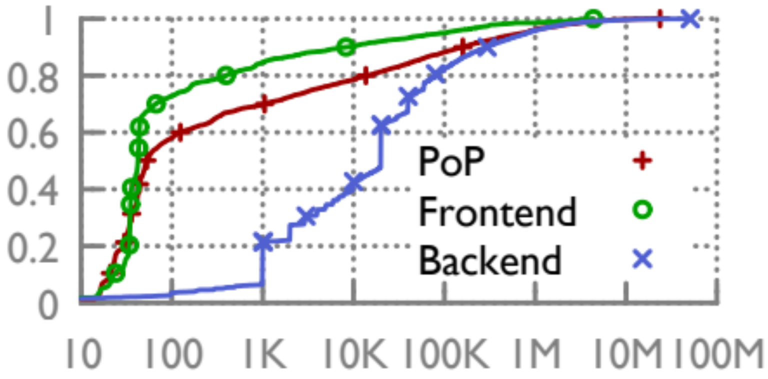
[SOSR 2019] USI, Barefoot (Jepsen et. al.)

Proposal #5

SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs

SilkRoad using a P4 switch to replace software load balancers.
It can handle millions of stateful connections using multi-level caching.





In the control-plane:
Accept incoming connections.

In the data-plane:
Keep track of existing connections.

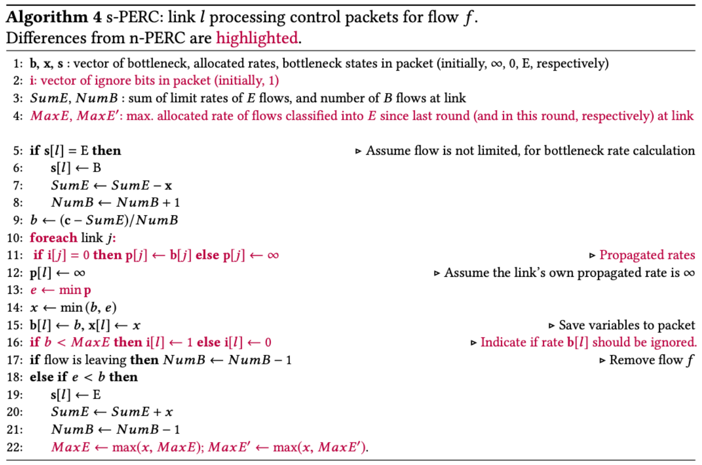
Evaluation:
Test performance at large scale.

[SIGCOMM 2017] USC, Yale, Facebook, Barefoot (Miao et. al.)

Proposal #6

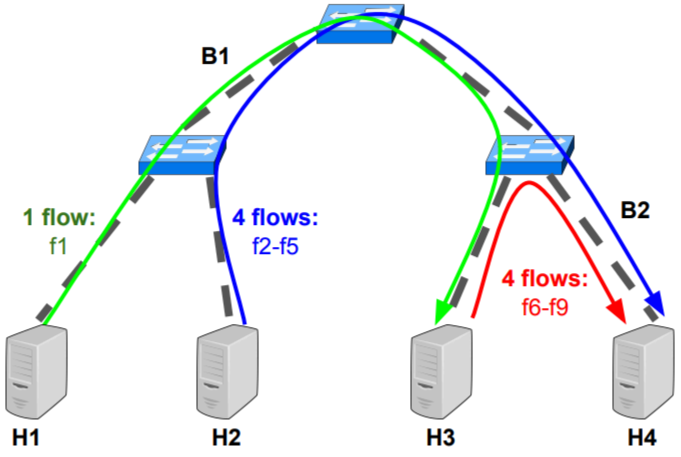
A Distributed Algorithm to Calculate Max–Min Fair Rates Without Per–Flow State

