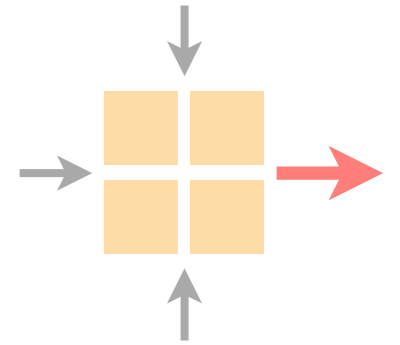


# Advanced Topics in Communication Networks

## Programming Network Data Planes



Laurent Vanbever

[nsg.ee.ethz.ch](http://nsg.ee.ethz.ch)

ETH Zürich

Oct 22 2019

Last week on

**Advanced Topics in Communication Networks**

*A bloom filter is a streaming algorithm*  
***answering specific questions approximately.***

*A bloom filter is a streaming algorithm*  
**answering *specific questions approximately.***

Is X in the stream?

What is in the stream? Invertible Bloom Filter

*A bloom filter is a streaming algorithm  
**answering specific questions approximately.***

|  
Is X in the stream?

What is in the stream? Invertible Bloom Filter

**What about other questions?**



*Is a certain flow in the stream?*

*Bloom Filter*

*What flows are in the stream?*

*Invertible Bloom Filter, HyperLogLog Sketch, ...*

***How frequently does an flow appear?***

*Count Sketch, CountMin Sketch, ...*

*What are the most frequent elements?*

*Count/CountMin + Heap, ...*

*How many flows belong to a certain subnet?*

*SketchLearn* SIGCOMM '18

In the worst case, an algorithm providing **exact frequencies** requires **linear space**.



*Data Stream*

**n** elements in total

→ **n distinct elements**

(in the worst case)

→ **n counters** required? :(

# Probabilistic datastructures can help again!

## **Bloom Filters**

*quickly “filter” only those elements that might be in the set*

*Save space by allowing false positives.*

## **Sketches**

*provide a approximate frequencies of elements in a data stream.*

*Save space by allowing mis-counting.*



*A **CountMin sketch** uses the same principles as a counting bloom filter, but is **designed** to have **provable L1 error bounds** for frequency queries.*

*A **CountMin sketch** uses the same principles as a counting bloom filter, but is **designed** to have **provable L1 error bounds** for frequency queries.*

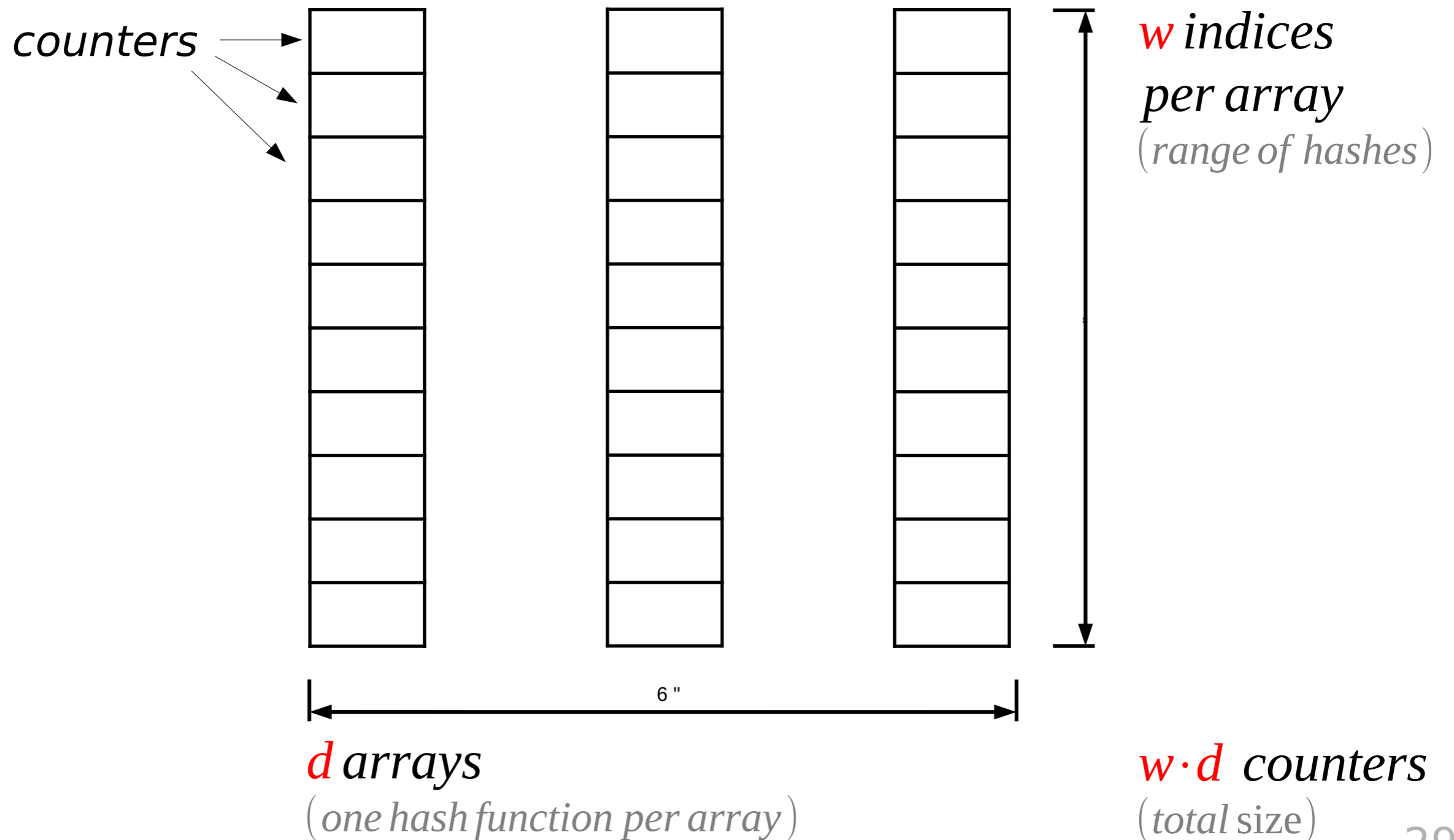
$$Pr \left[ \underset{\substack{\text{estimated} \\ \text{frequency}}}{\hat{x}_i} - \underset{\substack{\text{true} \\ \text{frequency}}}{x_i} \geq \underset{\substack{\text{sum of} \\ \text{frequencies}}}{\varepsilon \|\mathbf{x}\|_1} \right] \leq \delta$$

*relative to L1 norm*

The estimation error **exceeds**  $\varepsilon \|\mathbf{x}\|_1$   
 with a **probability smaller than**  $\delta$

*A **CountMin sketch** uses the same principles as a counting bloom filter, but is **designed** to have **provable L1 error bounds** for frequency queries.*

A **CountMin** Sketch uses multiple arrays and hashes.



A **CountMin sketch** uses the same principles as a counting bloom filter, but is **designed** to have **provable L1 error bounds** for frequency queries.

### **CountMin sketch recipe**

**Choose**  $d = \left\lceil \ln \frac{1}{\delta} \right\rceil$ ,  $w = \left\lceil \frac{e}{\epsilon} \right\rceil$

**Then**  $\hat{x}_i - x_i \geq \epsilon \|\mathbf{x}\|_1$  with a probability less than  $\delta$

*A **CountMin sketch** uses the same principles as a counting bloom filter, but is **designed** to have **provable L1 error bounds** for frequency queries.*

*→ only one design out of many!*

A **Count sketch** uses the same principles as a counting bloom filter, but is **designed** to have **provable L2 error bounds** for frequency queries.



The Count sketch uses **additional hashing** to give **L2 error bounds**, but requires more **resources**.

### *CountMin sketch recipe*

**Choose**  $d = \left\lceil \ln \frac{1}{\delta} \right\rceil, w = \left\lceil \frac{e}{\epsilon} \right\rceil$

**Then**  $\hat{x}_i - x_i \geq \epsilon \|\mathbf{x}\|_1$  with a probability less than  $\delta$

### *Count sketch recipe*

**Choose**  $d = \left\lceil \ln \frac{1}{\delta} \right\rceil, w = \left\lceil \frac{e}{\epsilon^2} \right\rceil$

**Then**  $\hat{x}_i - x_i \geq \epsilon \|\mathbf{x}\|_2$  with a probability less than  $\delta$

# Sketches are the new black

...and many more!

## OpenSketch

NSDI '13

[source]

**Software Defined Traffic Measurement with OpenSketch**

Minlan Yu<sup>†</sup> Lavanya Jose\* Rui Miao<sup>†</sup>  
<sup>†</sup>University of Southern California \*Princeton University

**Abstract**

Most network management tasks in software-defined networks (SDN) involve two stages: measurement and control. While many efforts have been focused on network control APIs for SDN, little attention goes into measurement. The key challenge of designing a new measurement API is to strike a careful balance between generality (supporting a wide variety of measurement tasks) and efficiency (enabling high link speed and low cost). We propose a software defined traffic measurement architecture OpenSketch, which separates the measurement data plane from the control plane. In the data plane, OpenSketch provides a simple three-stage pipeline (hashing, filtering, and counting), which can be implemented with commodity switch components and support many measurement tasks. In the control plane, OpenSketch provides a measurement library that automatically configures the pipeline and allocates resources for different measurement tasks. Our evaluations of real-world packet traces, our prototype on NetFlow, and the implementation of five measurement tasks on top of OpenSketch, demonstrate that OpenSketch is general, efficient and easily programmable.

**1 Introduction**

Recent advances in software-defined networking (SDN) have significantly improved network management. Network management involves two important stages: (1) measuring the network in real time (e.g., identifying traffic anomalies or large traffic aggregates) and then (2) adjusting the control of the network accordingly (e.g., routing, access control, and rate limiting). While there have been many efforts on designing the right APIs for network control (e.g., OpenFlow [29], ForCES [1], rule-based forwarding [33], etc.), little thought has gone into designing the right APIs for measurement. Since con-

work management, it is important to design and build a new software-defined measurement architecture. The key challenge is to strike a careful balance between generality (supporting a wide variety of measurement tasks) and efficiency (enabling high link speed and low cost).

Flow-based measurements such as NetFlow [2] and sFlow [42] provide generic support for different measurement tasks, but consume too resources (e.g., CPU, memory, bandwidth) [28, 18, 19]. For example, to identify the big flows whose byte volumes are above a threshold (i.e., heavy hitter detection which is important for traffic engineering in data centers [6]), NetFlow collects flow-level counts for sampled packets in the data plane. A high sampling rate would lead to too many counters, while a lower sampling rate may miss flows. While there are many NetFlow improvements for specific measurement tasks (e.g., [48, 19]), a different measurement task may need to focus on small flows (e.g., anomaly detection) and thus requiring another way of changing NetFlow. Instead, we should provide more customized and dynamic measurement data collection defined by the software written by operators based on the measurement requirements; and provide guarantees on the measurement accuracy.

As an alternative, many sketch-based streaming algorithms have been proposed in the theoretical research community [7, 12, 46, 8, 20, 47], which provide efficient measurement support for individual management tasks. However, these algorithms are not deployed in practice because of their lack of generality: Each of these algorithms answers just one question or produces just one statistic (e.g., the unique number of destinations), so it is too expensive for vendors to build new hardware to support each function. For example, the Space-Saving heavy hitter detection algorithm [8] maintains a hash table of items and counts, and requires customized operations such as keeping a pointer to the item with minimum counts and replacing the minimum-count entry with a

## UnivMon

SIGCOMM '16

[source]

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

Zaoxing Liu<sup>1</sup>, Antonis Manousis<sup>1</sup>, Gregory Vorsanger<sup>1</sup>, Vyas Sekar<sup>1</sup>, Vladimir Braverman<sup>1</sup>  
<sup>1</sup>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

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 to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permission from permissions@acm.org.

SIGCOMM '16, August 22–26, 2016, Florianoópolis, Brazil  
© 2016 ACM. ISBN 978-1-4503-4160-8. \$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

## SketchLearn

SIGCOMM '18

[source]

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

Qun Huang<sup>†</sup>, Patrick P. C. Lee<sup>‡</sup>, and Yungang Bao<sup>†</sup>  
<sup>†</sup>State Key Lab of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences  
<sup>‡</sup>Department of Computer Science and Engineering, The Chinese University of Hong Kong

**ABSTRACT**

Network measurement is challenged to fulfill stringent resource requirements in the face of massive network traffic. While approximate measurement can trade accuracy for resource savings, it demands intensive manual efforts to configure the right resource-accuracy trade-offs in real deployment. Such user burdens are caused by how existing approximate measurement approaches inherently deal with resource conflicts when tracking massive network traffic with limited resources. In particular, they tightly couple resource configurations with accuracy parameters, so as to provision sufficient resources to bound the measurement errors. We design SketchLearn, a novel sketch-based measurement framework that resolves resource conflicts by learning their statistical properties to eliminate conflicting traffic components. We prototype SketchLearn on OpenVSwitch and P4, and our testbed experiments and stress-test simulation show that SketchLearn accurately and automatically monitors various traffic statistics and effectively supports network-wide measurement with limited resources.

**CCS CONCEPTS**

•Networks → Network measurement;

**KEYWORDS**

Sketch; Network measurement

**ACM Reference Format:**

Qun Huang, Patrick P. C. Lee, and Yungang Bao. 2018. SketchLearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference. In SIGCOMM '18: ACM SIGCOMM 2018 Conference, August 20–25, 2018, Budapest, Hungary. ACM, New York, NY, USA, 17 pages. <https://doi.org/10.1145/3230543.3230559>

**1 INTRODUCTION**

Network measurement is indispensable to modern network management in clouds and data centers. Administrators measure a variety of traffic statistics, such as per-flow frequency, to infer the key behaviors or any unexpected patterns in operational networks. They use the measured traffic statistics to form the basis of management operations such as traffic engineering, performance diagnosis, and intrusion prevention. Unfortunately, measuring traffic statistics is non-trivial in the face of massive network traffic and large-scale network deployment. Error-free measurement requires per-flow tracking [15], yet today’s data center networks can have thousands of concurrent flows in a very small period from 50ms [2] down to even 5ns [56]. This would require tremendous resources for performing per-flow tracking.

In view of the resource constraints, many approaches in the literature leverage approximation techniques to trade between resource usage and measurement accuracy. Examples include sampling [9, 37, 64], top-*k* counting [5, 43, 44, 46], and sketch-based approaches [18, 33, 40, 42, 58], which we collectively refer to as *approximate measurement approaches*. Their idea is to construct compact sub-linear data structures to record traffic statistics, backed by theoretical guarantees on how to achieve accurate measurement with limited resources. Approximate measurement has formed building blocks in many state-of-the-art network-wide measurement systems (e.g., [32, 48, 55, 66, 62, 67]), and is also adopted in production data centers [31, 68].

Although theoretically sound, existing approximate measurement approaches are inconvenient for use. In such approaches, massive network traffic competes for the limited resources, thereby introducing measurement errors due to resource conflicts (e.g., multiple flows are mapped to the same counter in sketch-based measurement). To mitigate errors, sufficient resources must be provisioned in approximate measurement based on its theoretical guarantees. Thus, there exists a *tight binding between resource configurations and accuracy parameters*. Such tight binding leads to several practical limitations (see §2.2 for details): (i) administrators need

Today we'll talk about: important questions,  
how 'sketches' answer them,  
**limitations of 'sketches',**  
and my master thesis :)

Sketches **compute statistical summaries**, favoring elements with **high frequency**.

*Let*  $\varepsilon = 0.01$ ,  $\|\mathbf{x}\|_1 = 10000$  ( $\Rightarrow \varepsilon \cdot \|\mathbf{x}\|_1 = 100$ )

*Assume two flows*  $x_a$ ,  $x_b$ ,

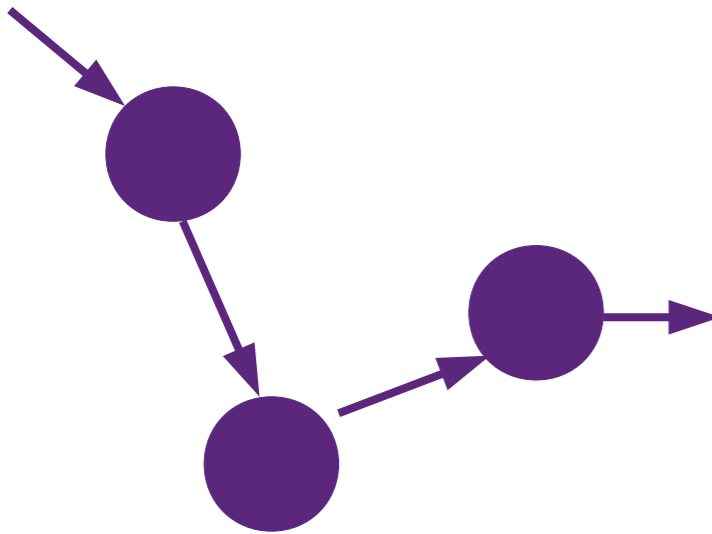
*with*  $\|x_a\|_1 = 1000$ ,  $\|x_b\|_1 = 50$

Error relative to **stream size**: 1%

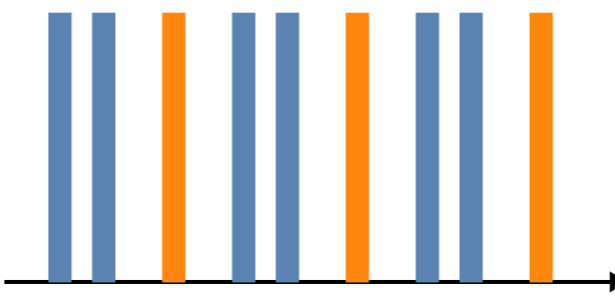
**flow size**:  $x_a$ : 10%,  $x_b$ : **200%**

# Other Problems a Sketch **can't** handle

*causality*



*patterns*



*rare things*



This week on

# Advanced Topics in Communication Networks

P4 hardware  
target

How do we build a *fast*  
reprogrammable switch?

P4-based  
applications

What cool things  
can we do with it?

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.”**

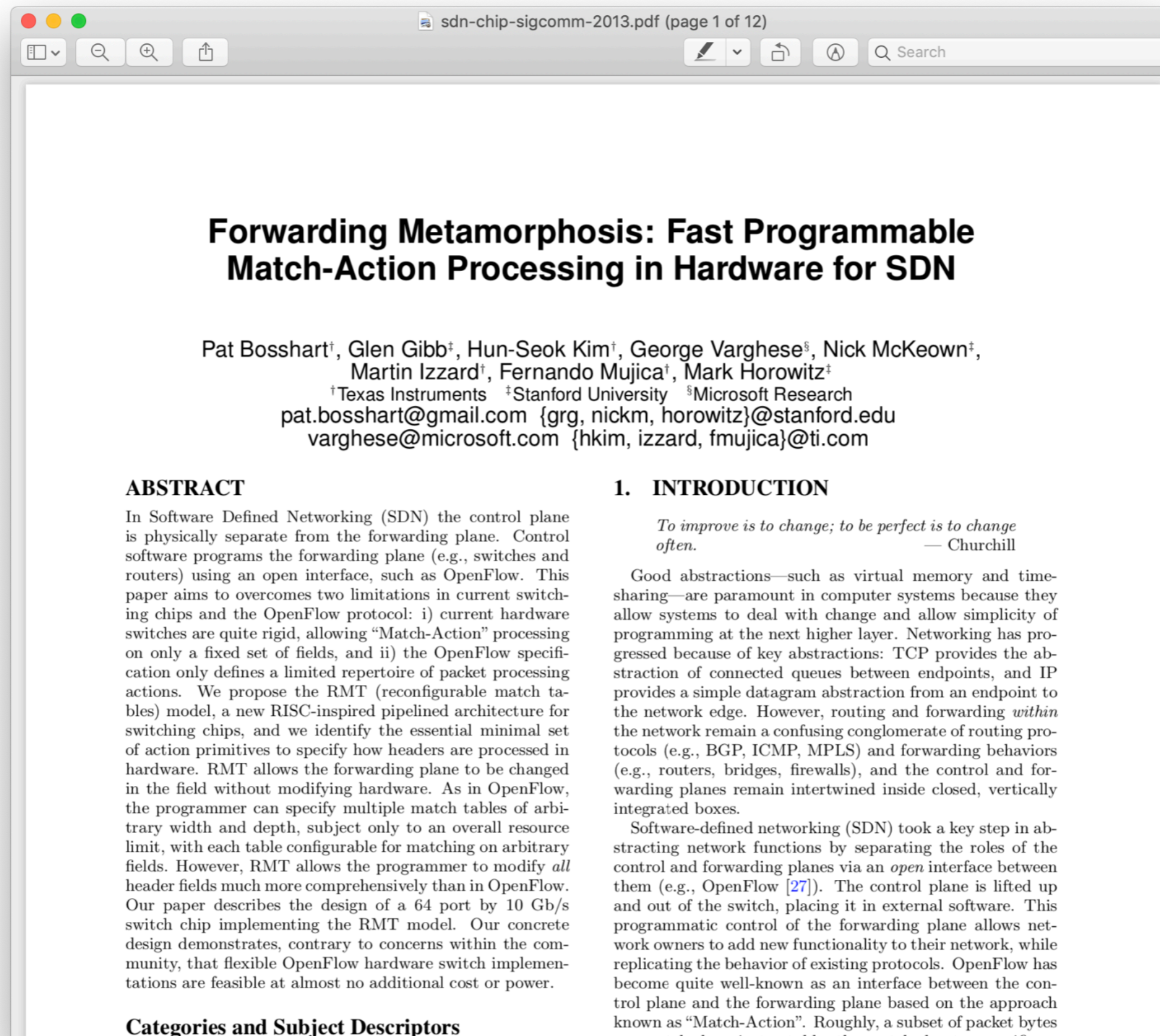
**Conventional wisdom in networking**



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)



sdn-chip-sigcomm-2013.pdf (page 1 of 12)

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

Pat Bosshart<sup>†</sup>, Glen Gibb<sup>‡</sup>, Hun-Seok Kim<sup>†</sup>, George Varghese<sup>§</sup>, Nick McKeown<sup>‡</sup>,  
Martin Izzard<sup>†</sup>, Fernando Mujica<sup>†</sup>, Mark Horowitz<sup>‡</sup>  
<sup>†</sup>Texas Instruments <sup>‡</sup>Stanford University <sup>§</sup>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 overcome 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.