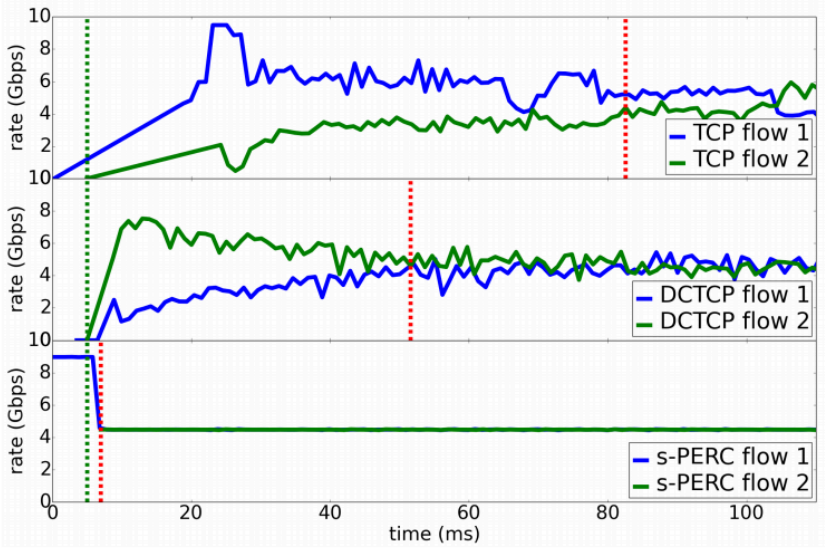
s-Perc is a congestion control algorithm that proactively assigns per-flow sending rates without per-flow state on devices.



Algorithm 4 s-PERC: link l processing control packets for flow f.
Differences from n-PERC are highlighted.

```
b, x, s : vector of bottleneck, allocated rates, bottleneck states in packet (initially, ∞, 0, E, respectively)
i: vector of ignore bits in packet (initially, 1)
$SumE$, $NumB$ : sum of limit rates of $E$ flows, and number of $B$ flows at link
$MaxE$, $MaxE'$: max. allocated rate of flows classified into $E$ since last round (and in this round, respectively) at link

if $s[l] = E$ then                                  ▷ Assume flow is not limited, for bottleneck rate calculation
    $s[l] \leftarrow B$
    $SumE \leftarrow SumE - x$
    $NumB \leftarrow NumB + 1$
$b \leftarrow (c - SumE)/NumB$
foreach link $j$:
 if $i[j] = 0$ then $p[j] \leftarrow b[j]$ else $p[j] \leftarrow \infty$                  ▷ Propagated rates
$p[l] \leftarrow \infty$                                   ▷ Assume the link's own propagated rate is ∞
$e \leftarrow \min p$
$x \leftarrow \min(b, e)$
$b[l] \leftarrow b$, $x[l] \leftarrow x$                            ▷ Save variables to packet
if $b < MaxE$ then $i[l] \leftarrow 1$ else $i[l] \leftarrow 0$          ▷ Indicate if rate b[l] should be ignored.
if flow is leaving then $NumB \leftarrow NumB - 1$                 ▷ Remove flow $f$
else if $e < b$ then
    $s[l] \leftarrow E$
    $SumE \leftarrow SumE + x$
    $NumB \leftarrow NumB - 1$
    $MaxE \leftarrow \max(x, MaxE)$; $MaxE' \leftarrow \max(x, MaxE')$.
```





In the control-plane:
Implement the s-Perc algorithm.

In the data-plane:
Create and parse control messages.

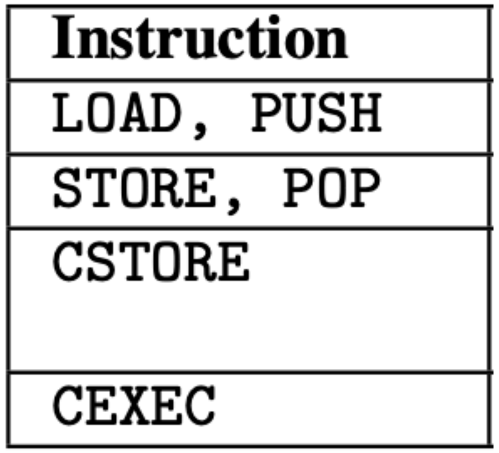
Evaluation:
Compare with TCP and other protocols.

[SIGMETRICS 2019] Stanford University, MIT CSAIL (Jose et. al.)

Proposal #7

Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility

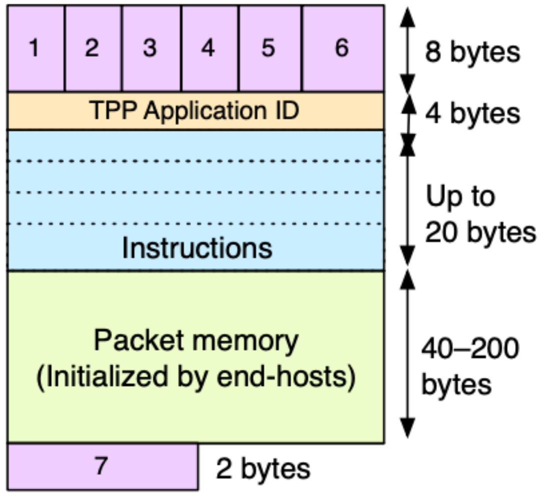
In active networks, packets carry programs, which are run by switches.