### 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

### Categories and Subject Descriptors

[SIGCOMM'13]

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

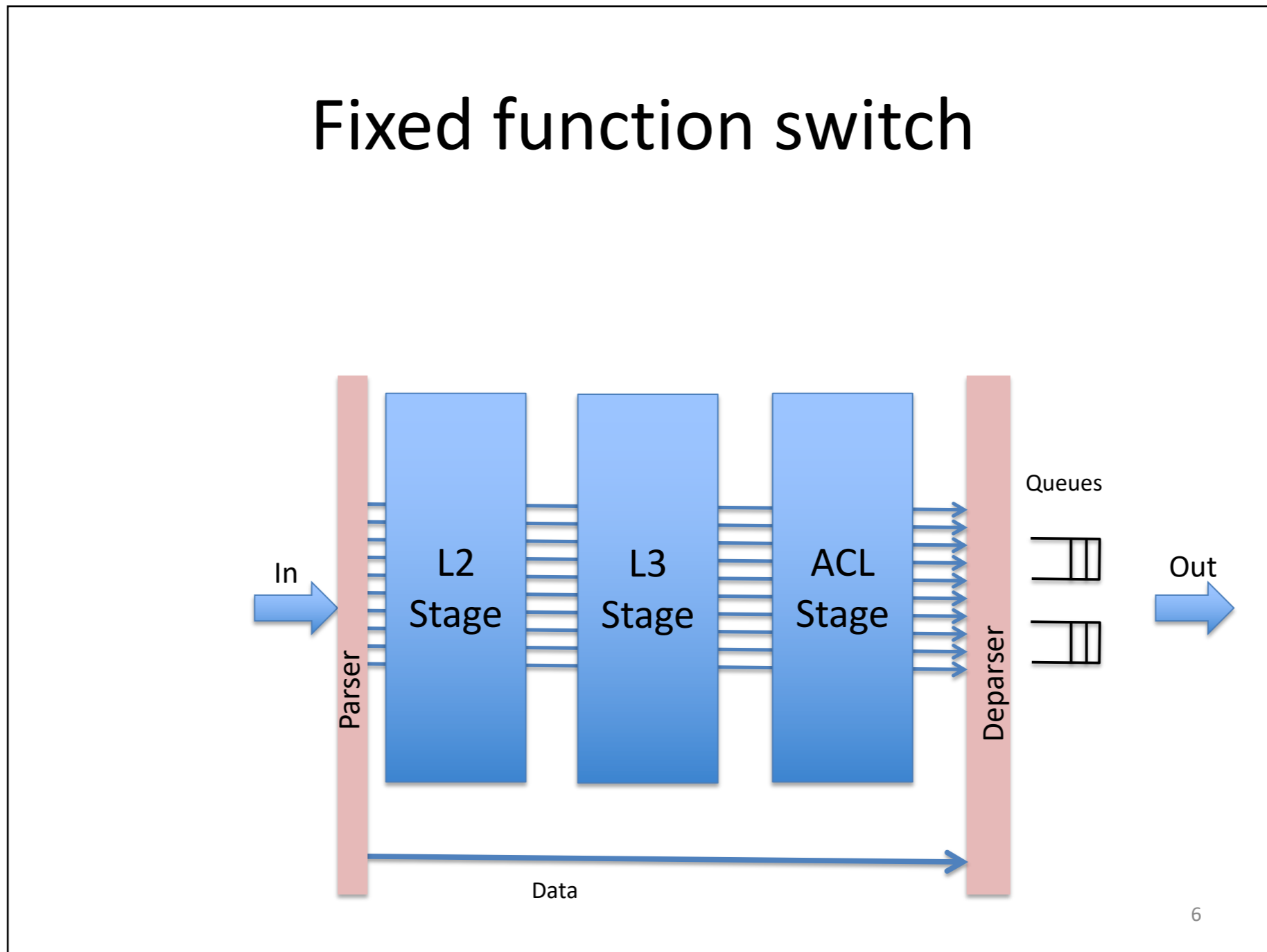
Texas Instruments, Stanford University, Microsoft

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

## Outline

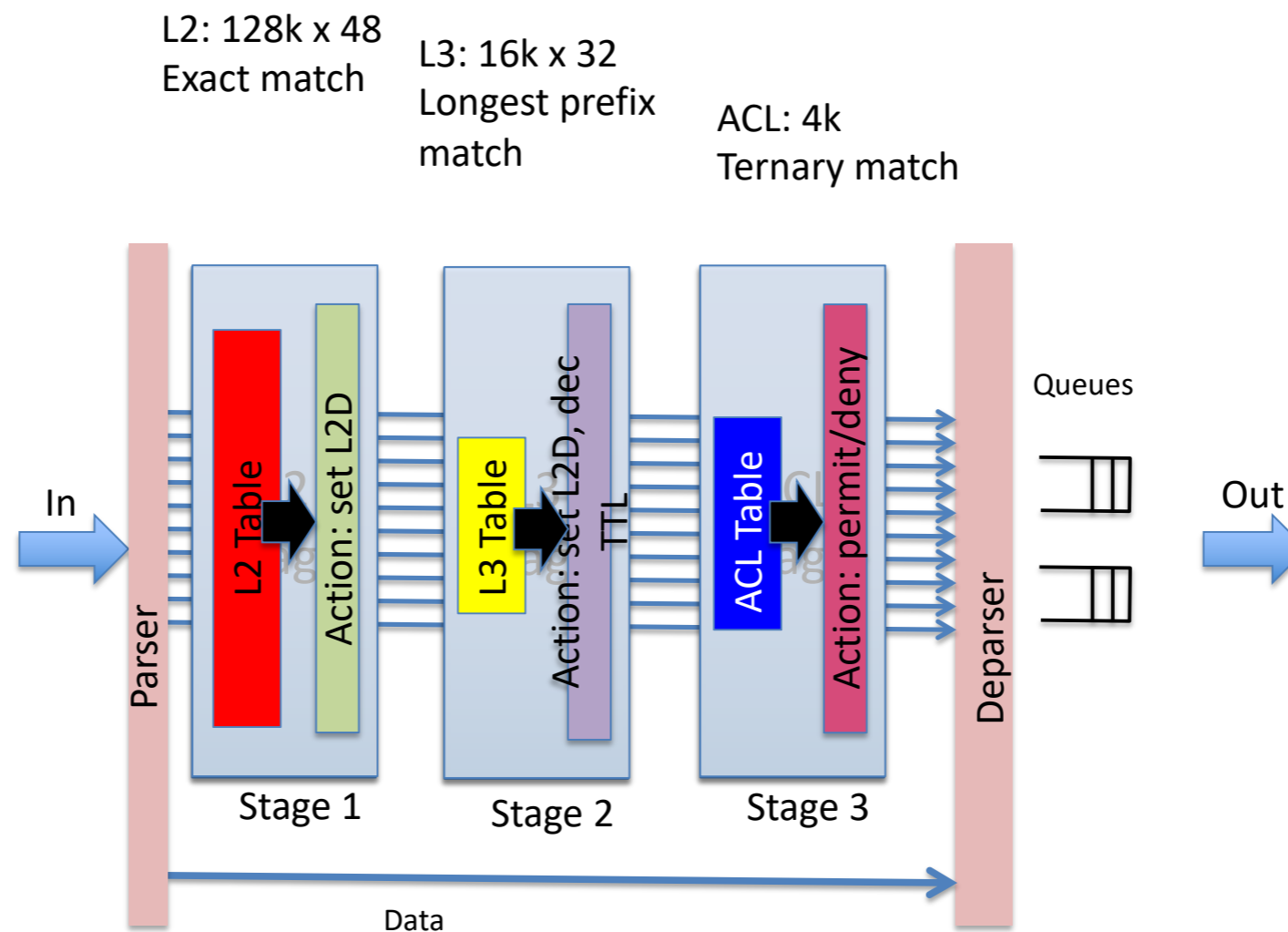
- Conventional switch chips are inflexible
- SDN demands flexibility...sounds expensive...
- How do we do it: The RMT switch model
- Flexibility costs less than 15%

Let's look first at a fixed-function switch composed of a (de-)parser and a sequence of processing stages

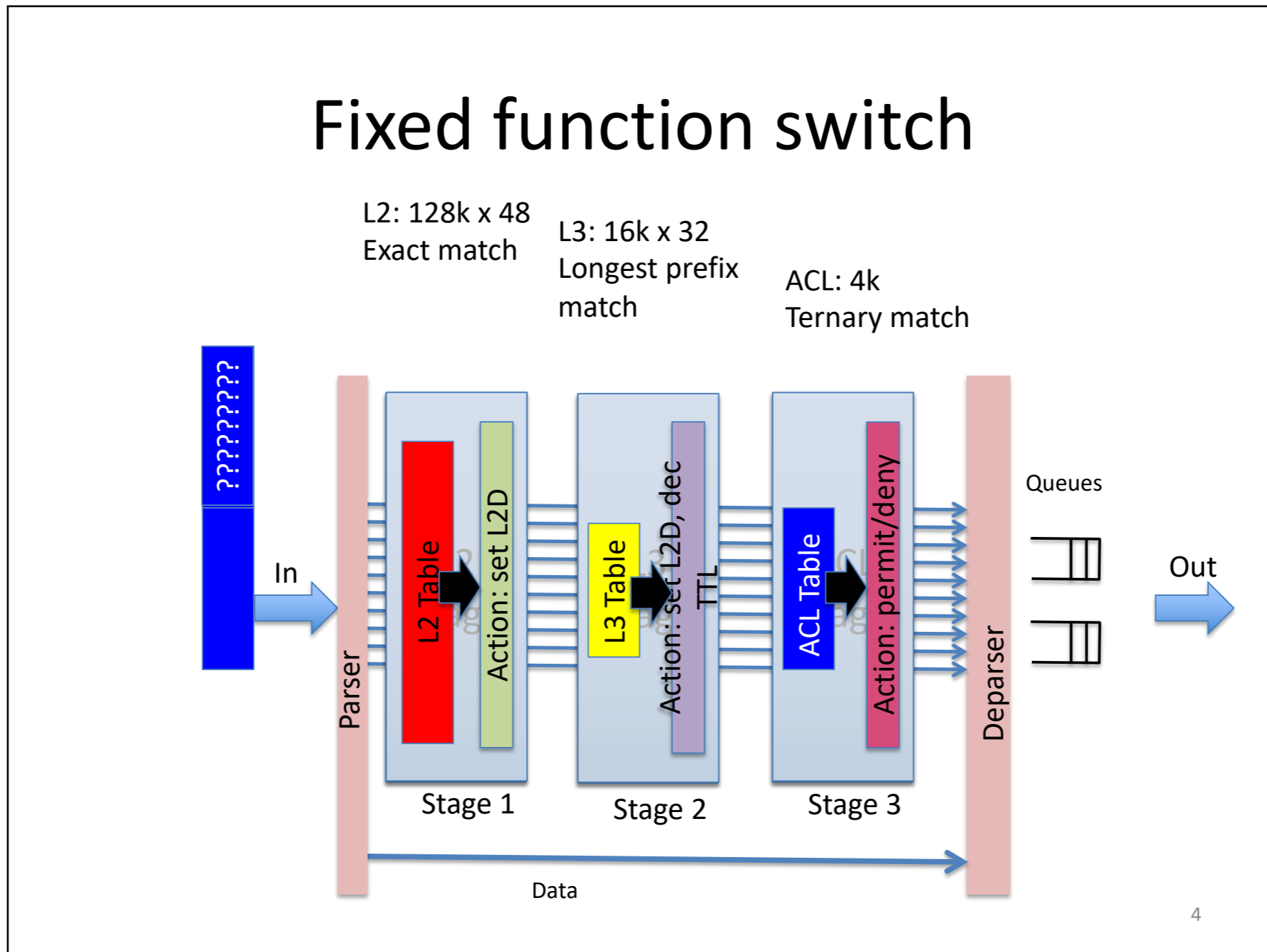


In such a switch,  
each stage is particularized to its usage

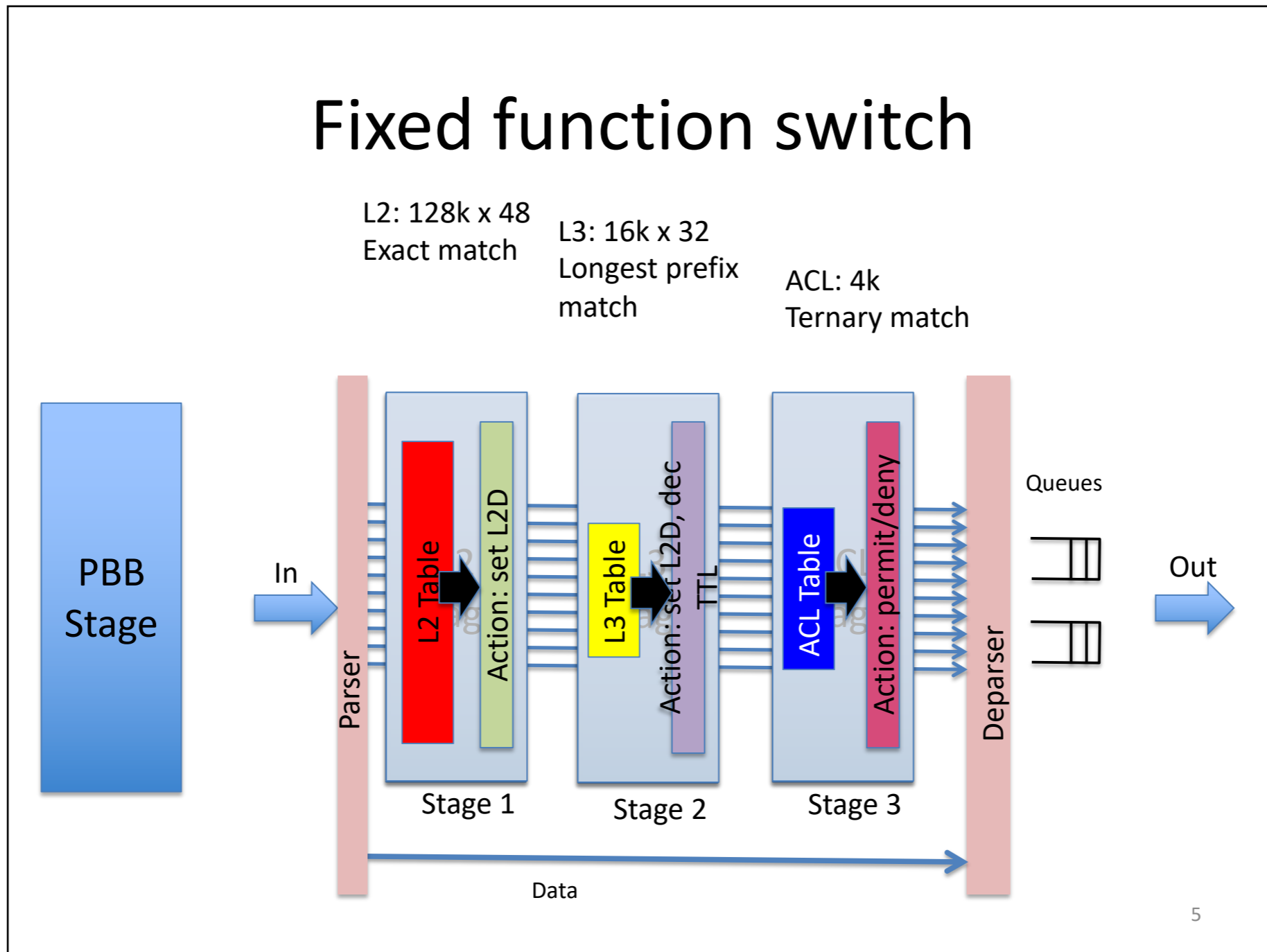
## Fixed function switch



This specificity makes it impossible to...  
trade memory size for another

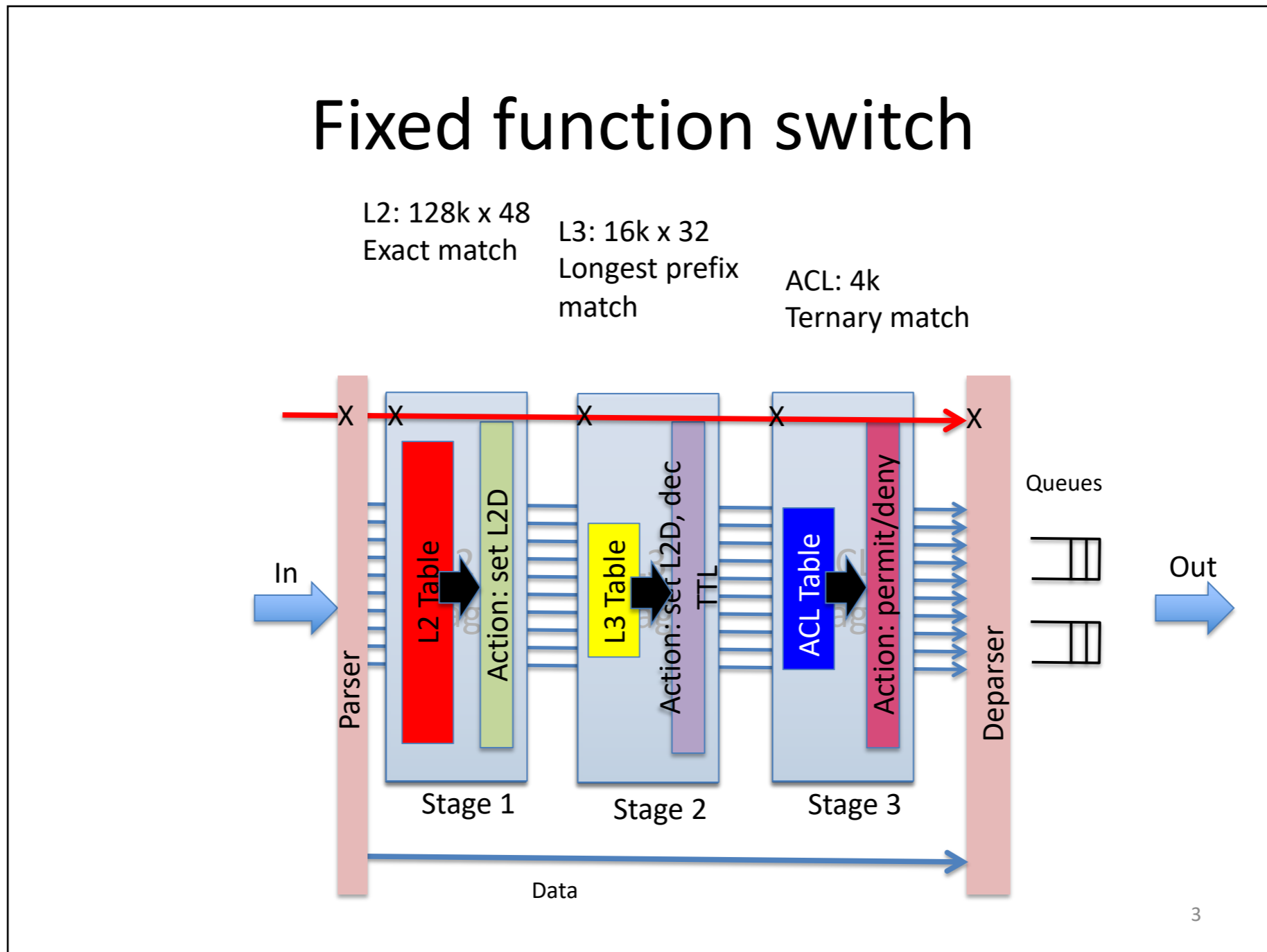


This specificity makes it impossible to...  
add a new table





This specificity makes it impossible to...  
support new headers or new actions



# What if you need flexibility?

- Flexibility to:
  - Trade one memory size for another
  - Add a new table
  - Add a new header field
  - Add a different action
- SDN accentuates the need for flexibility
  - Gives programmatic control to control plane, expects to be able to use flexibility

## What does SDN want?

- Multiple stages of match-action
  - Flexible allocation
- Flexible actions
- Flexible header fields
- No coincidence OpenFlow built this way...

Alternative ways to enable flexibility don't compare in terms of cost-performance ratio

What about Alternatives?  
Aren't there other ways to get flexibility?

- Software? 100x too slow, expensive
- NPUs? 10x too slow, expensive
- FPGAs? 10x too slow, expensive

## What We Set Out To Learn

- How do I design a flexible switch chip?
- What does the flexibility cost?

Unsurprisingly...

building flexible switching chipset *is* challenging

## What's Hard about a Flexible Switch Chip?

- Big chip
- High frequency
- Wiring intensive
- Many crossbars
- Lots of TCAM
- Interaction between physical design and architecture
- Good news? No need to read 7000 IETF RFC's!

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

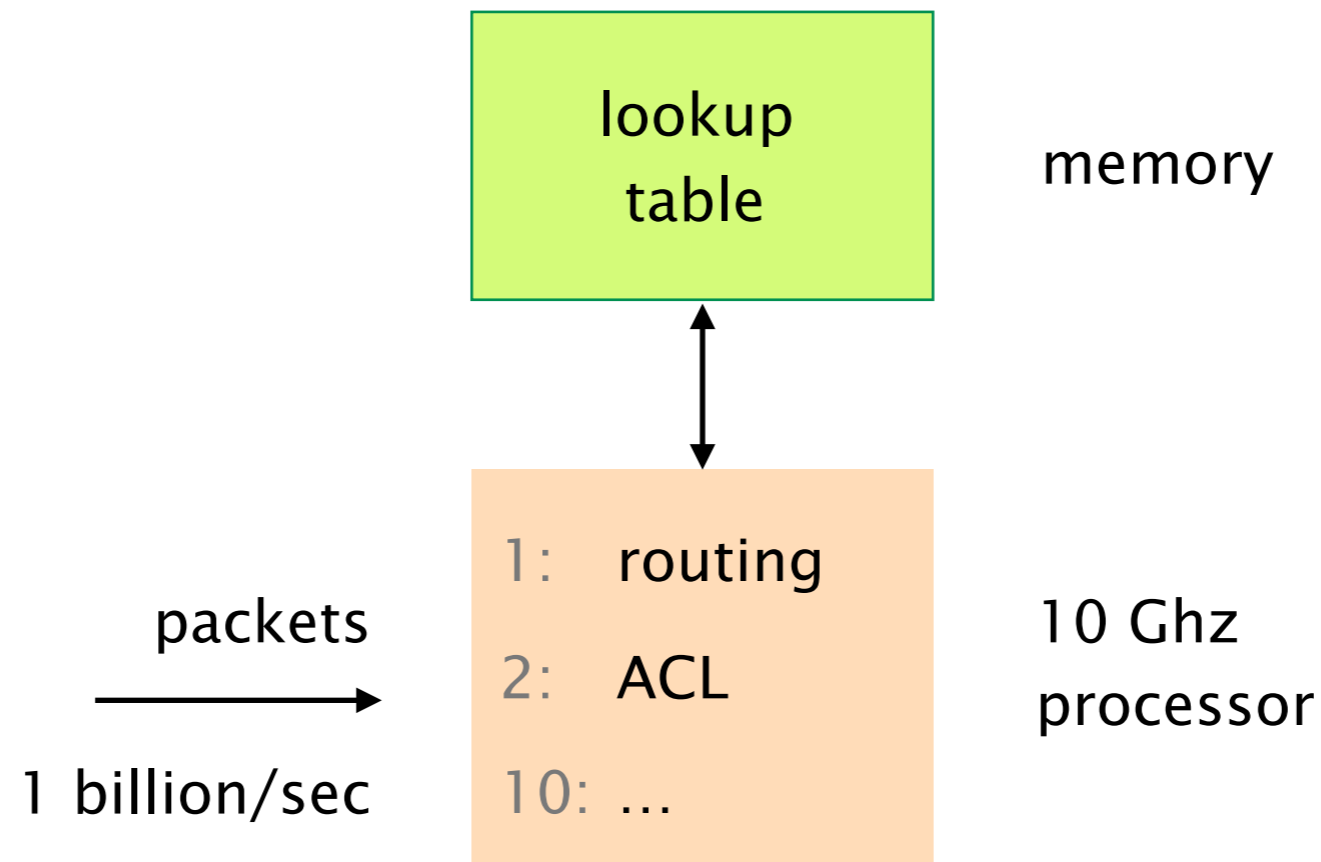
Requirements

10 billion op./second



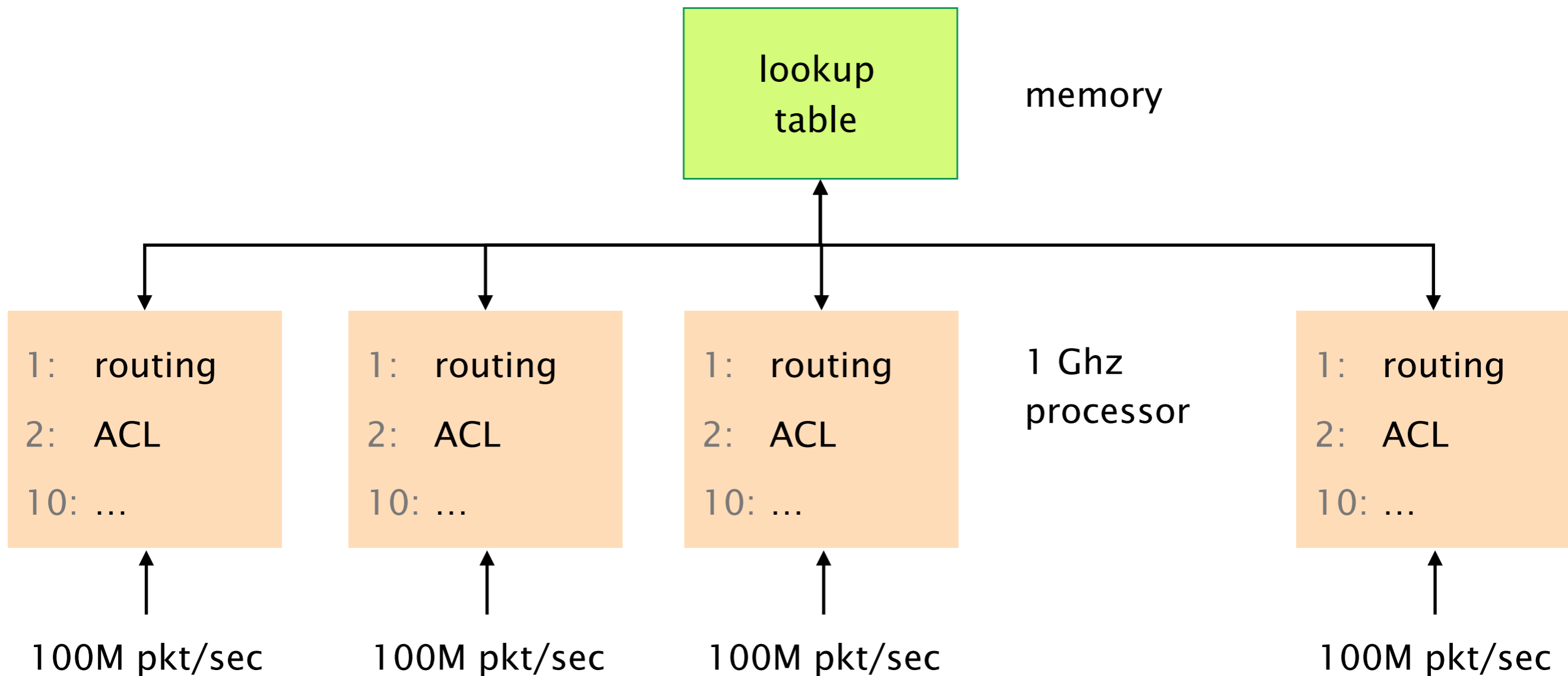
If our switch has a single processor,  
this would require us to run it at **10 Ghz...**

not feasible

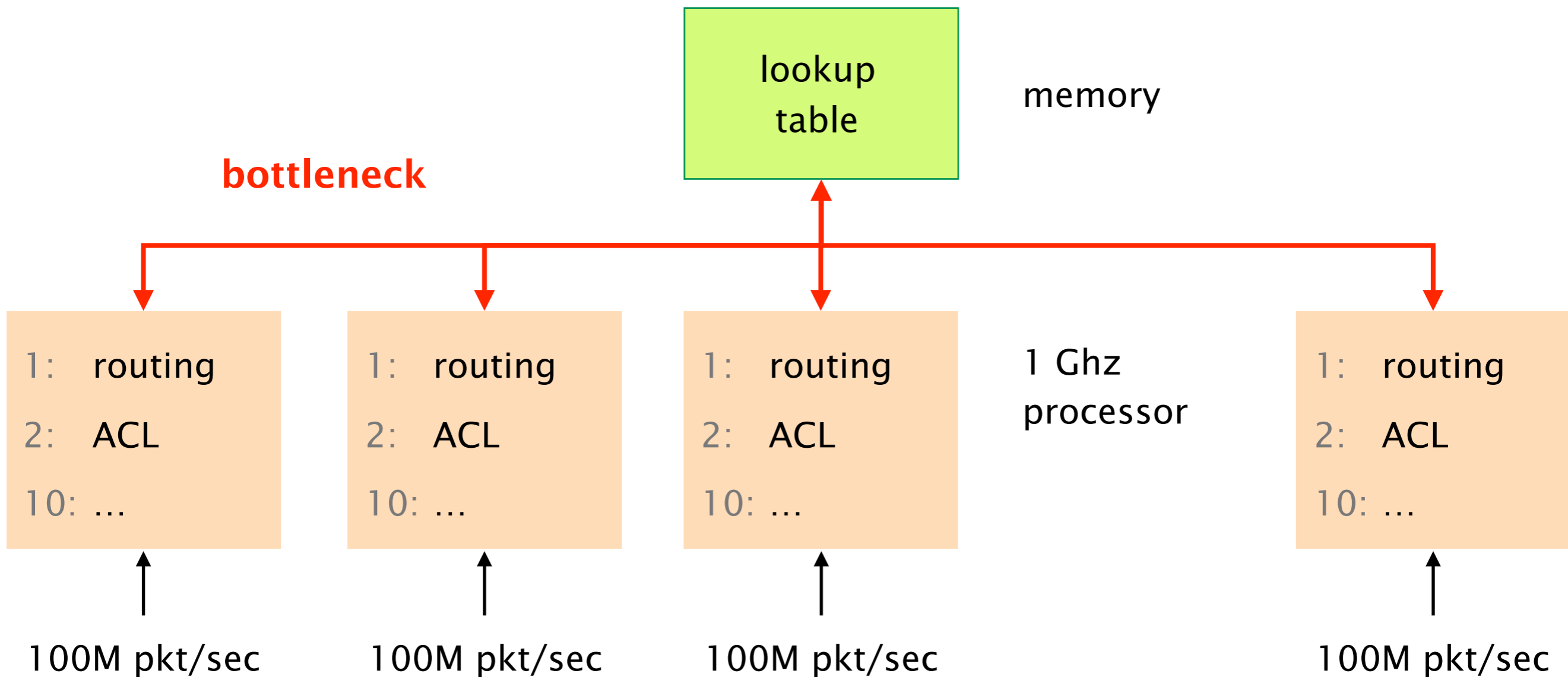


Let's parallelize things with a  
**packet-parallel architecture**

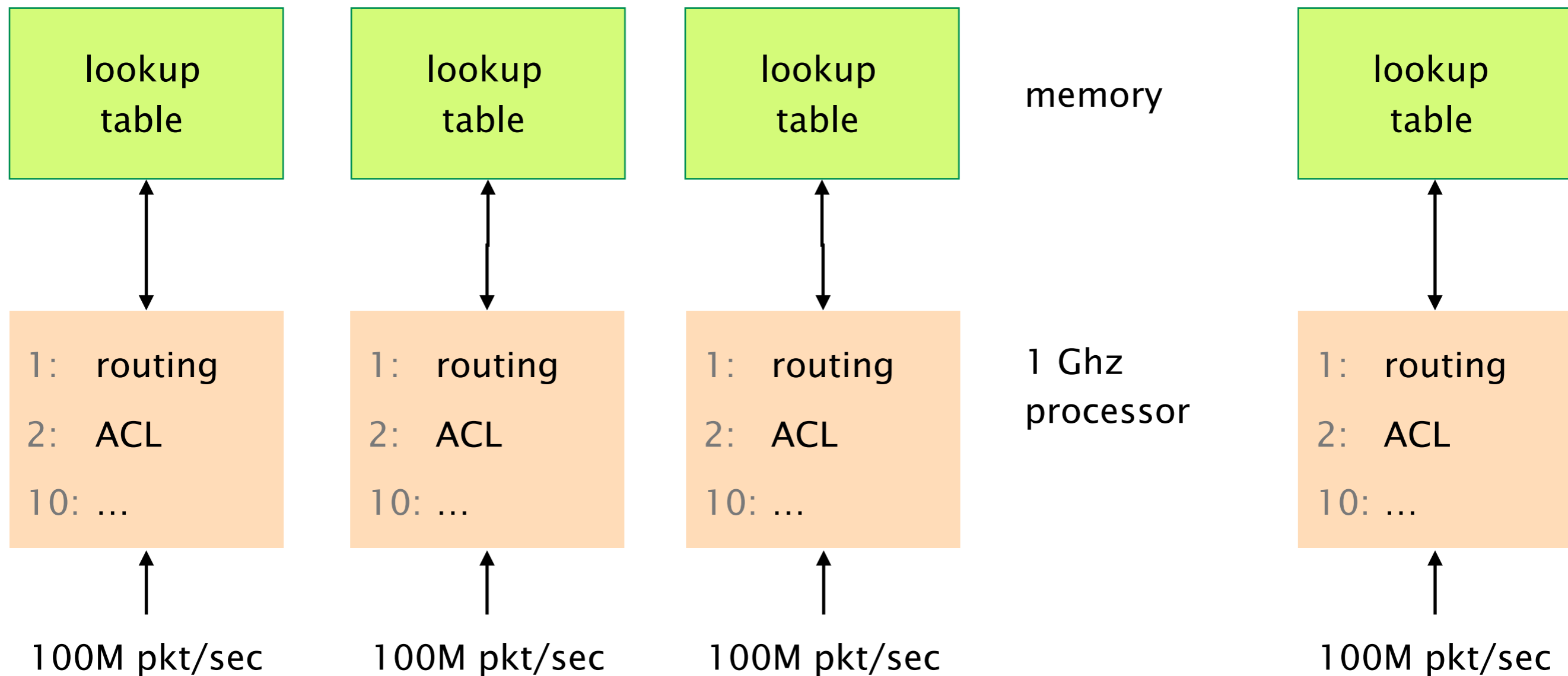
What about we duplicate the processing units?  
Each of which clocked at 1 Ghz



One issue though is to scale  
the memory-to-CPU bandwidth

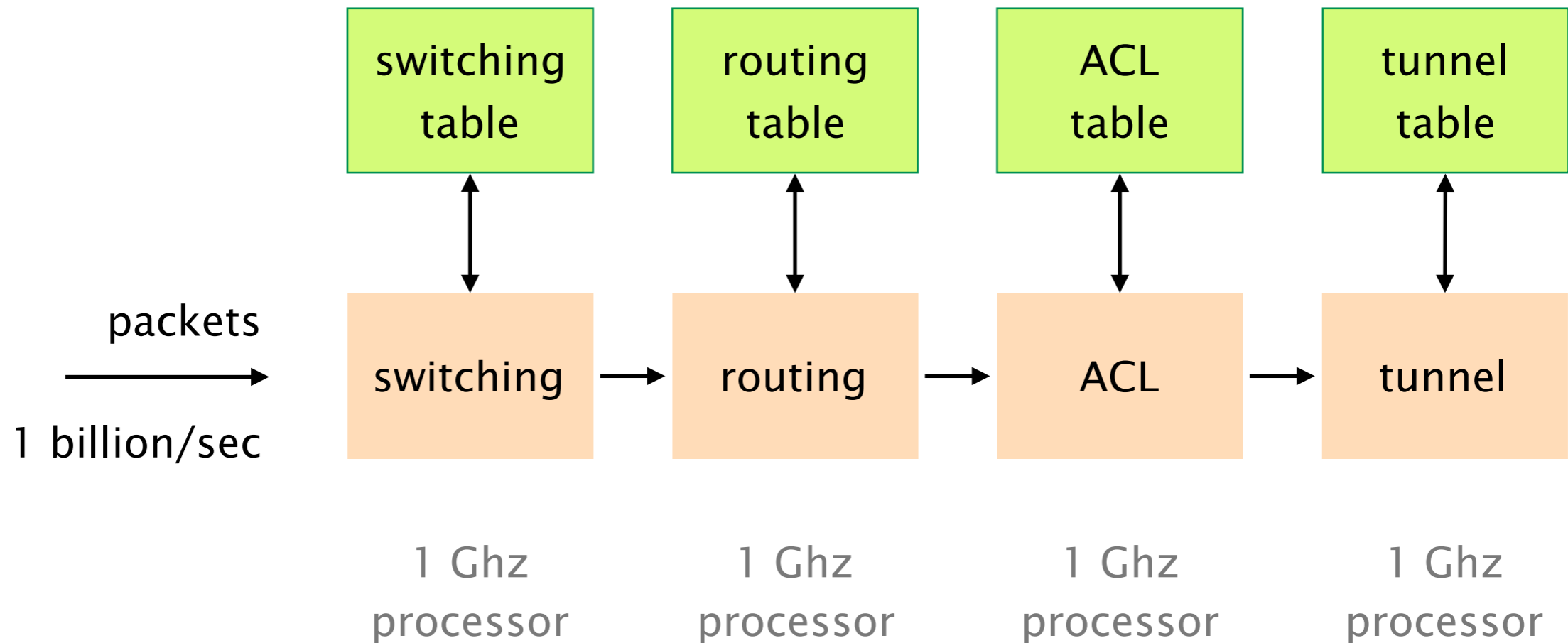


We could replicate the memory of course...  
but that comes at **a huge costs in die area**

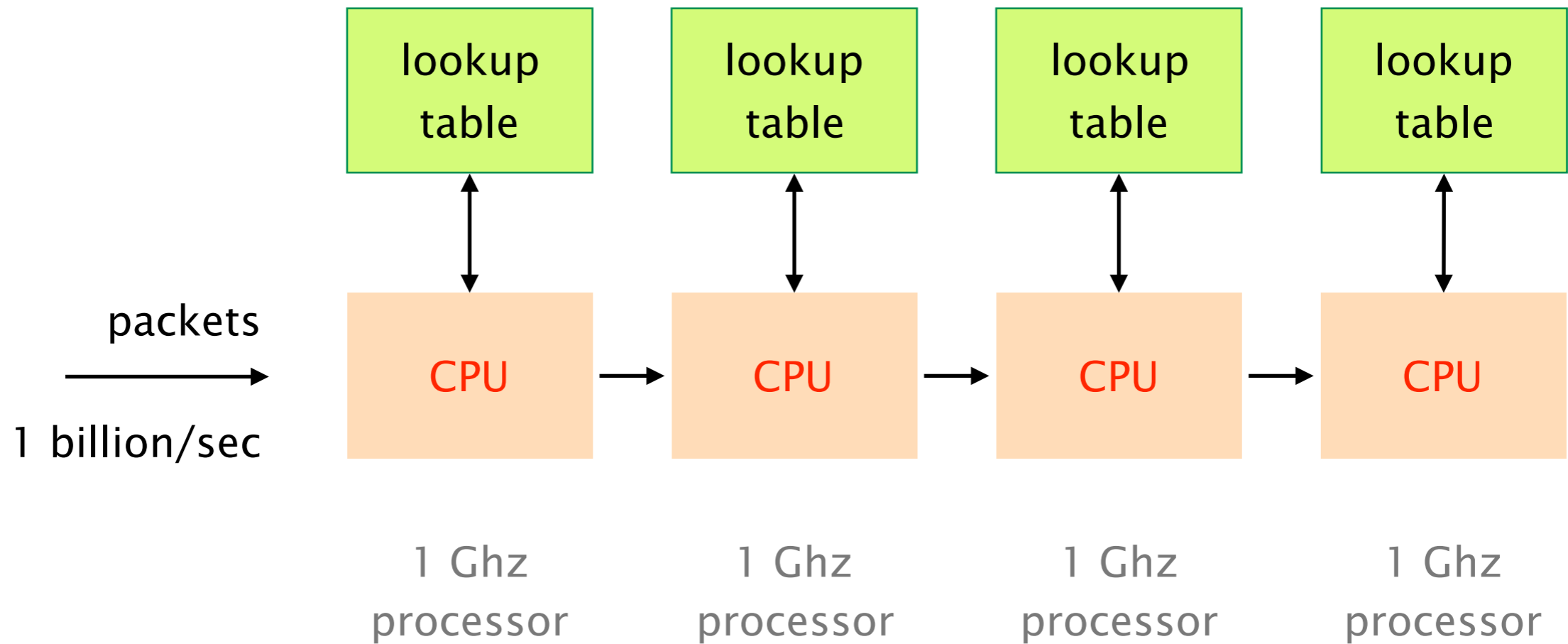


What if we organize the processing  
as a **pipeline** instead?