Instruction
LOAD, PUSH
STORE, POP
CSTORE
CEXEC

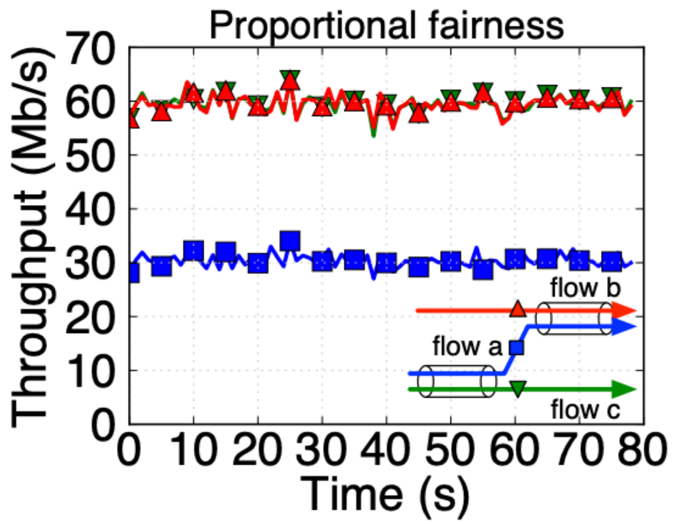
In the control-plane:

Compile and start packet programs.



In the data-plane:

Parse packets and execute instructions.



Evaluation:

Test the performance of packet programs.

[SIGCOMM 2014] Stanford University, Cisco, Barefoot (Jeyakumar et. al.)