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

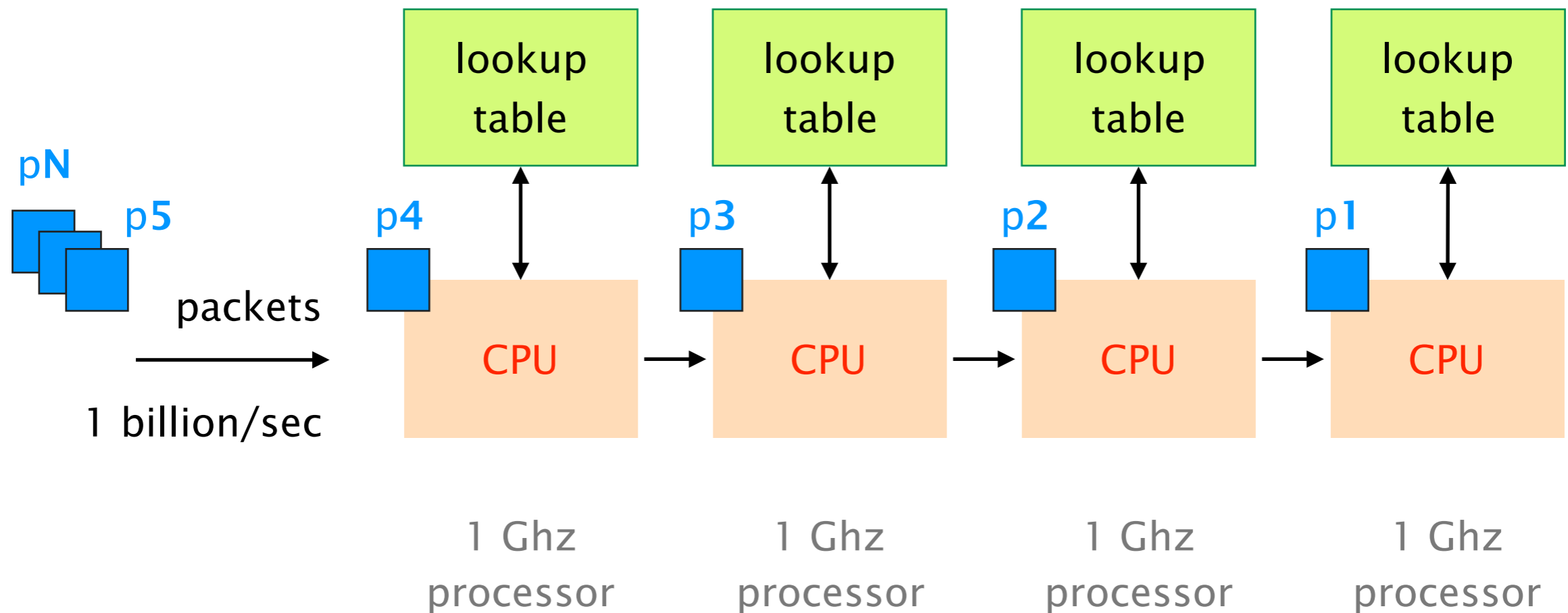


For flexibility,  
each processing unit/memory can be made generic





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



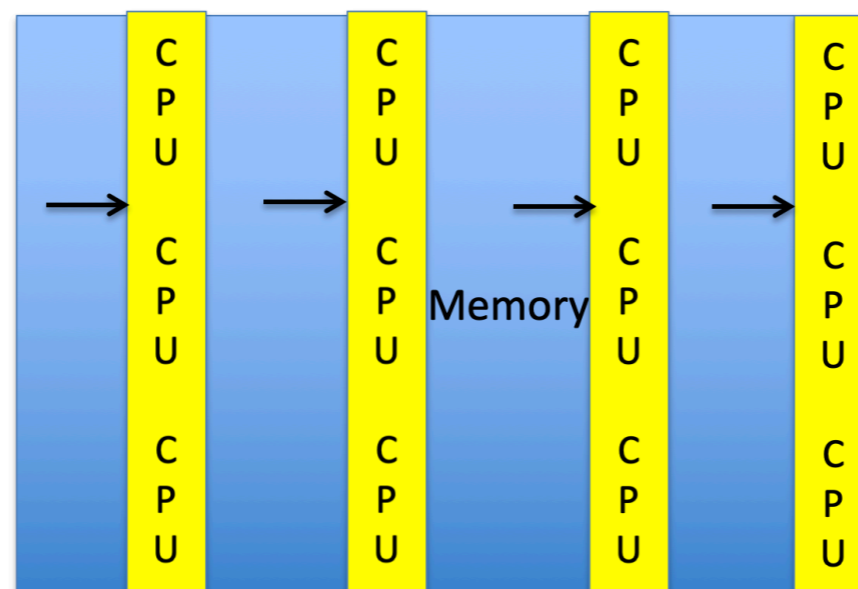
# Performance vs Flexibility

- Multiprocessor: memory bottleneck
- Change to pipeline
- Fixed function chips specialize processors
- Flexible switch needs general purpose CPUs

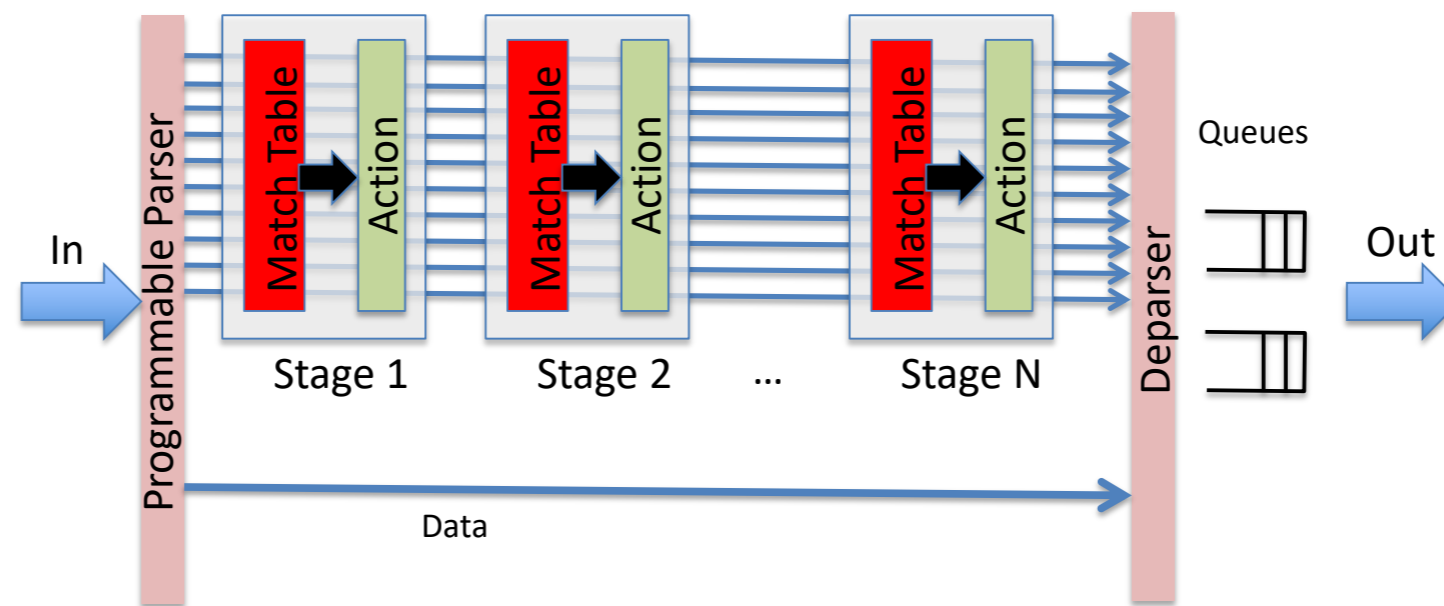


## How We Did It

- Memory to CPU bottleneck
- Replicate CPUs
- More stages for finer granularity
- Higher CPU cost ok



# Match/Action Forwarding Model



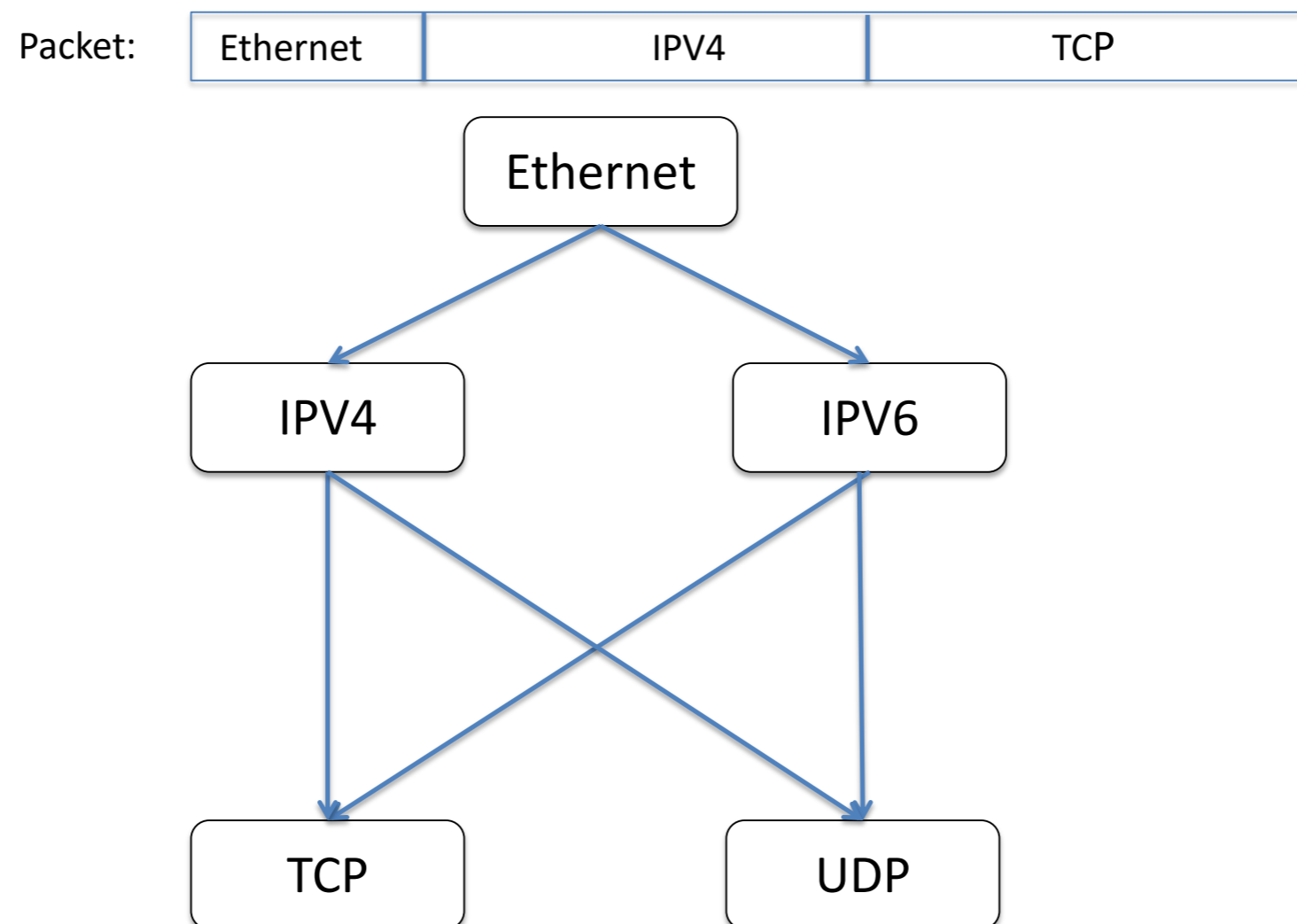
The runtime behavior of the parser & the match stages is defined through the RMT abstract model

## The RMT Abstract Model

- Parse graph
- Table graph

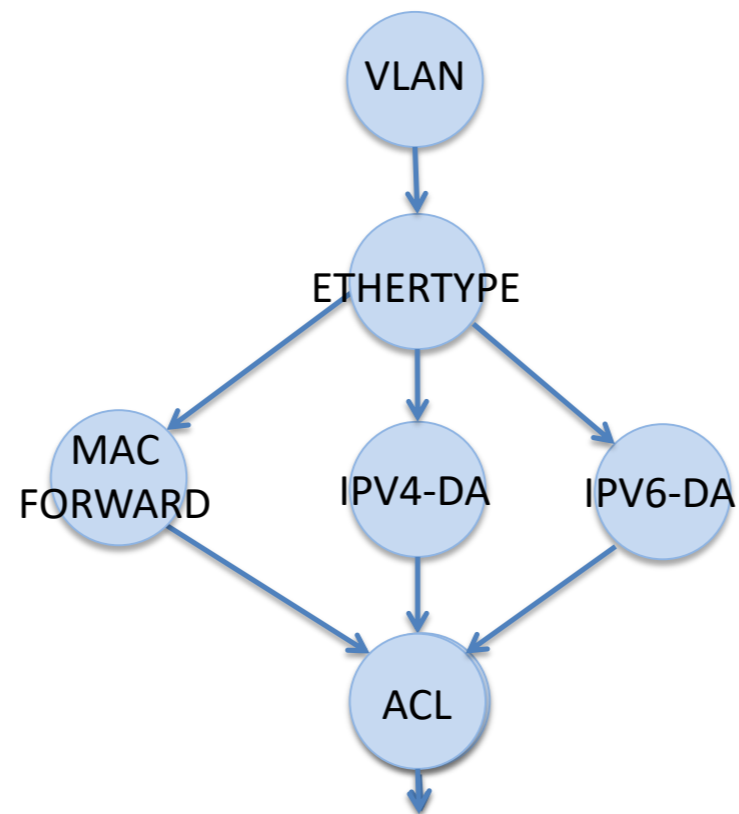
The parse graph contains nodes which corresponds to a header field and identifies the next field that follows

## Arbitrary Fields: The Parse Graph



The table graph contains nodes,  
each of which represents a match table

## Reconfigurable Match Tables: The 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?

anics48-gibb.pdf (page 1 of 12)

## Design Principles for Packet Parsers

Glen Gibb<sup>†</sup>, George Varghese<sup>‡</sup>, Mark Horowitz<sup>†</sup>, Nick McKeown<sup>†</sup>  
<sup>†</sup>Stanford University <sup>‡</sup>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  $64 \times 10$  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

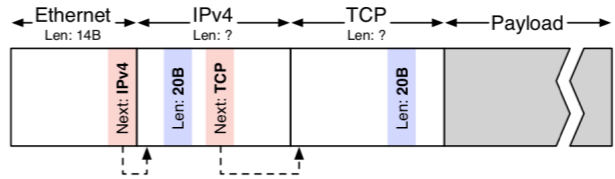
C.2.1 [Computer-Communication Networks]: Network Architecture and Design—*Network Communications*

### Keywords

Parsing; Design principles; Reconfigurable parsers

## 1. INTRODUCTION

Despite their variety, *every* network device examines fields



The diagram illustrates the structure of a TCP packet. It is divided into four main sections: Ethernet (14B), IPv4 (20B), TCP (20B), and Payload. The Ethernet section is the largest, followed by IPv4, TCP, and then the Payload. The IPv4 and TCP sections are further divided into smaller fields, with labels indicating their lengths and the next header type. For example, the IPv4 section is labeled 'Next: IPv4' and 'Len: 20B'. The TCP section is labeled 'Next: TCP' and 'Len: 20B'. The Payload section is the largest and is shown as a jagged shape, indicating it can vary in size. Above the diagram, arrows indicate the direction of the packet flow: Ethernet (Len: 14B), IPv4 (Len: ?), TCP (Len: ?), and Payload.

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<sup>1</sup> in a college campus, or MPLS<sup>2</sup> 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  $64 \times 40$  Gb/s ports must process a new packet every 270 ns.

[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

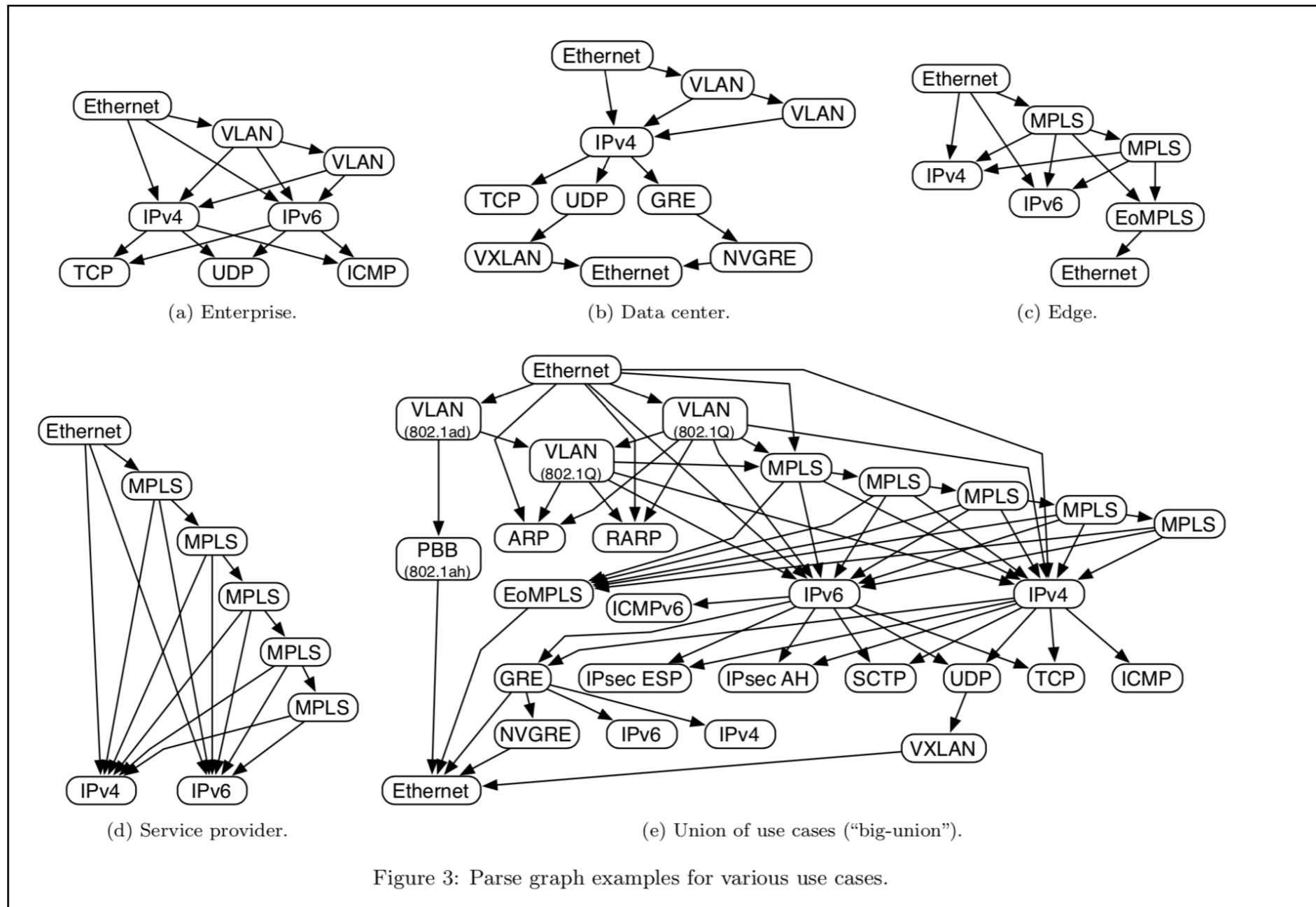
Incompleteness

Some headers do not even identify the subsequent header

Heterogeneity

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

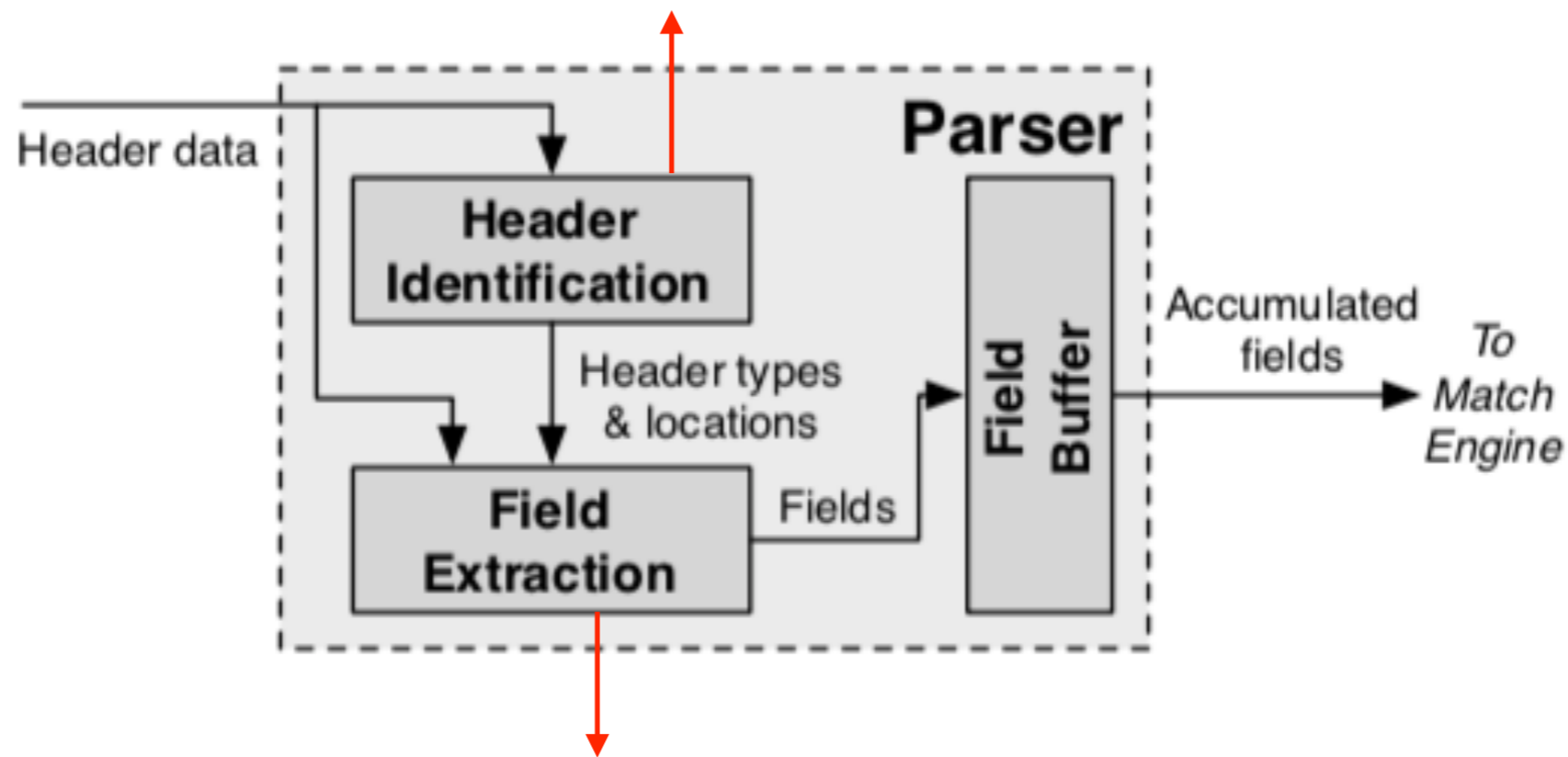
# Parse graphs are directed acyclic graphs encoding header types and their sequence