Quick overview of the proposals

Albert

Thomas

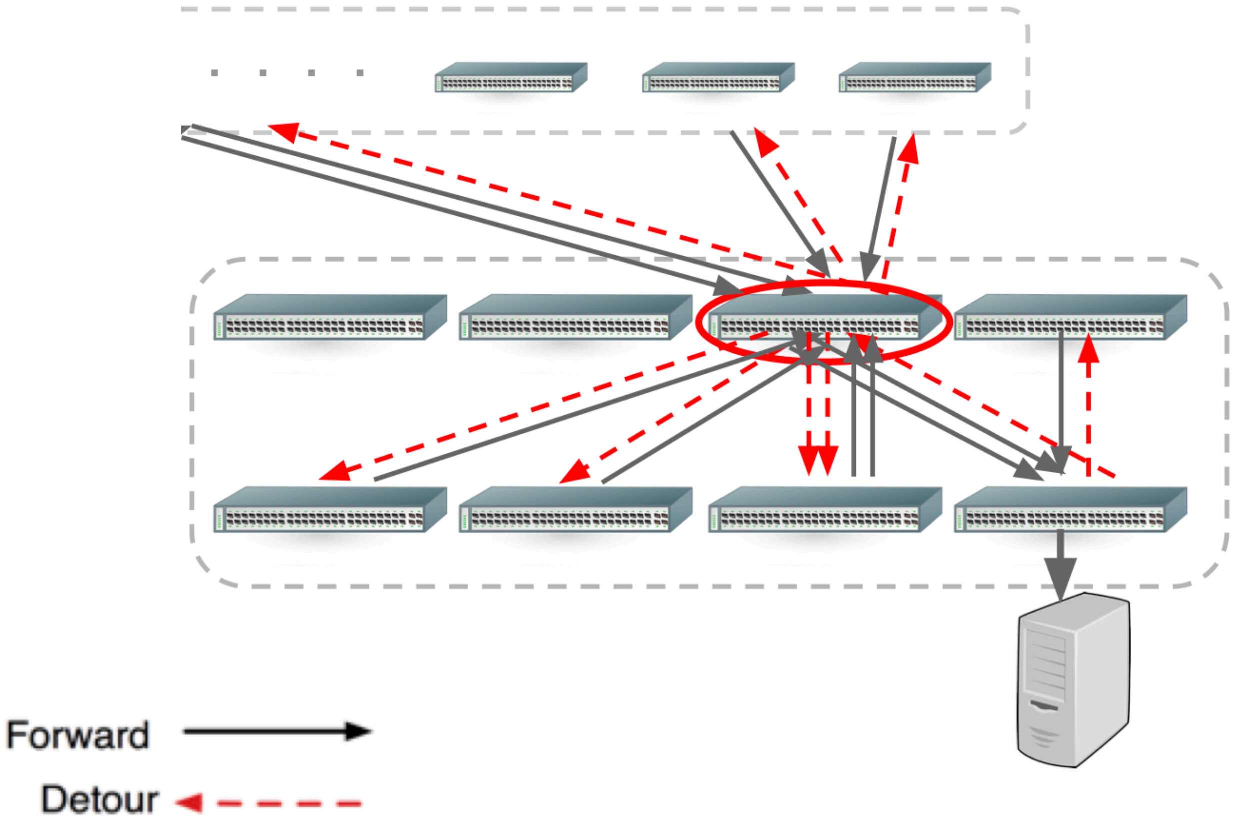
Roland

Alexander

Maria

Edgar

DIBS: Just-in-time congestion mitigation for Data Centers



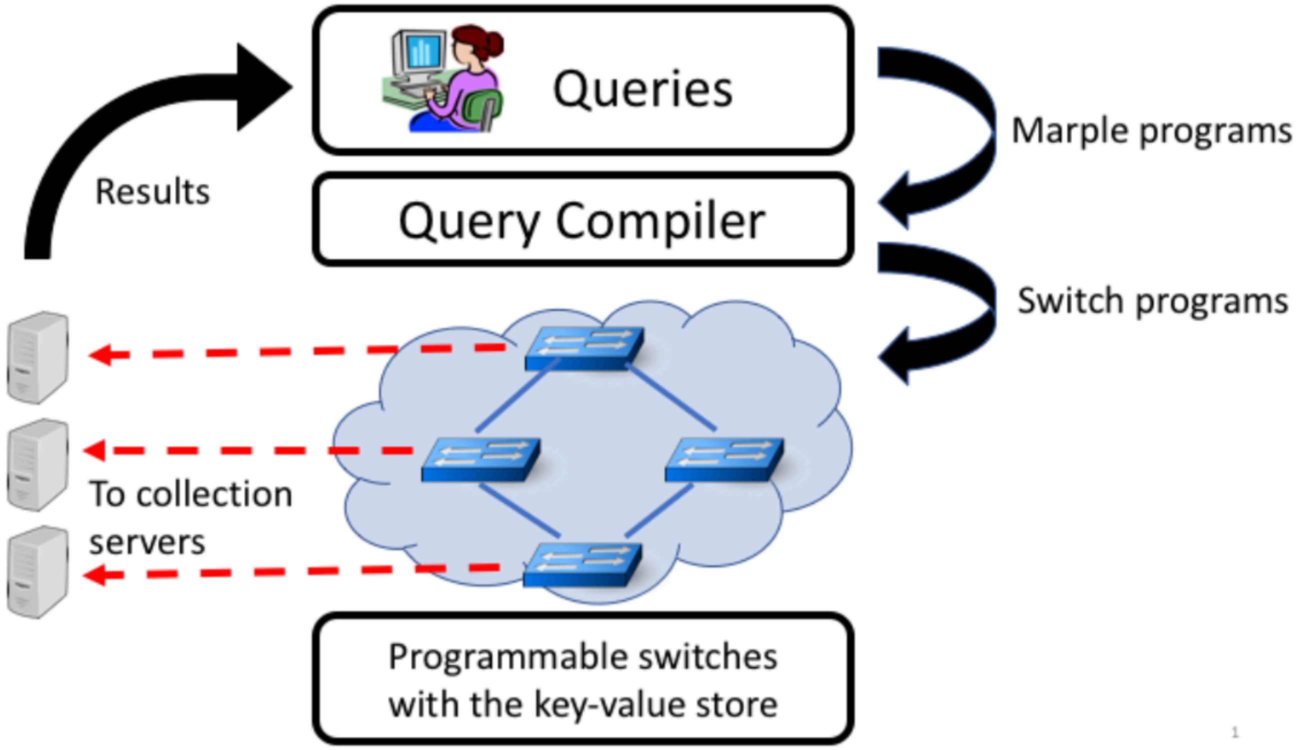
currently

- DC patterns can cause **congestion.**
- Switches drop packets they cannot buffer.

with DIBS

- **detours** to neighboring switches.
- minimizes drops, which speeds up **job completion time.**

Marple: Language-Directed Hardware Design for Network Performance Monitoring



- The operator writes a query in a domain-specific language called Marple.

- The query is compiled into a switch program that runs on the network's programmable switches, augmented with new switch hardware primitives that we design in service of Marple.

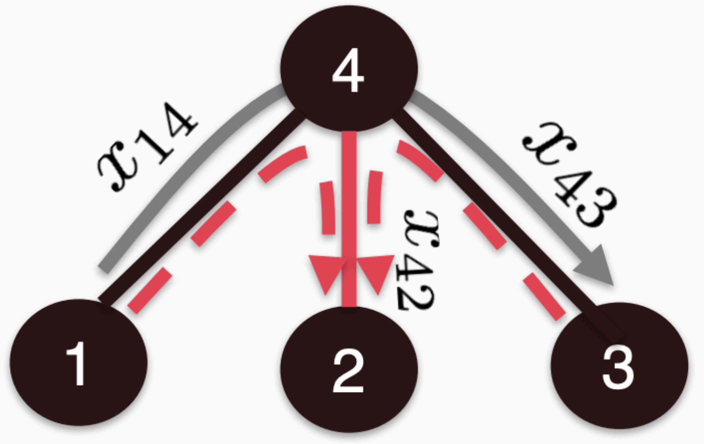
- The switches stream results out to collection servers, where the operator can retrieve query results.

007: Democratically Finding the Cause of Packet Drops

$$x_{14} + x_{43} = 0$$

$$x_{14} + x_{42} = 1$$

$$x_{34} + x_{42} = 1$$



$$x_{ij} \in \{0, 1\}$$

Need to detect short-lived & concurrent failures despite noise

- 007 scales by uses traceroute to find paths of flows that had packet drops
- 007 finds faulty links democratically democracy by letting hosts vote

Implementation with p4 switches.

- detect retransmissions in switches
- issue traceroutes directly from data plane
- combine traceroutes in control plane

Quick overview of the proposals

Albert

Thomas

Roland

Alexander

Maria

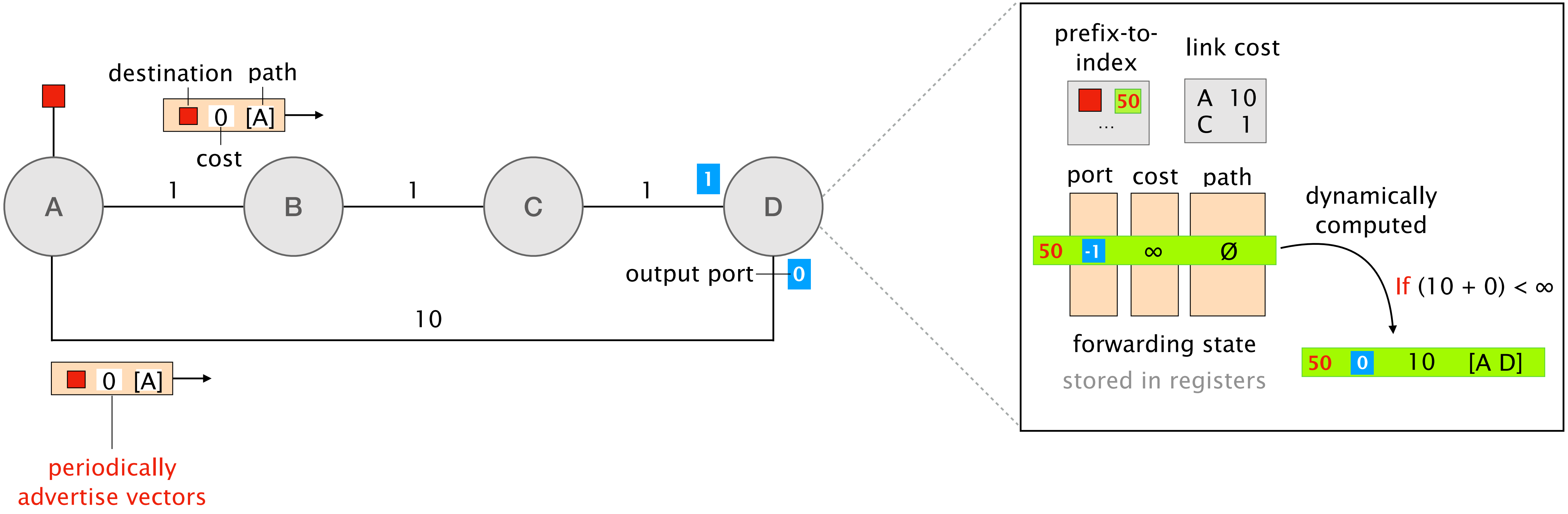
Edgar

Hardware-Accelerated Network Control Planes

[HotNets 2018] ETH, (Molero et. al.)