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

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

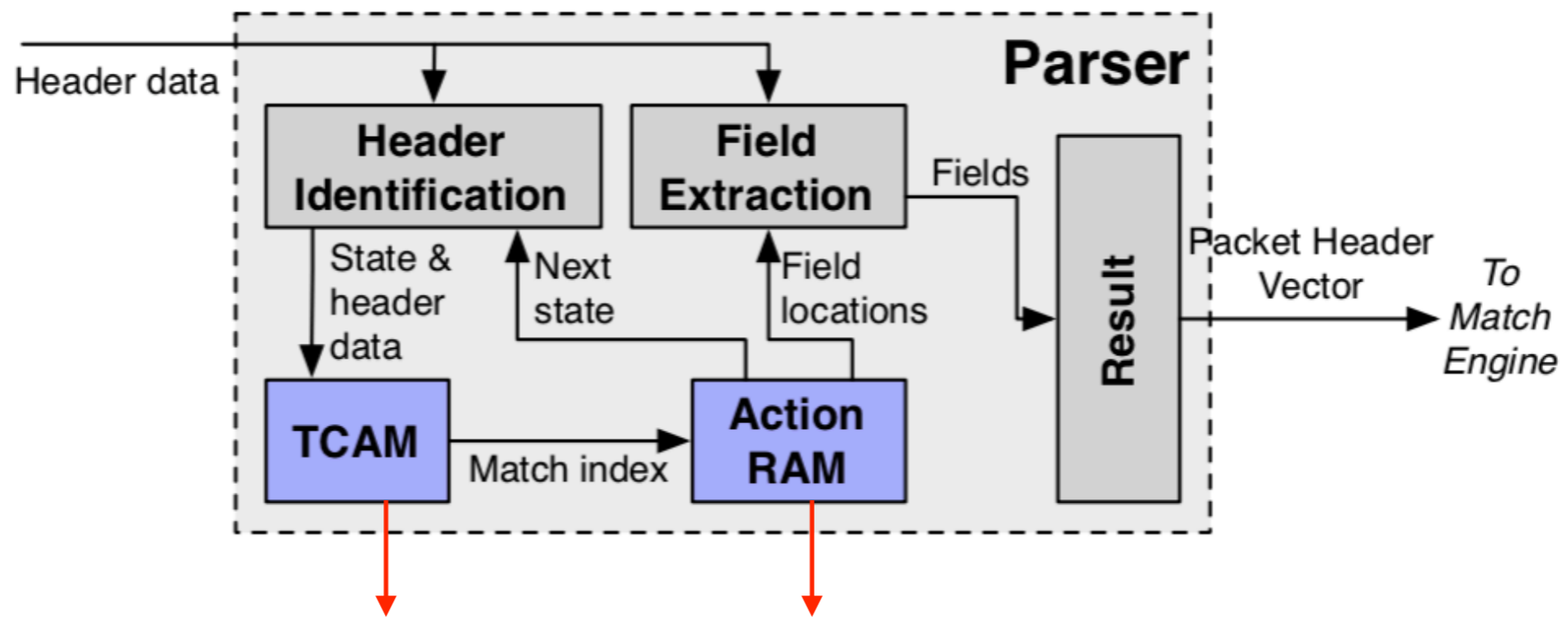
implements the parse graph's  
state machine



extracts the chosen fields  
from identified headers

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

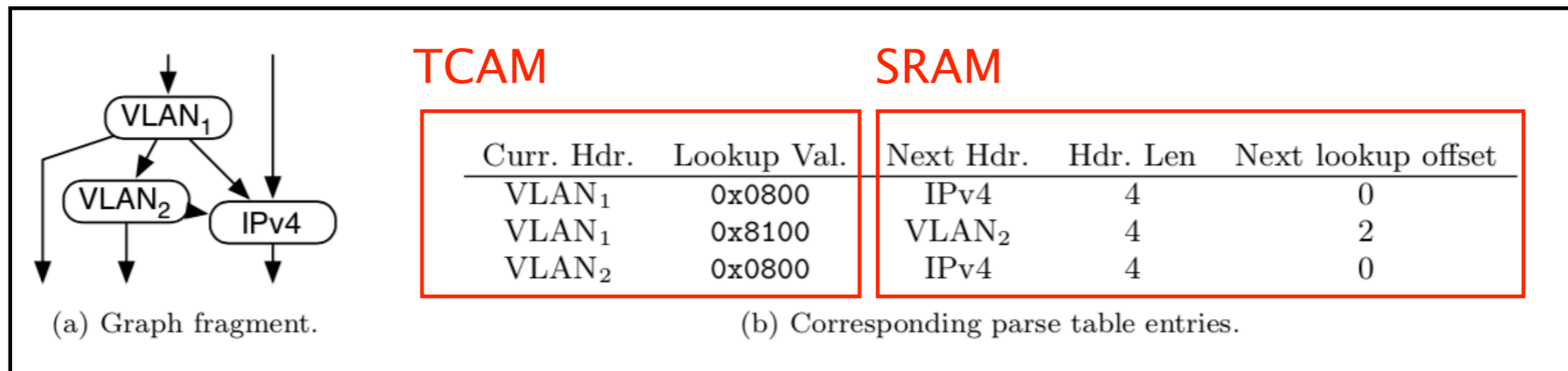
stored in memory,  
e.g. in RAM and/or TCAM



stores the bit sequences  
that identify the headers

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

Linked together, a SRAM and TCAM can encode the transition table attached to a parsing graph

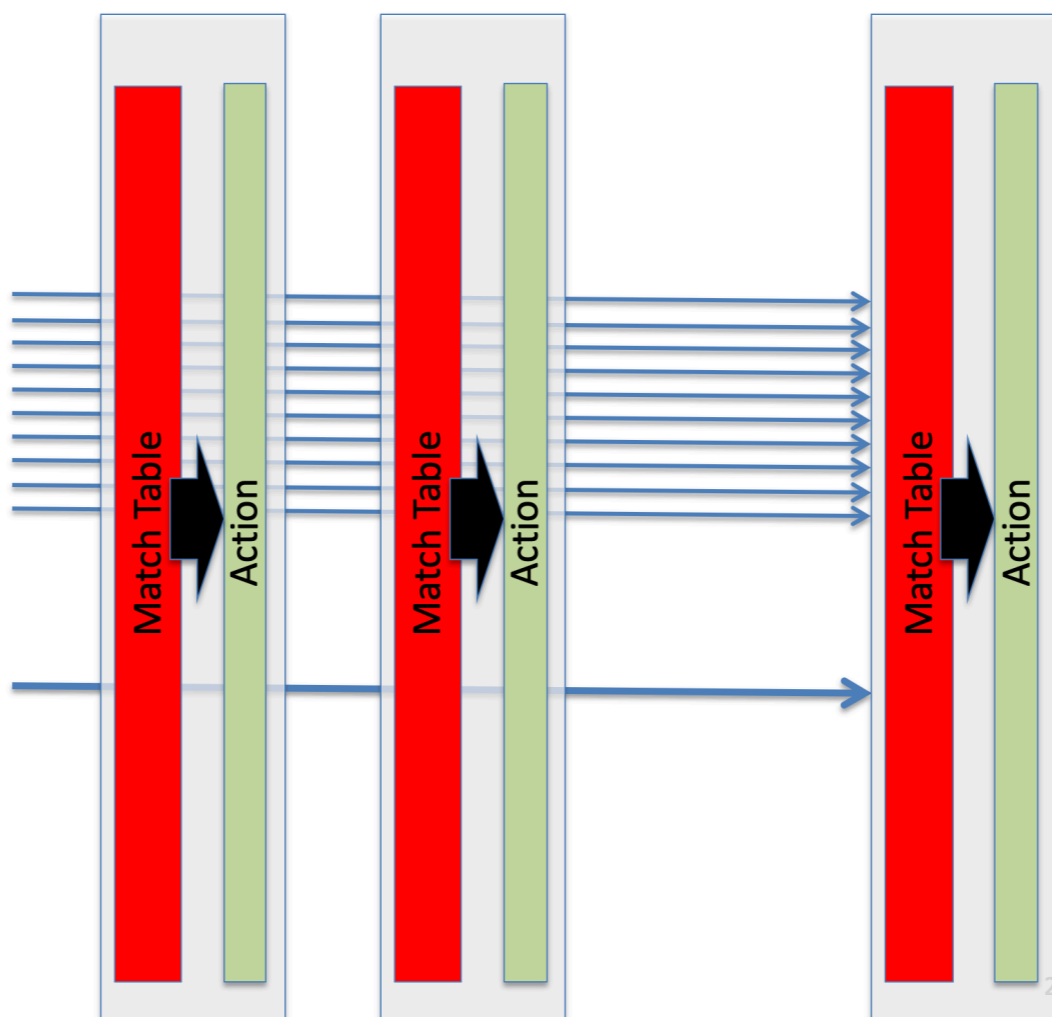
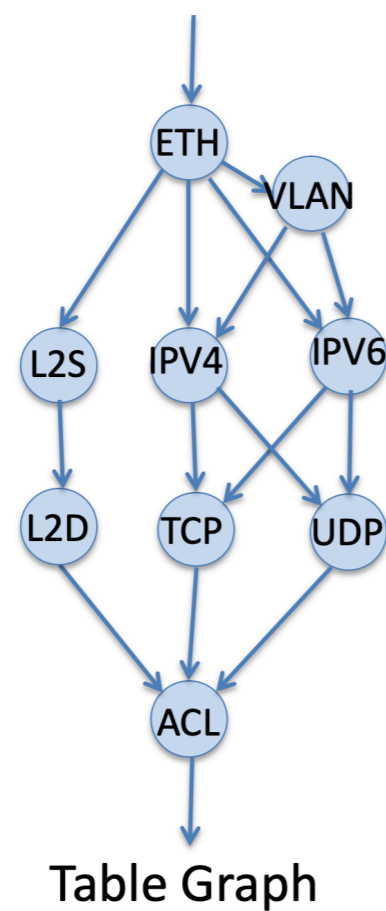


Source: Design Principles for Packet Parsers, Gibb et al.

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

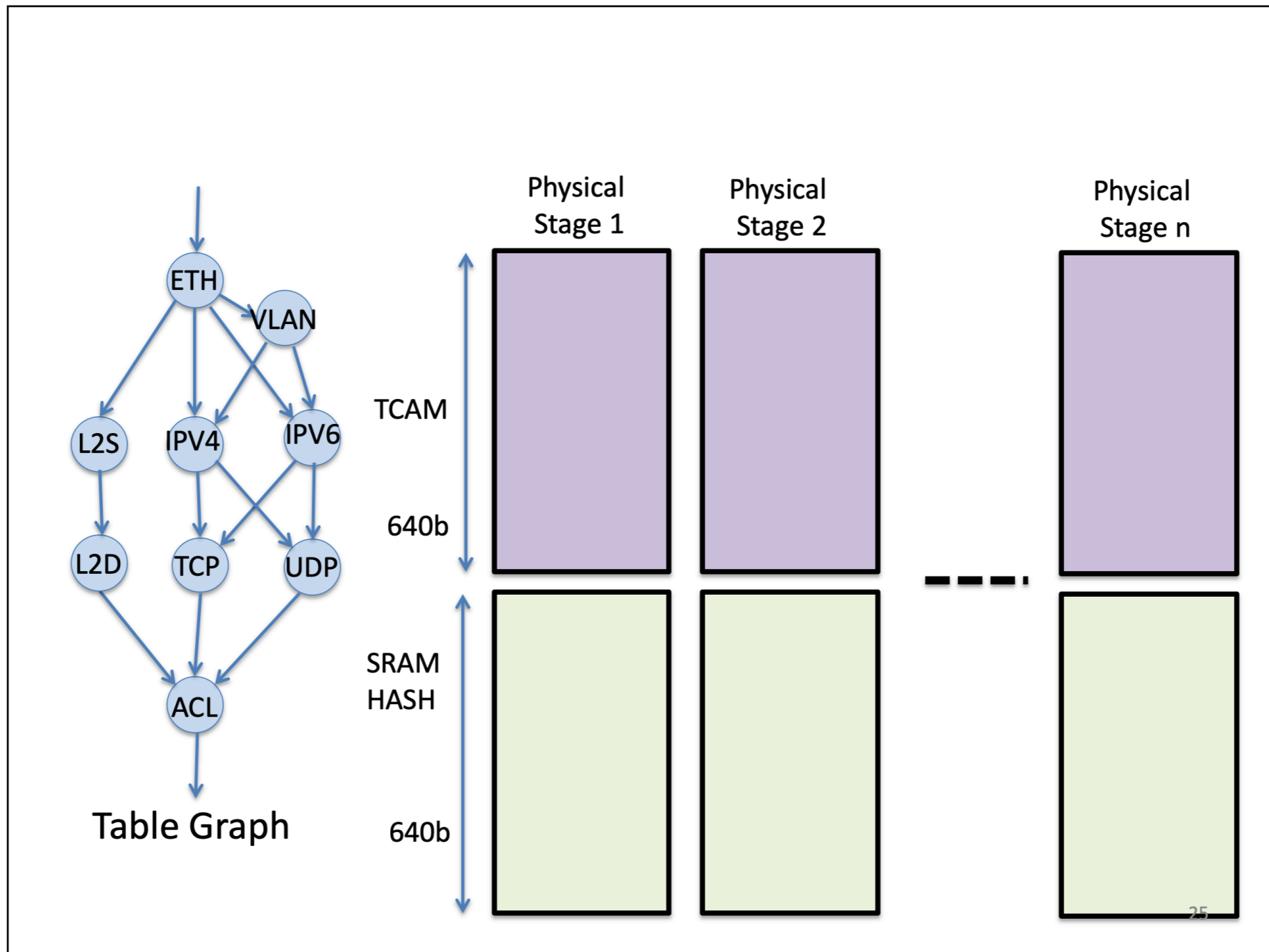
A compiler translates a given RMT logical pipeline (specified in P4) into a physical one

## RMT Logical to Physical Table Mapping

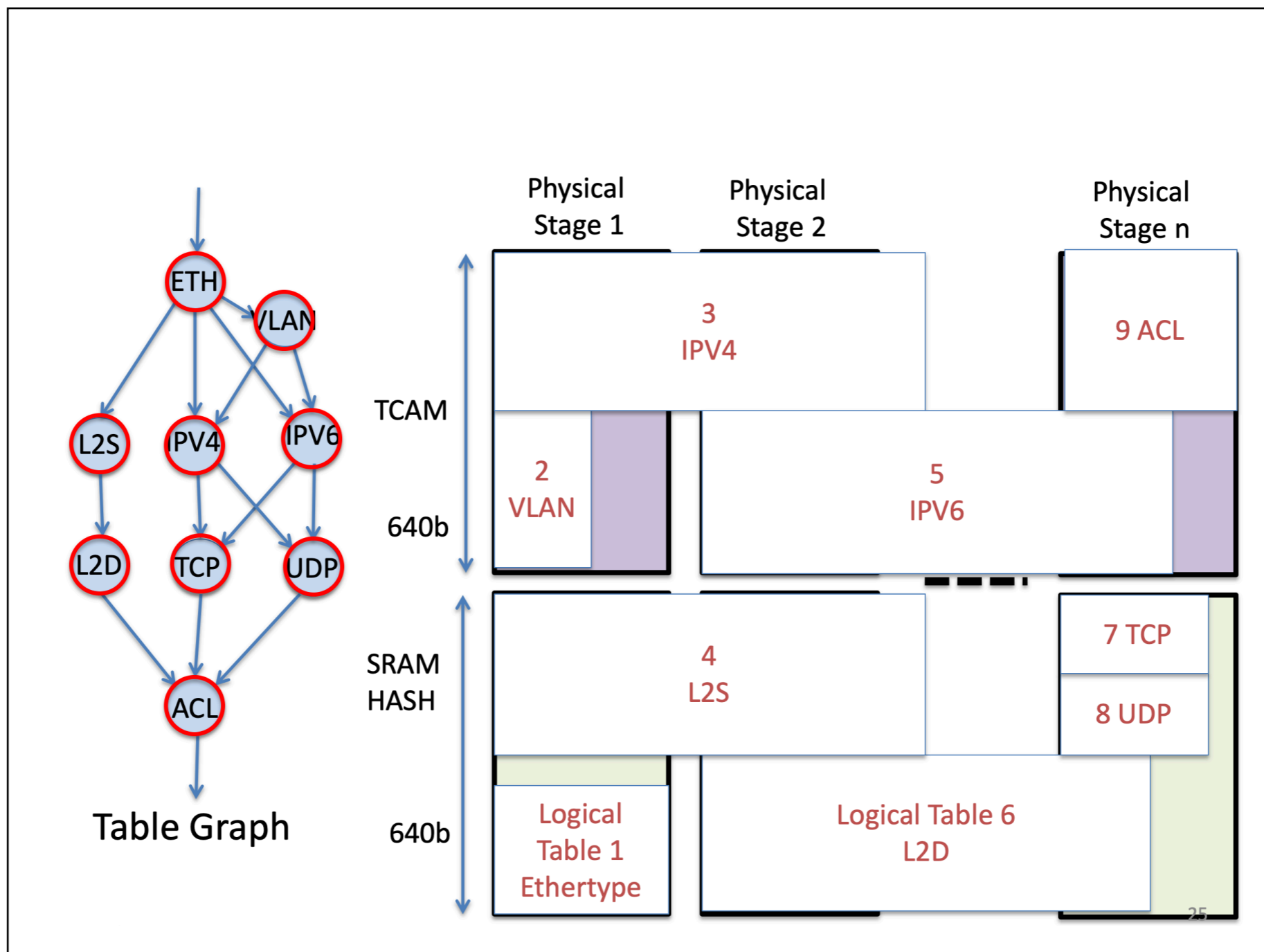




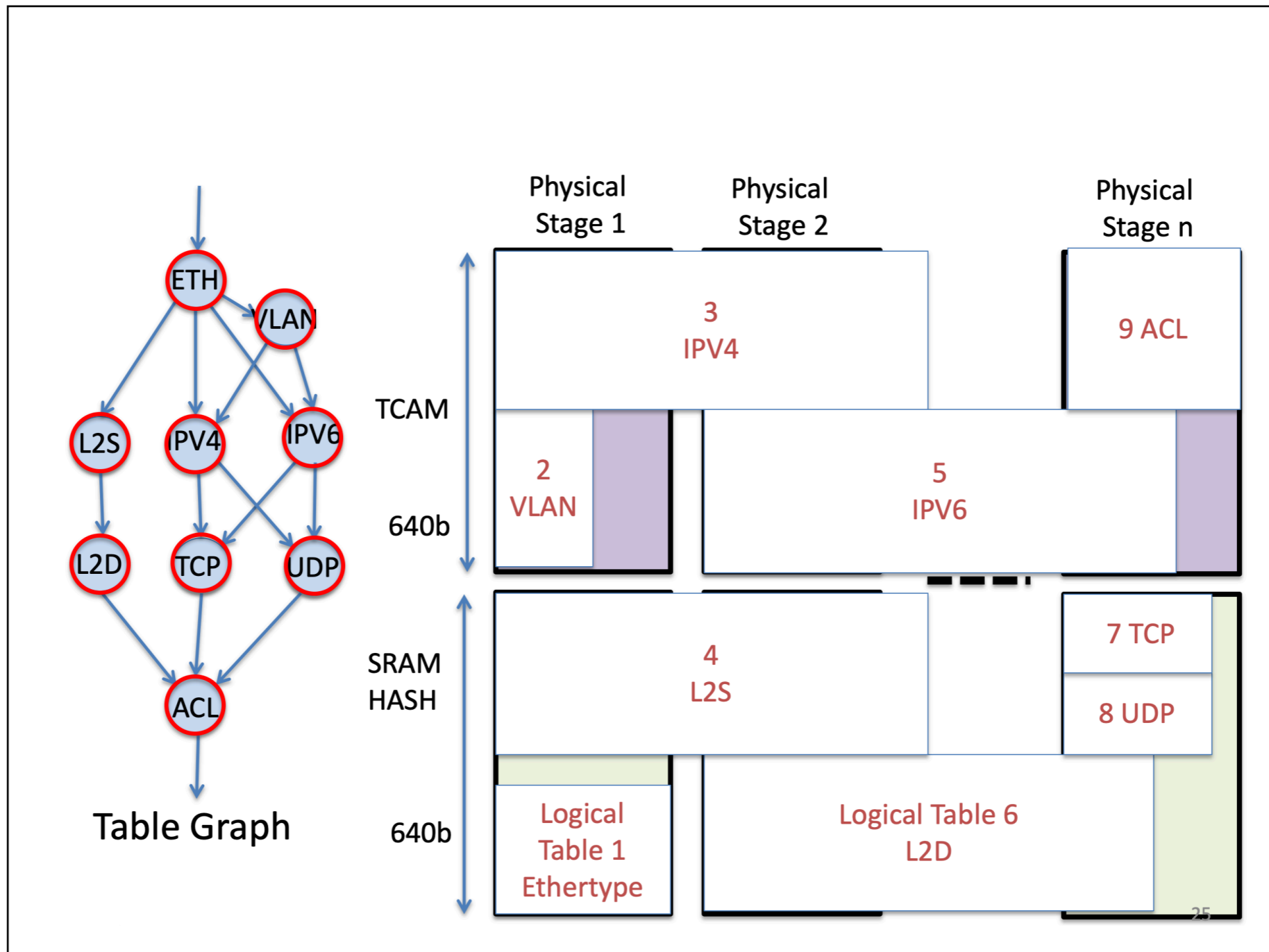
Each physical stage contains dedicated SRAM, for exact matches, and TCAM, for ternary matches



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

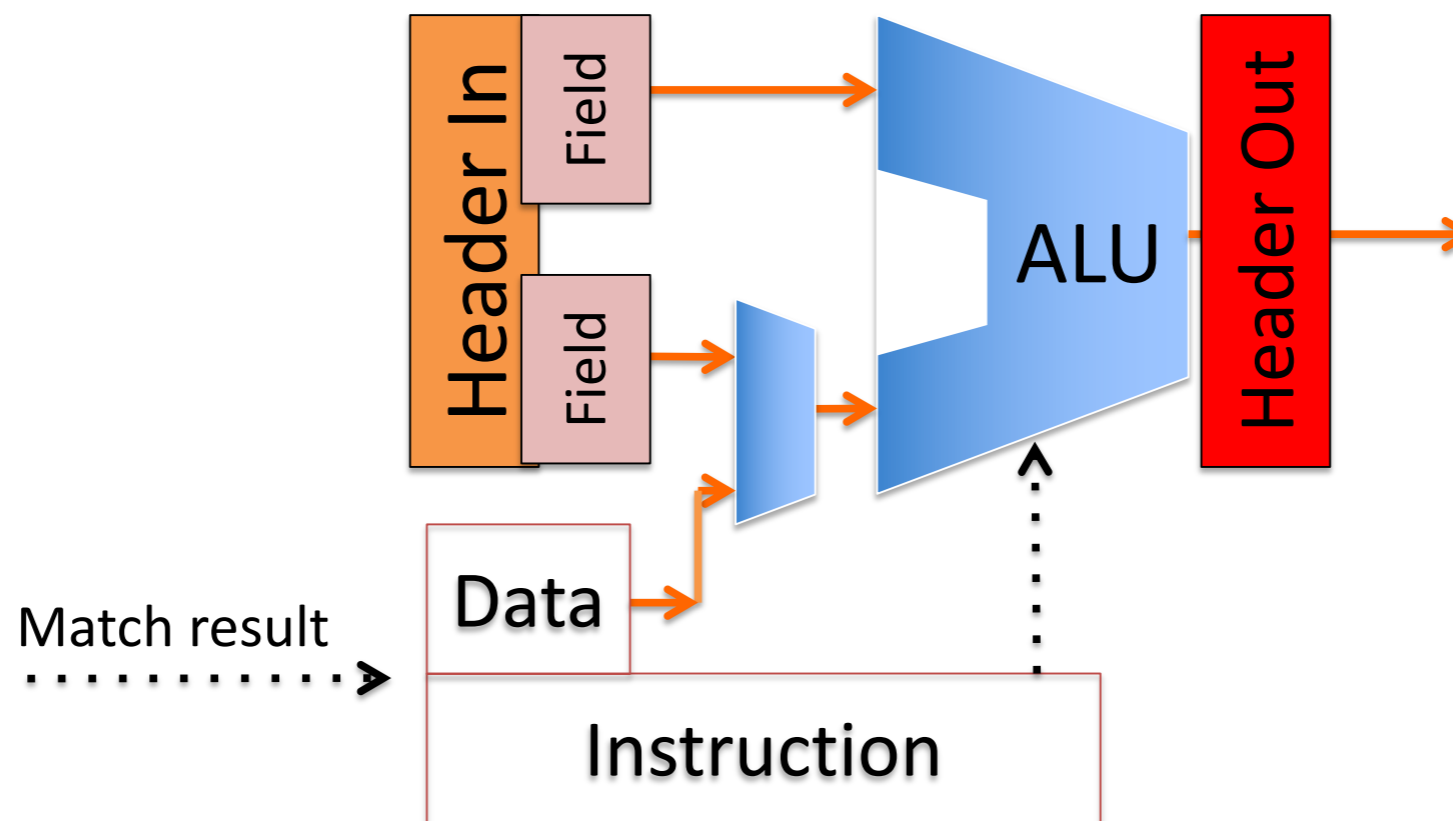


Small tables can share a stage (up to 16 per stage), while large tables can span multiple ones



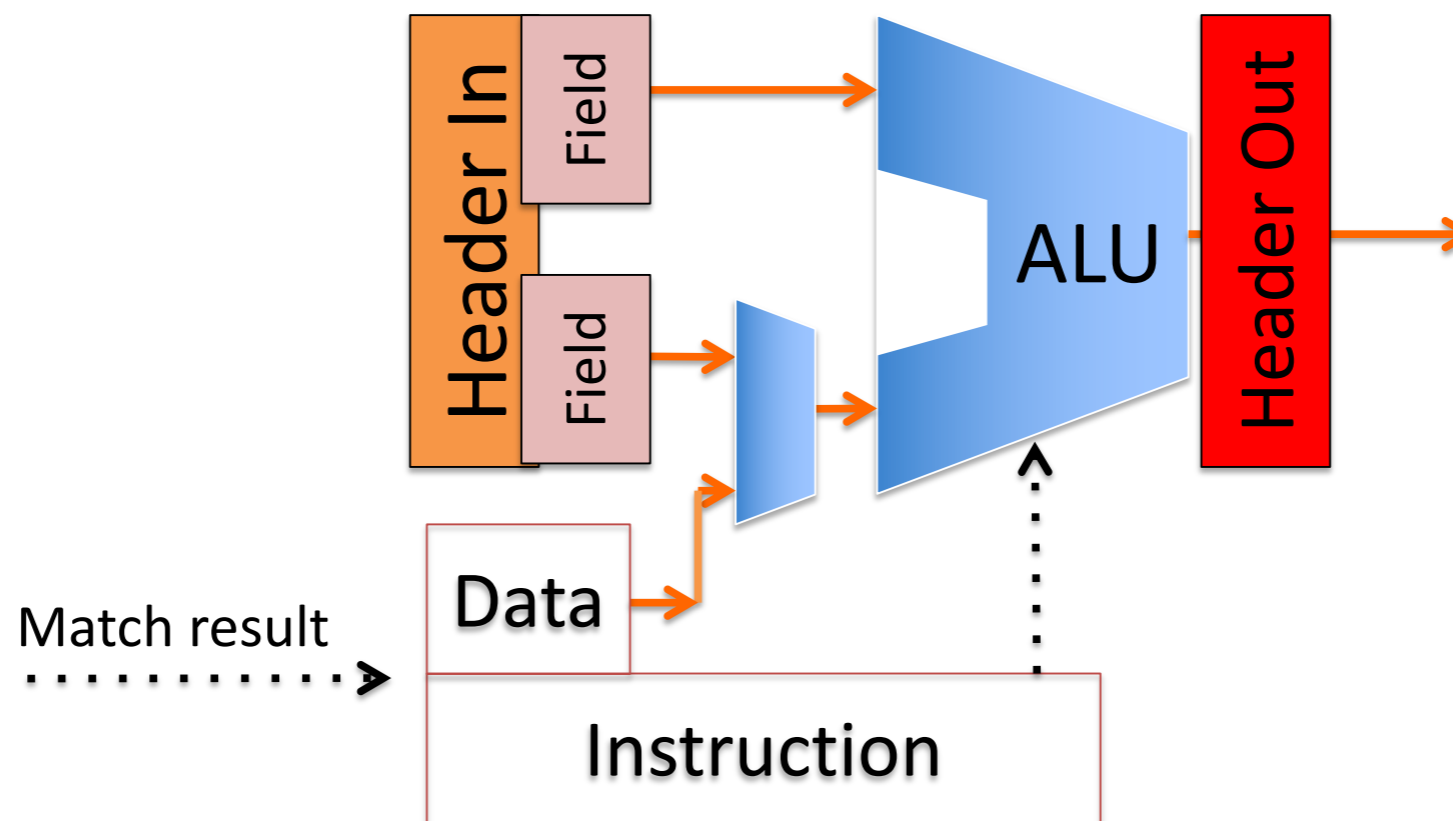
The RMT pipeline relies on many Arithmetic Logic Units (ALU) to perform actions on the result of a match

## Action Processing Model



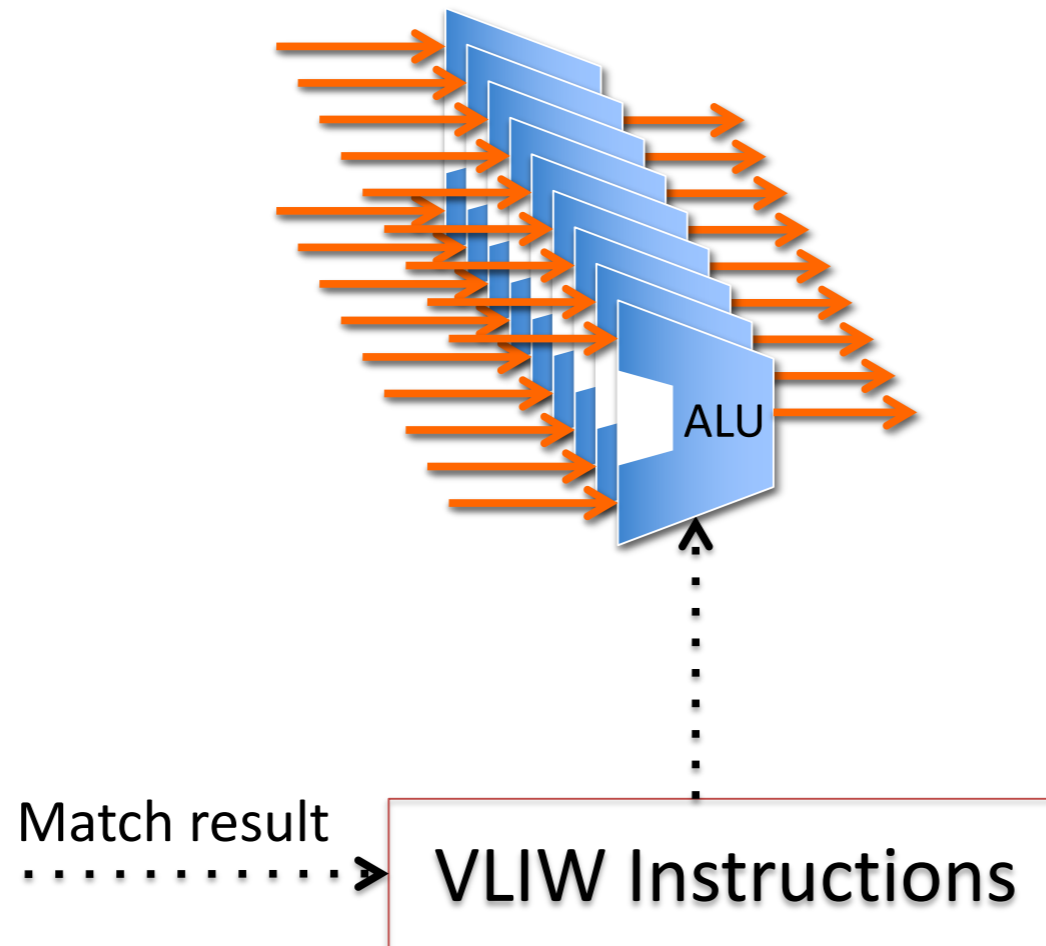
Each ALU modifies only one word of a header  
(a header is composed of *many* words)

## Action Processing Model



Each stage of the RMT pipeline contains  
one ALU per word of the header vector (that's *a lot* of ALUs)

Modeled as Multiple VLIW CPUs per Stage



# 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

In terms of die area, flexibility is not very expensive  
at least, not anymore... mainly thanks to Moore's law

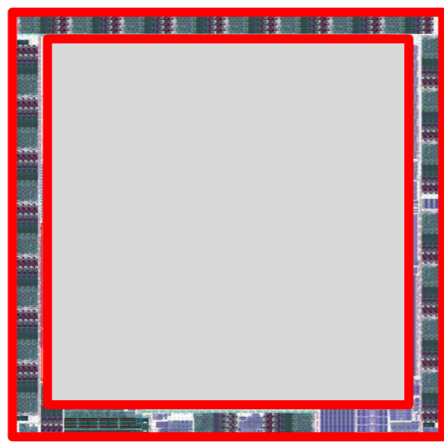
## Chip Comparison with Fixed Function Switches

Area

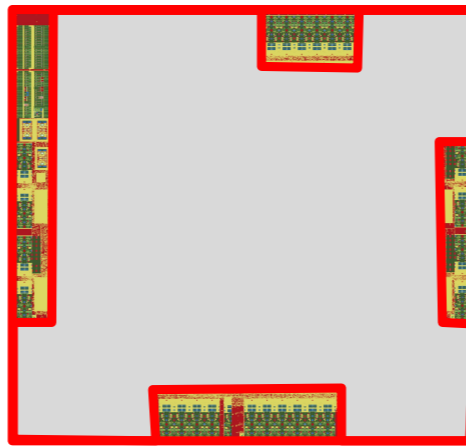
Section	Area % of chip	Extra Cost
→ IO, buffer, queue, CPU, etc	37%	0.0%
→ Match memory & logic	54.3%	8.0%
→ VLIW action engine	7.4%	5.5%
Parser + deparser	1.3%	0.7%
<b>Total extra area cost</b>		<b>14.2%</b>

Serializer/Deserializer (SerDes) usually account for 30% of the area

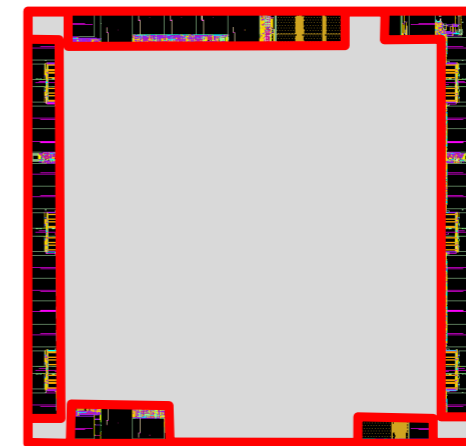
## Serial I/O: About 30% of switch chip area



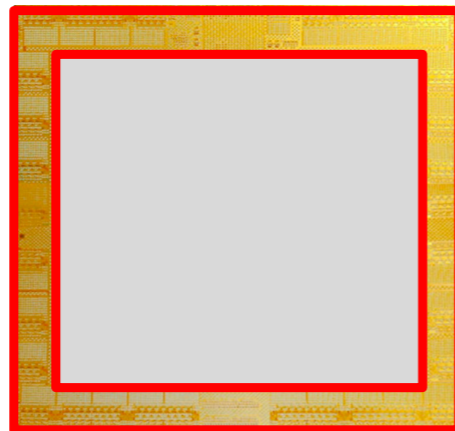
Intel Alta (2011)



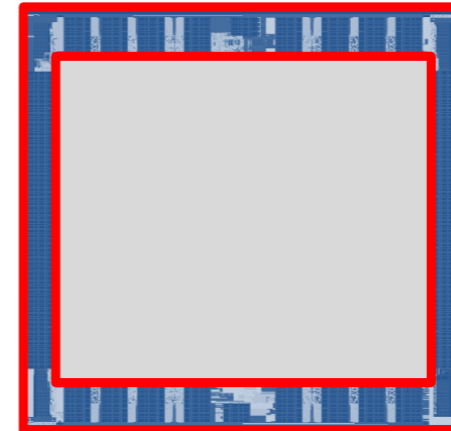
Cisco (2011)



Ericsson Spider (2011)

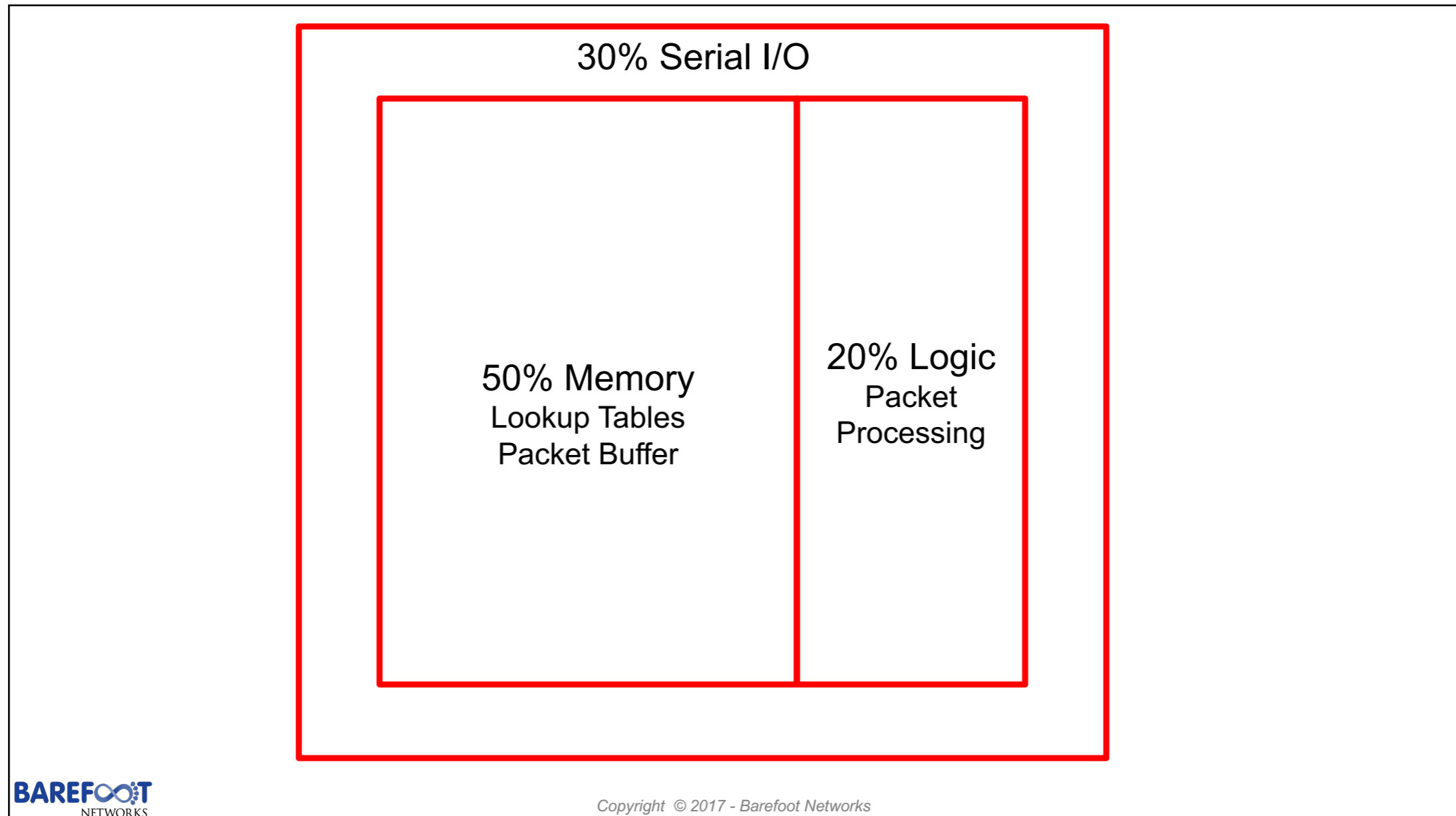


Broadcom Tomahawk (2014)



Barefoot Tofino (2016)

Memory usually account for ~50% of the die area,  
leaving us around 20% for the processing logic



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

As SerDes and memory technologies progress,  
the relative area dedicated to logic shrinks

Observations

With every new generation of network devices,  
people expect larger speeds and more memory

Consequences

relative areas of SerDes/memory stay roughly equivalent  
**logic shrinks**

Even with an increased space for logic,  
the device tends to be relatively the same