Modern programmable devices can perform small computations on billions of small packets per second.

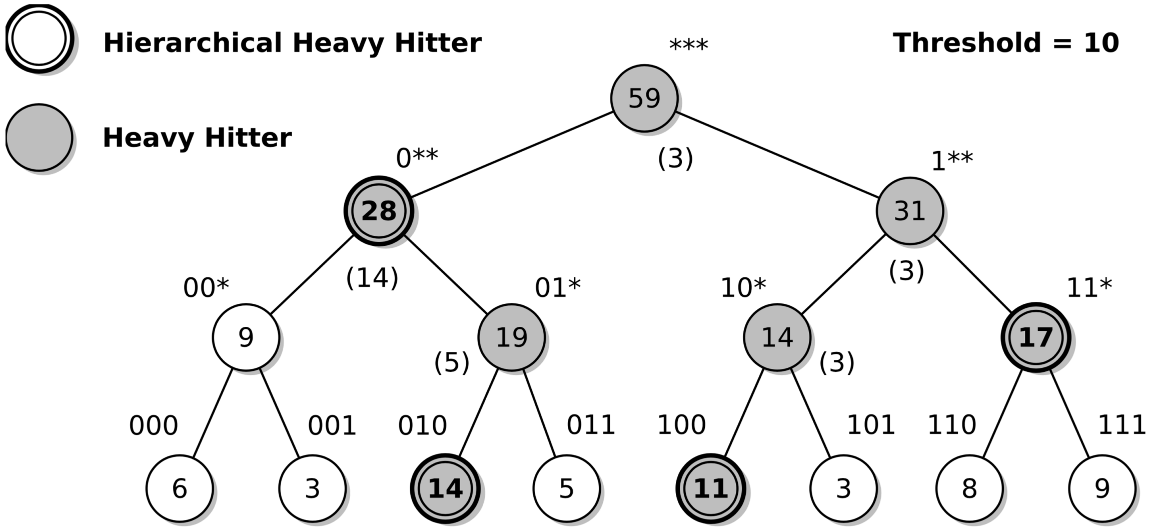
This paper shows how to leverage that to run control plane algorithms directly in the data plane



Seek and Push: Detecting Large Traffic Aggregates in the Dataplane

[arXiv 2018] CESNET, Cambridge (Kučera et. al.)

They present a data structure called *Elastic Tire* that is able to detect: heavy hitters, traffic shifts and superspreaders.



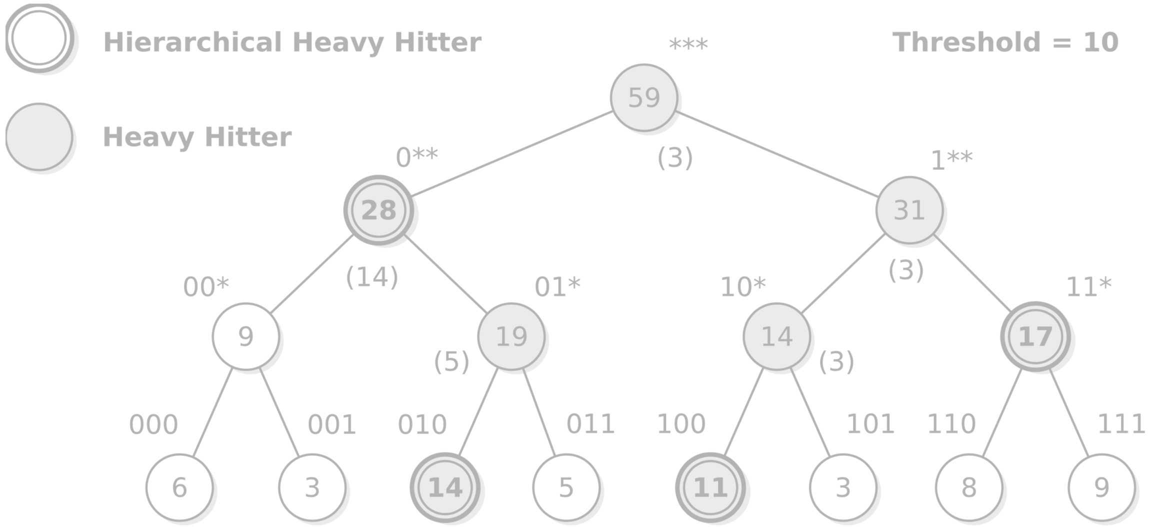
Seek and Push: Detecting Large Traffic Aggregates in the Dataplane