## Chip Comparison with Fixed Function Switches

Area

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

# The same lesson applies for power

## Chip Comparison with Fixed Function Switches

Area

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

Power

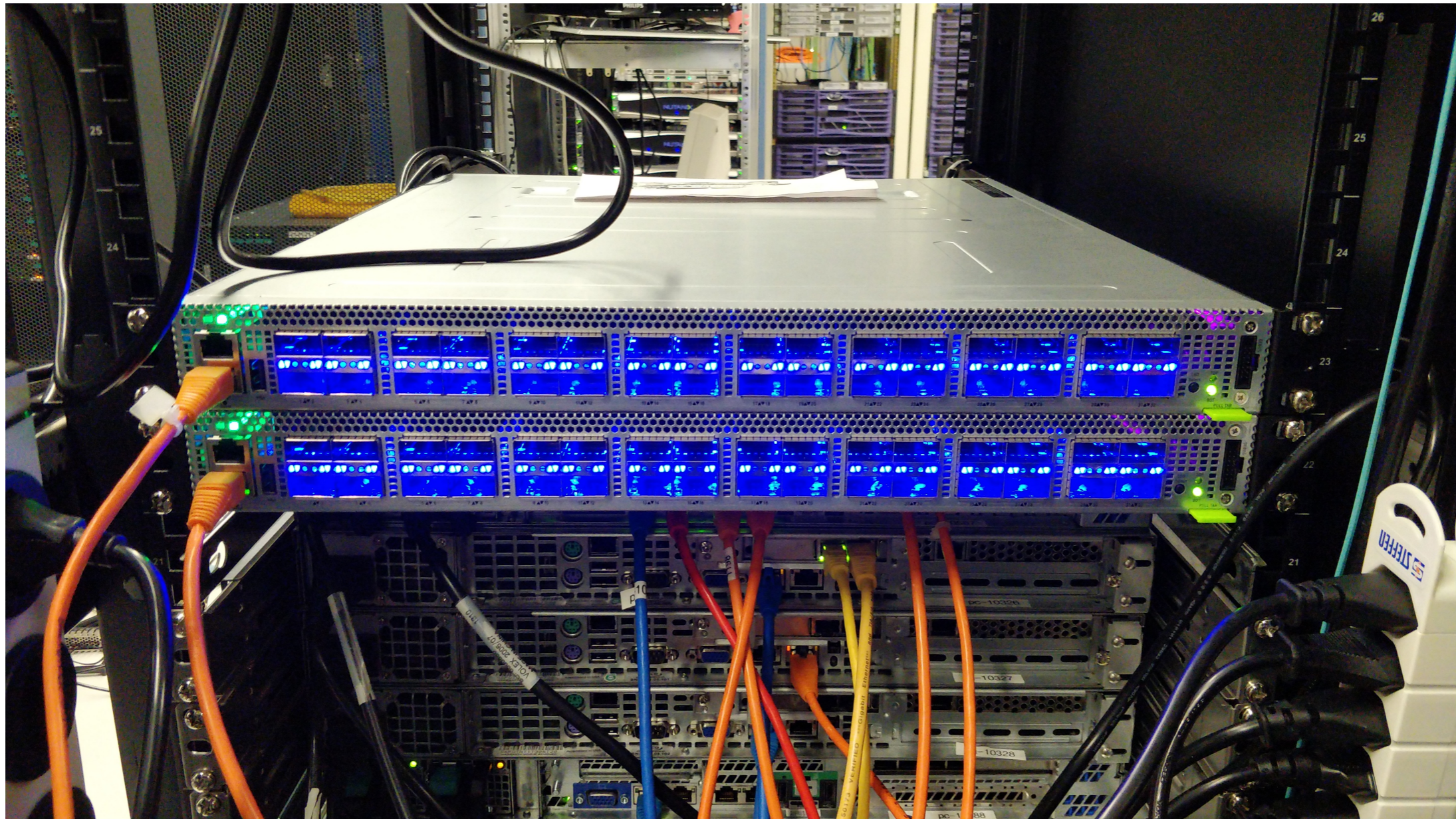
Section	Power % of chip	Extra Cost
→ I/O	26.0%	0.0%
→ Memory leakage	43.7%	4.0%
Logic leakage	7.3%	2.5%
RAM active	2.7%	0.4%
TCAM active	3.5%	0.0%
→ Logic active	16.8%	5.5%
<b>Total extra power cost</b>		<b>12.4%</b>

# Conclusion

- How do we design a flexible chip?
  - The RMT switch model
  - Bring processing close to the memories:
    - pipeline of many stages
  - Bring the processing to the wires:
    - 224 action CPUs per stage
- How much does it cost?
  - 15%
- Lots of the details how we designed this in 28nm CMOS are in the paper



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



Programmable Data Plane at Terabit Speeds

Vladimir Gurevich  
May 16, 2017



Copyright © 2017 - Barefoot Networks

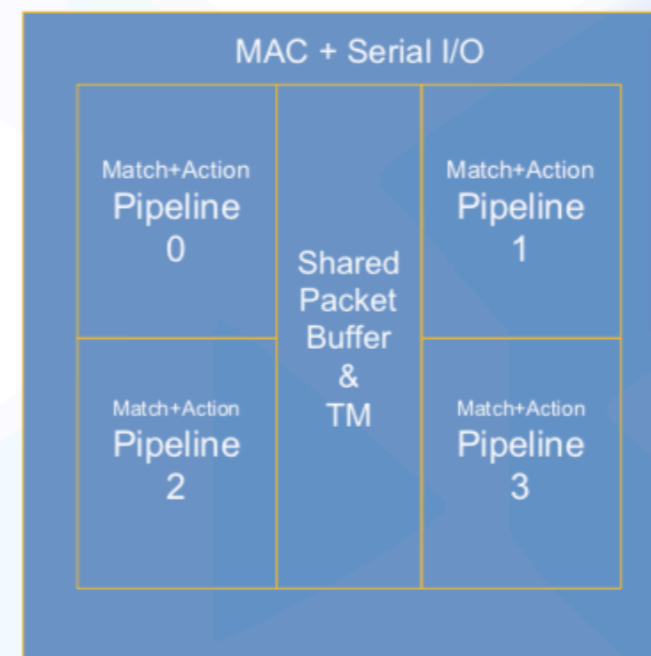
Source: Programmable Data Planes at Terabit Speeds, Vladimir Gurevich, 2017

# Barefoot Tofino 6.5 Tbps backplane

several billion packets per second at line rate

## 6.5Tb/s Tofino™ Summary

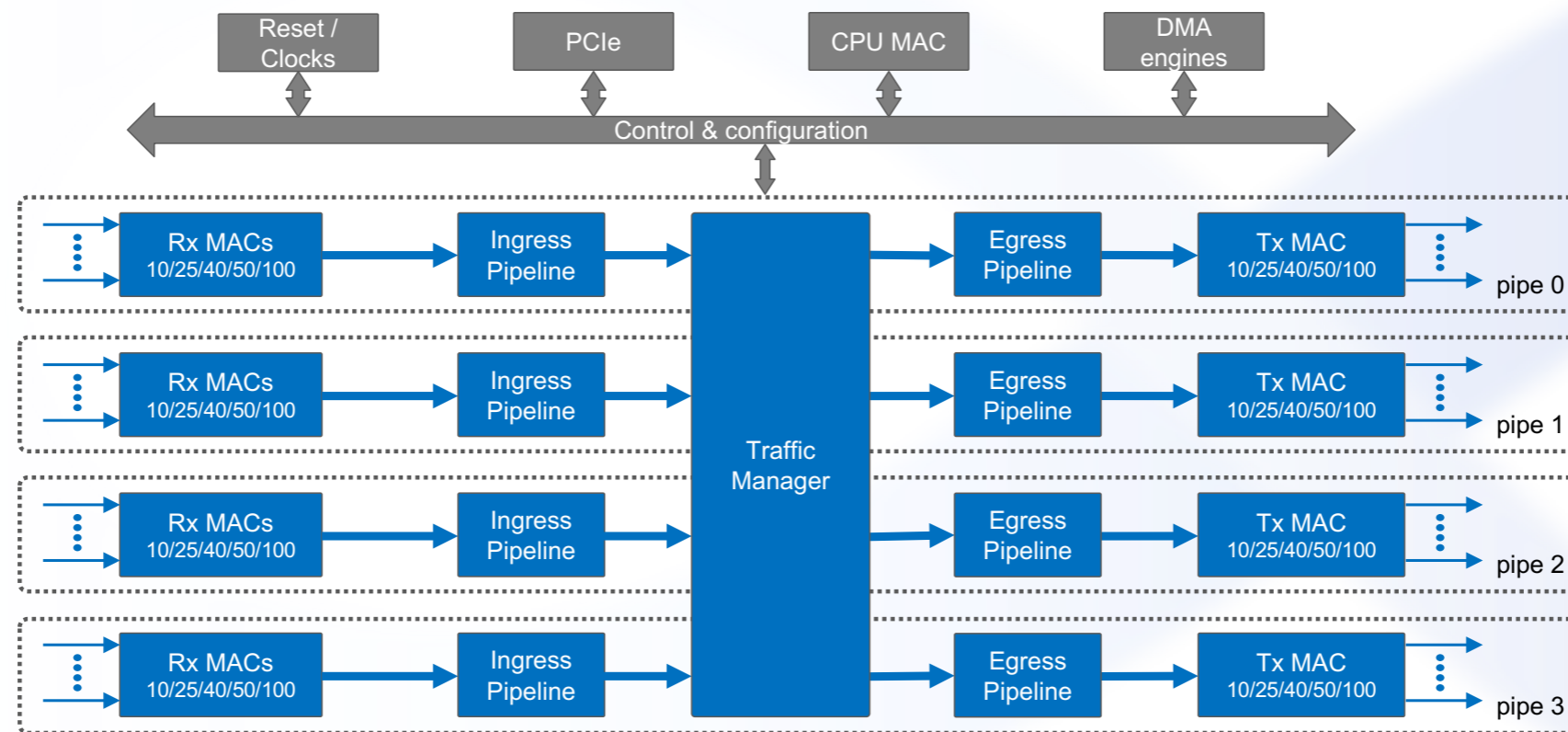
- **State of the art design**
  - Single Shared Packet Buffer
  - TSMC 16nm FinFET+
- **Four Match+Action Pipelines**
  - Fully programmable PISA Embodiment
  - All compiled programs run at line-rate.
  - Up to 1.3 million IPv4 routes
- **Port Configurations**
  - 65 x 100GE/40GE
  - 130 x 50GE
  - 260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



# 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**

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

## Packet Header Vector (PHV)

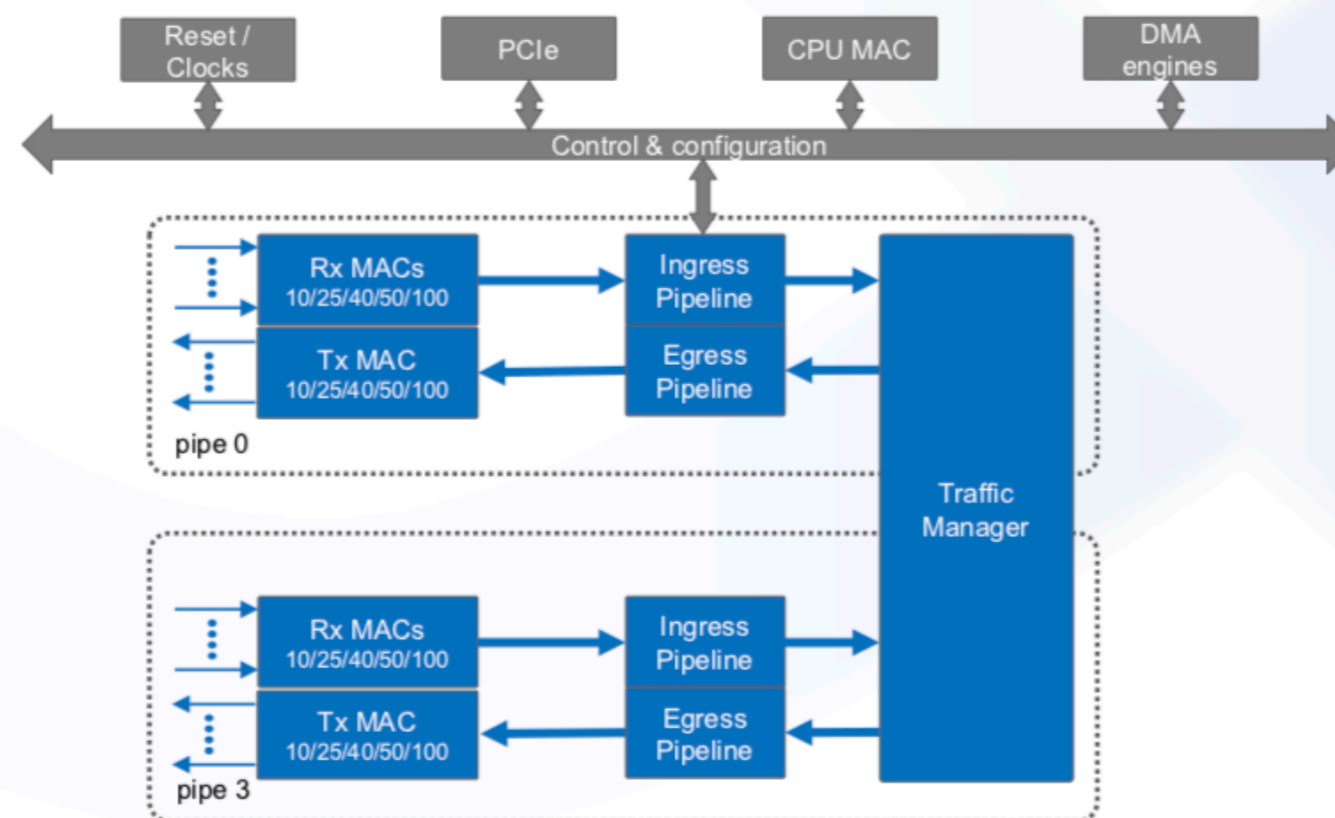
- A set of uniform containers that carry the headers and metadata along the pipeline
- Fields can be packed into any container or their combination
- PHV Allocation step in the compiler decides the actual packing



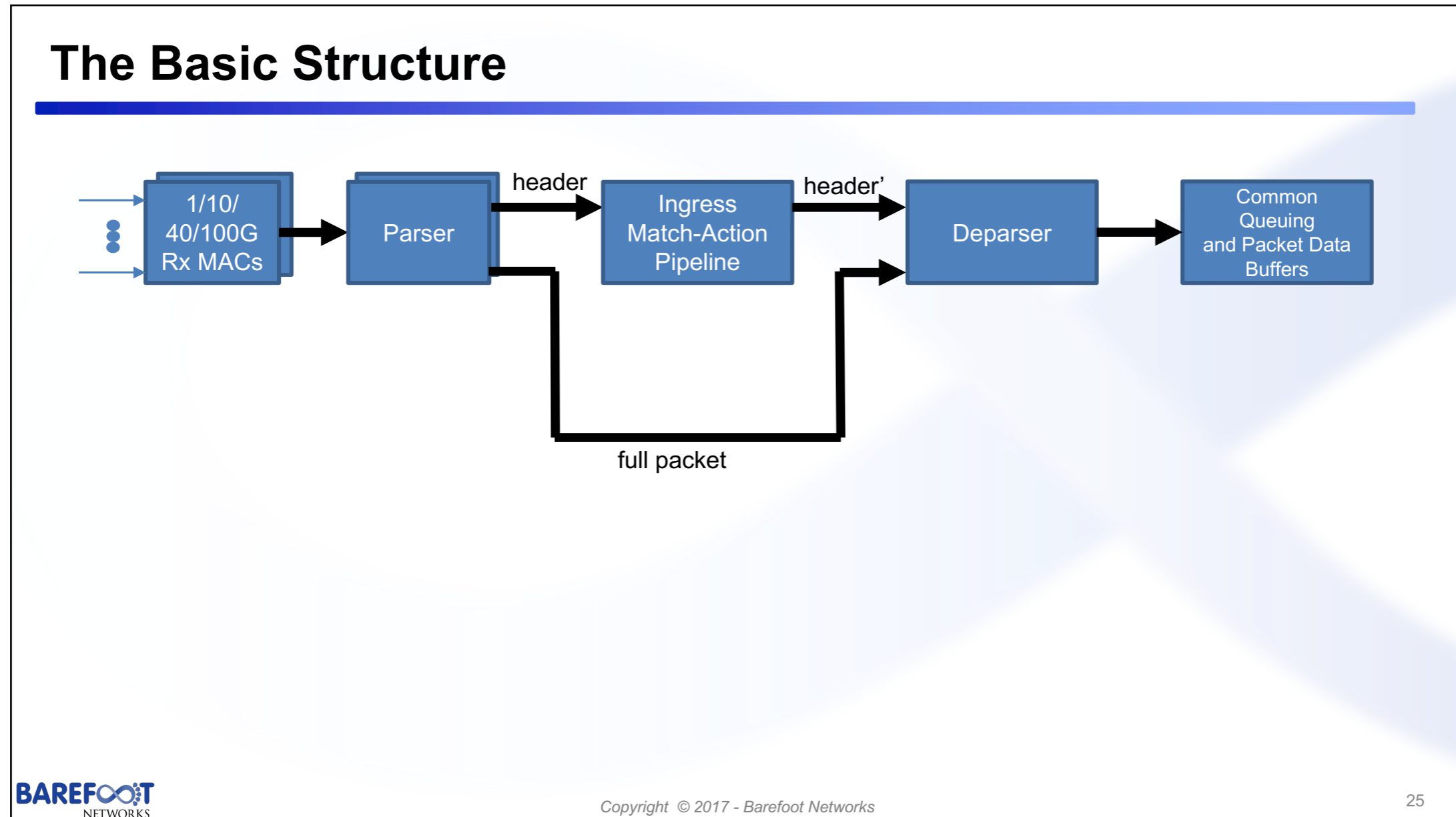
Tofino uses a folded pipeline in which the *same* stages are used for both the ingress and the egress pipeline

## Unified Pipeline

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

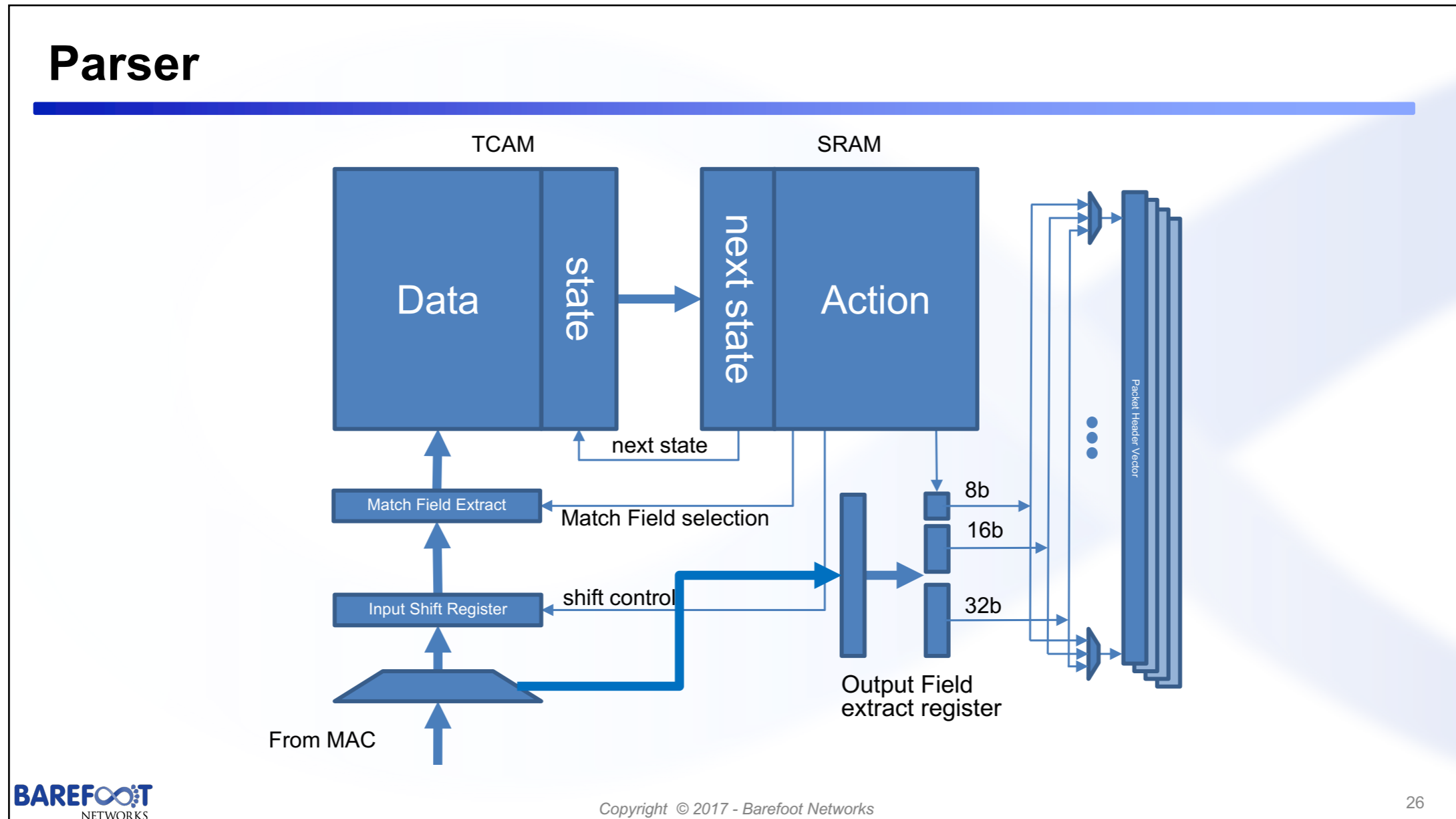


In terms of structure,  
Tofino basically follows the RMT pipeline



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

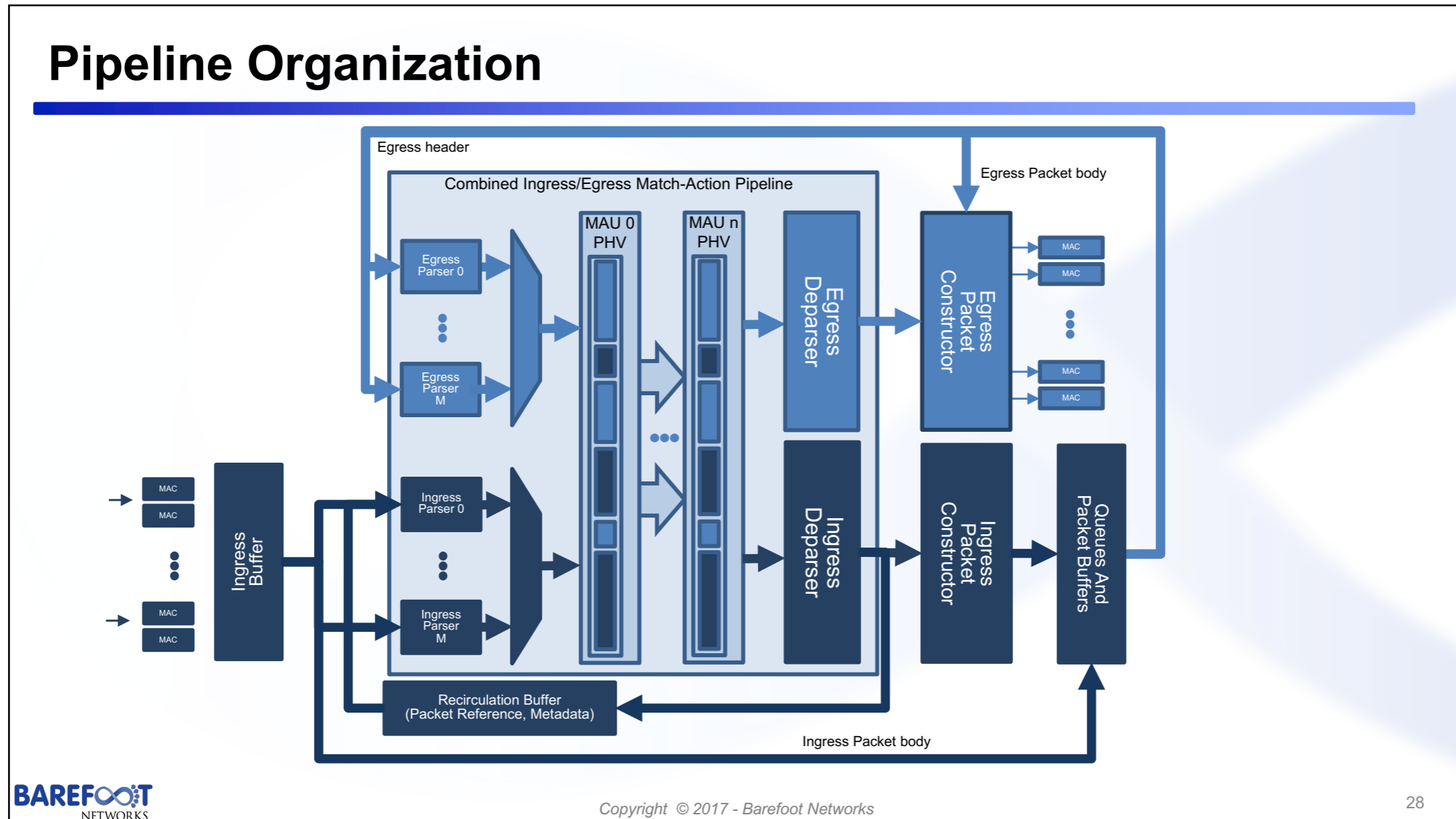
# The same goes for the design of the programmable parser