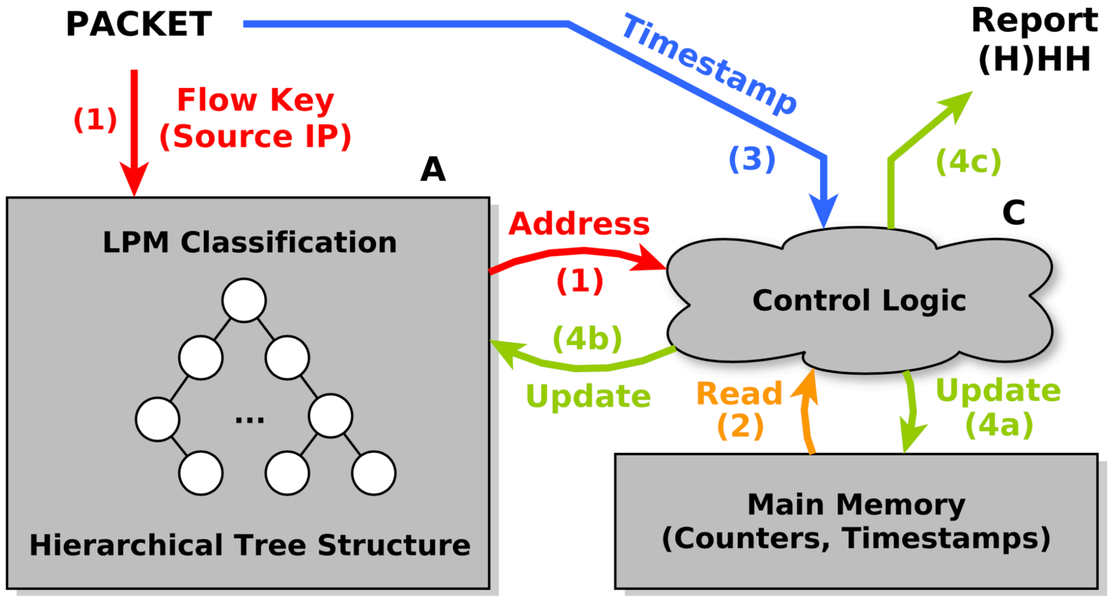
[arXiv 2018] CESNET, Cambridge (Kučera et. al.)

They present a data structure called *Elastic Tire* that is able to detect: heavy hitters, traffic shifts and superspreaders.

High-level architecture:

1. Matching the flow using a dynamic LPM tree
2. Update Statistics
3. Control logic to update or report

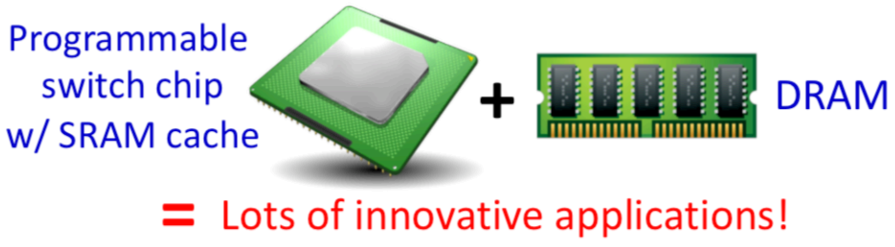




Generic External Memory for Switch Data Planes

[HotNets 2018] CMU, Microsoft, Barefoot Networks (Kim et. al.)

Programmable switches are flexible but only have a limited on-ship SRAM and TCAMS



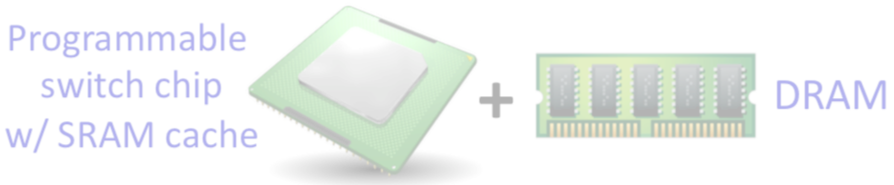
Programmable switch chip w/ SRAM cache + DRAM

= Lots of innovative applications!

Generic External Memory for Switch Data Planes

[HotNets 2018] CMU, Microsoft, Barefoot Networks (Kim et. al.)

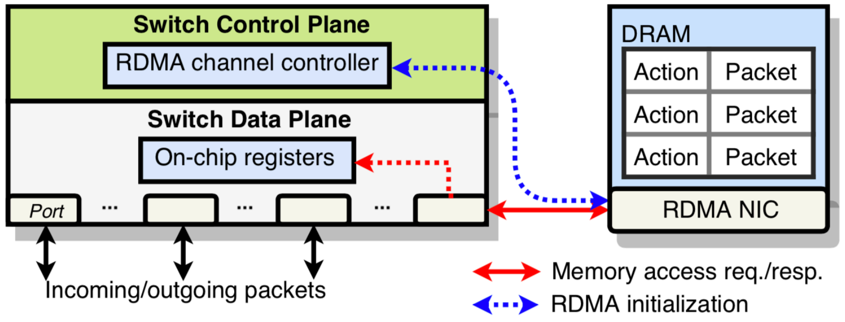
Programmable switches are flexible but only have a limited on-ship SRAM and TCAMS



Programmable switch chip w/ SRAM cache + DRAM

= Lots of innovative applications!

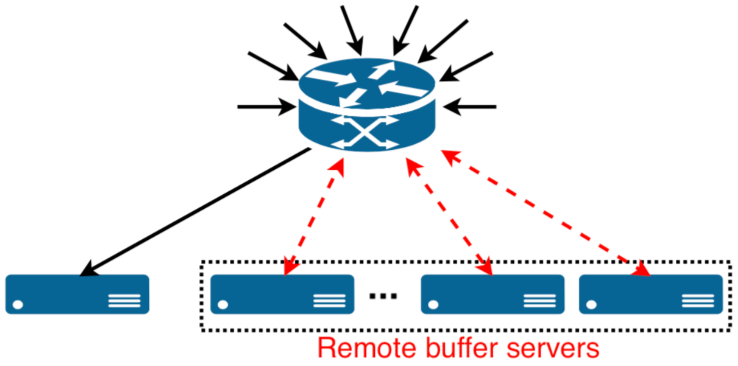
Leverage RDMA to access remote memories at minimal latency and CPU usage



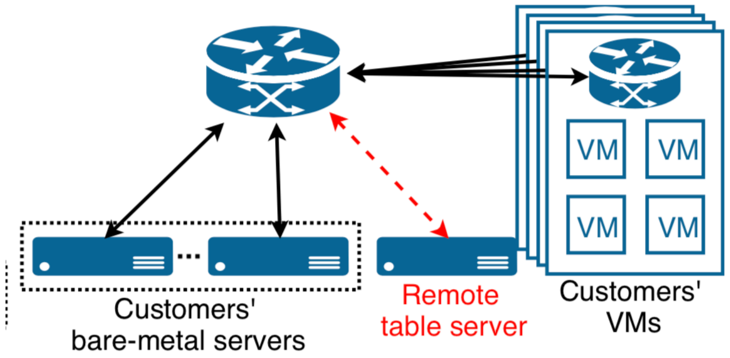
Generic External Memory for Switch Data Planes

[HotNets 2018] CMU, Microsoft, Barefoot Networks (Kim et. al.)

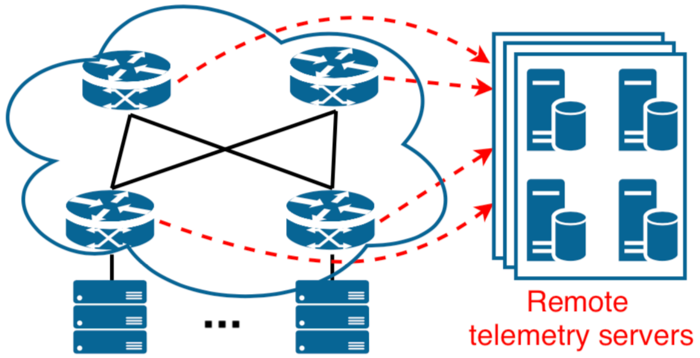
Packet buffer extension



Extending Lookup Tables



Extending State for network monitoring



Advanced Topics in Communication Networks

Programming Network Data Planes

Laurent Vanbever

nsg.ee.ethz.ch

ETH Zürich

Oct 29 2019