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



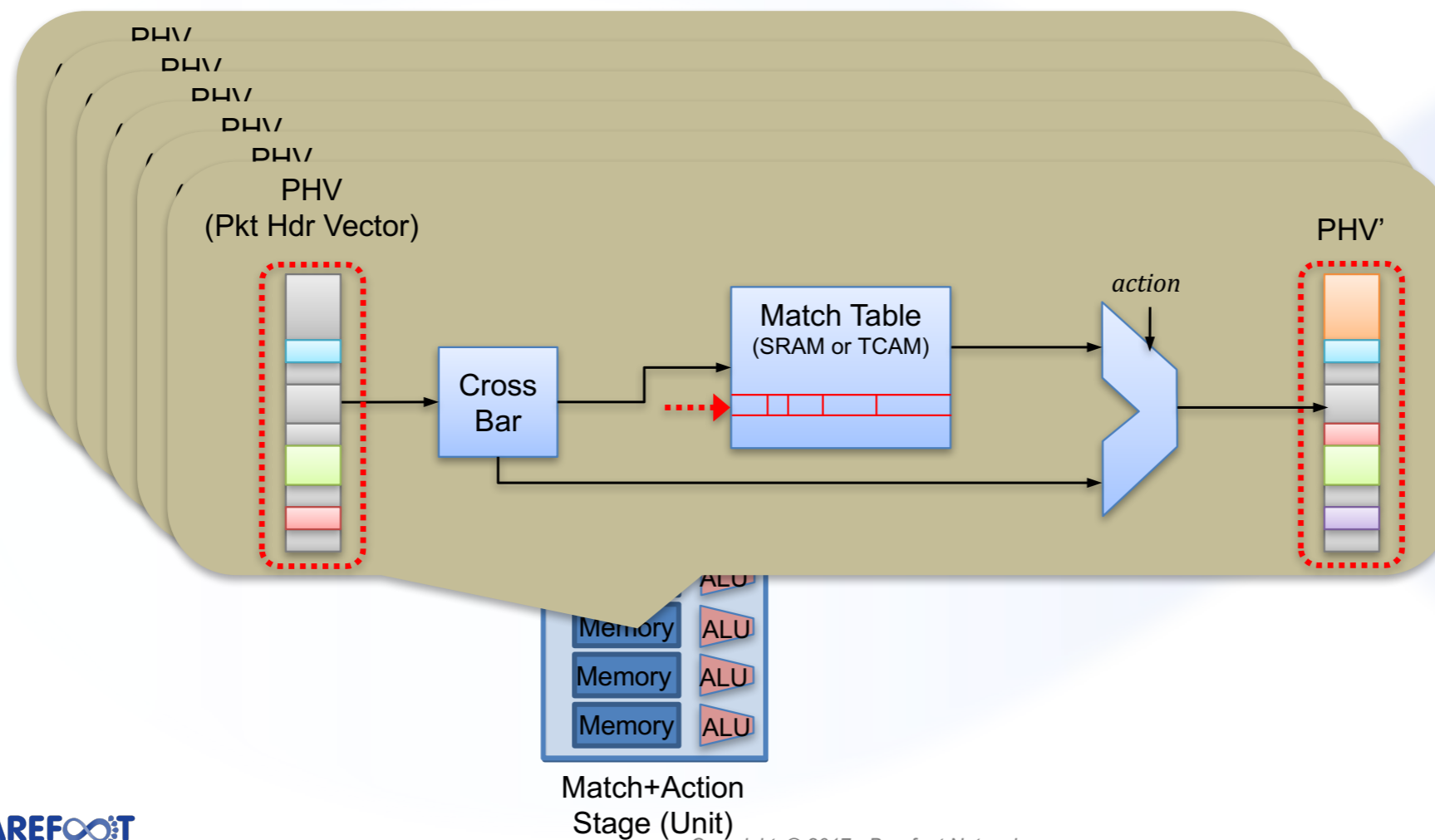
# Putting everything together



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

Each match action stage is regularly structured around:  
crossbars, memory units, and ALUs

## What Happens Inside?



# Parallelism in P4

```
apply {
  /* Parallel lookups possible */
  subnet_vlan.apply();
  mac_vlan.apply();
  protocol_vlan.apply();
  port_vlan.apply();

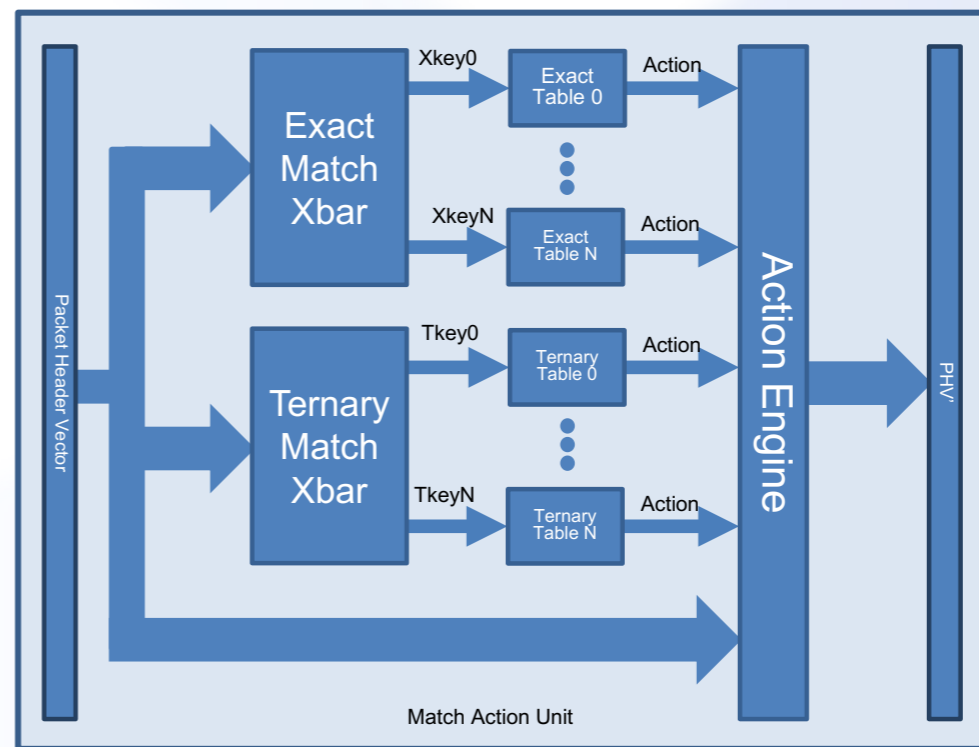
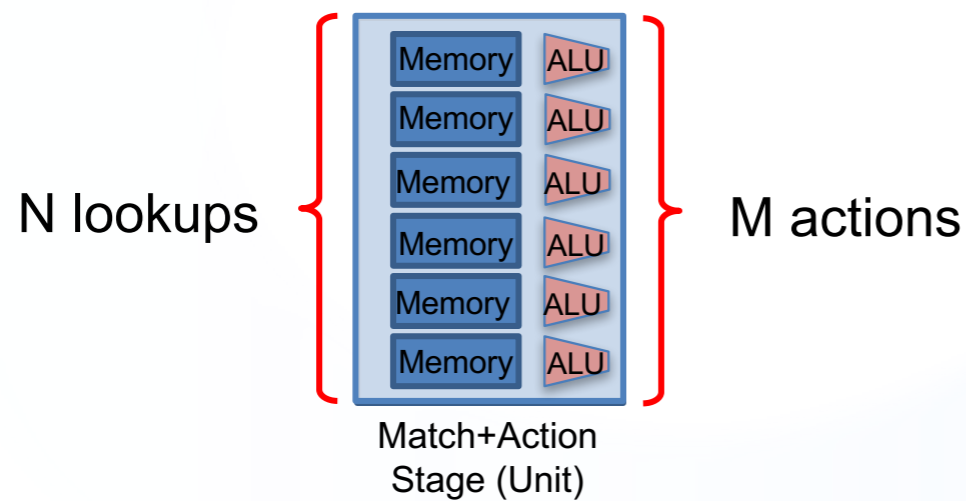
  /* Resolution in next stage */
  resolve_vlan.apply();
}

apply {
  if (!subnet_vlan.apply().hit) {
    if (!mac_vlan.apply().hit) {
      if (!protocol_vlan.apply().hit) {
        port_vlan.apply();
      }
    }
  }
}
```

- **Most P4 programs have inherent parallelism**
- **Others can be executed speculatively**
- **Switch.p4**
  - ~100 tables and if() statements
  - ~22 stages divided between ingress and egress
  - Degree of parallelism ~4.5

# How Tofino Supports Parallel Processing

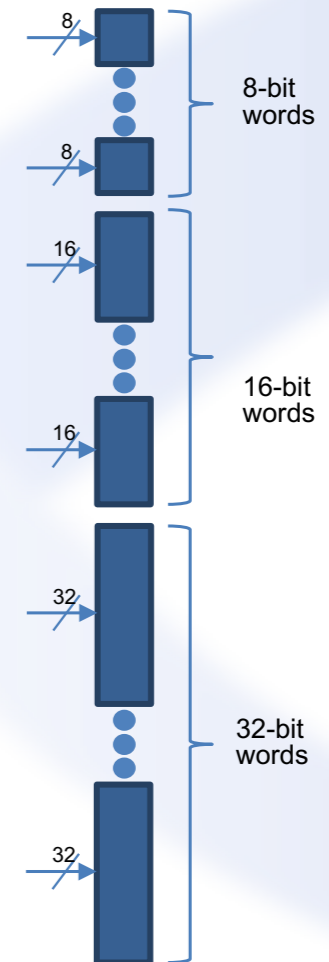
- Multiple tables mean multiple parallel lookups
- All actions from all active tables are combined



# Parallelism in P4

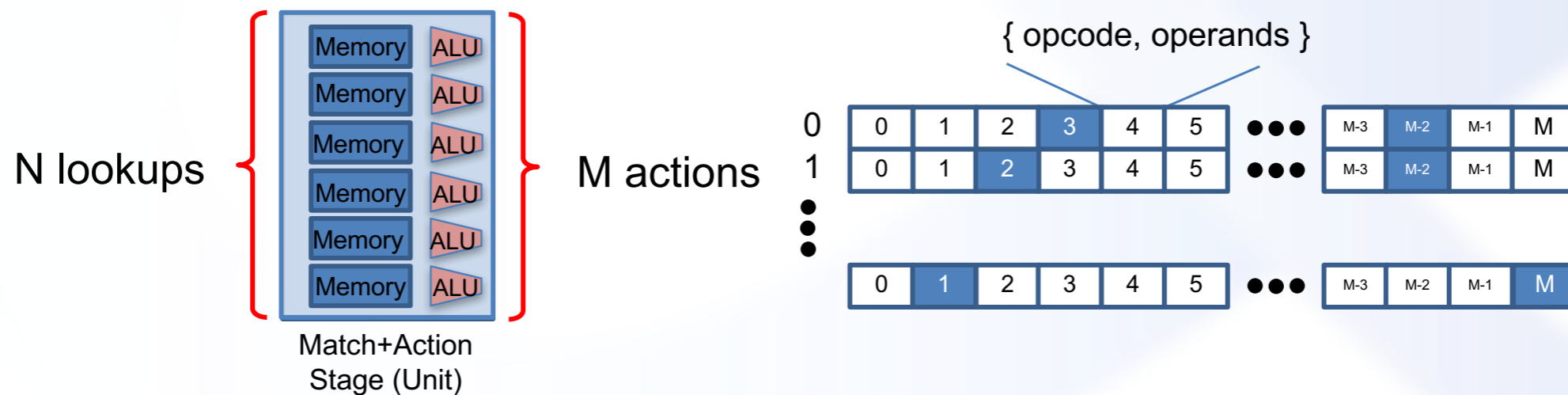
```
action ipv4_in_mpls(in bit<20> label1, in bit<20> label2) {  
  hdr.mpls[0].setValid();  
  hdr.mpls[0].label = label1;  
  hdr.mpls[0].exp = 0;  
  hdr.mpls[0].bos = 0;  
  hdr.mpls[0].ttl = 64;  
  
  hdr.mpls[1].setValid();  
  hdr.mpls[1] = { label2, 0, 1, 128 };  
  
  if (hdr.vlan_tag.isValid()) {  
    hdr.vlan_tag.etherType = 0x8847;  
  } else {  
    hdr.ethernet.etherType = 0x8847;  
  }  
}
```

- **Most actions can be easily parallelized**
  - This action can be executed in 1 cycle
    - Number of parallel operations: 12
- **Keep fields in separate containers**



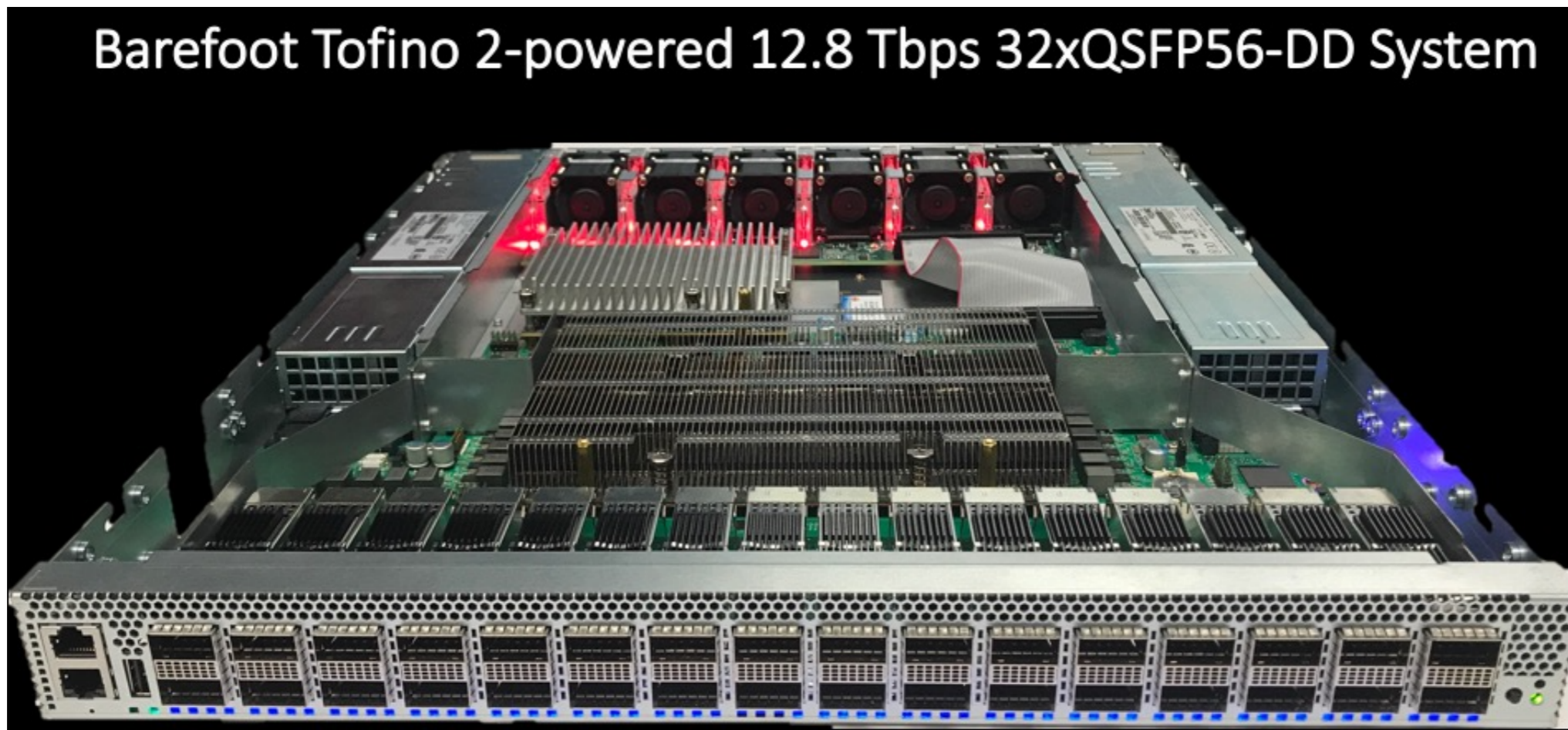
# How Tofino Supports Parallel Processing

- Multiple tables mean multiple parallel lookups
- All actions from all active tables are combined
- Each PHV container has its own, independent processor



What's next?

Tofino 2: 12.8 Tbps (7 nm switching ASIC)



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

P4 hardware  
target

P4-based  
applications

What cool things  
can we do with it?



A high-level, **non-exhaustive overview** of the research surrounding data plane programmability

A high-level, non-exhaustive overview of the research surrounding data plane programmability

Data plane  
programmability for

Performance  
Monitoring  
Applications offloading

Platforms for Data plane  
Correctness programmability  
Management

Data plane  
programmability for

**Performance**

Monitoring

Applications offloading

Platforms

for

Data plane  
programmability

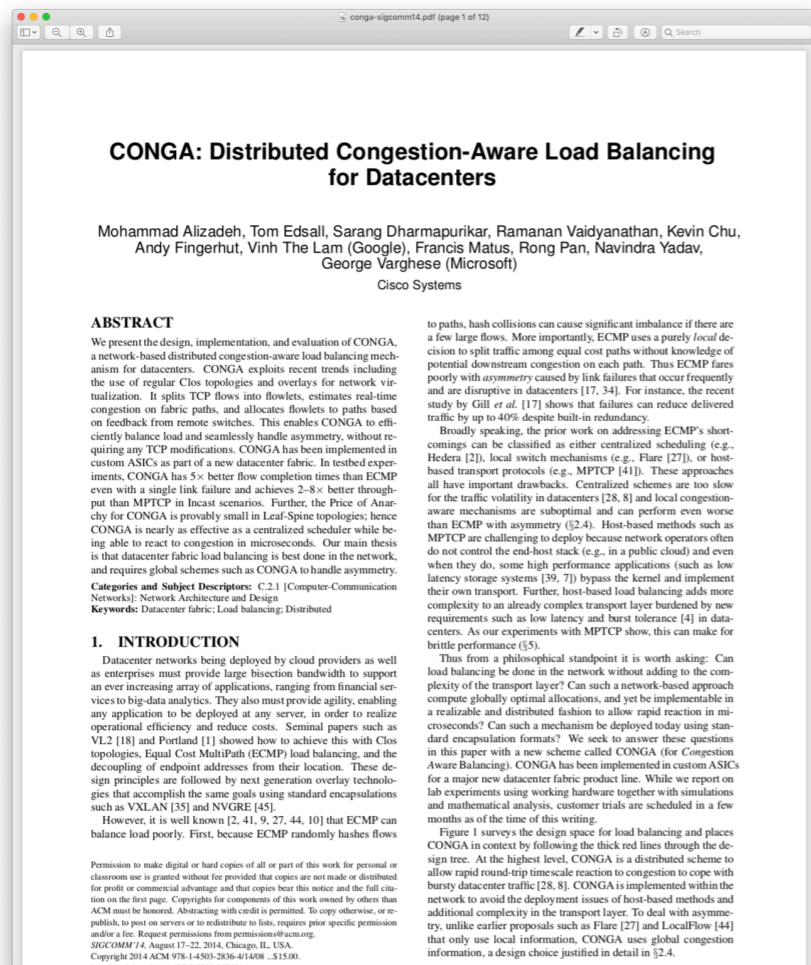
Correctness

Management

# A large set of papers on programmable data planes aim at improving performance, esp. load balancing

HULA [SOSR'16]

DRILL [SIGCOMM'17]

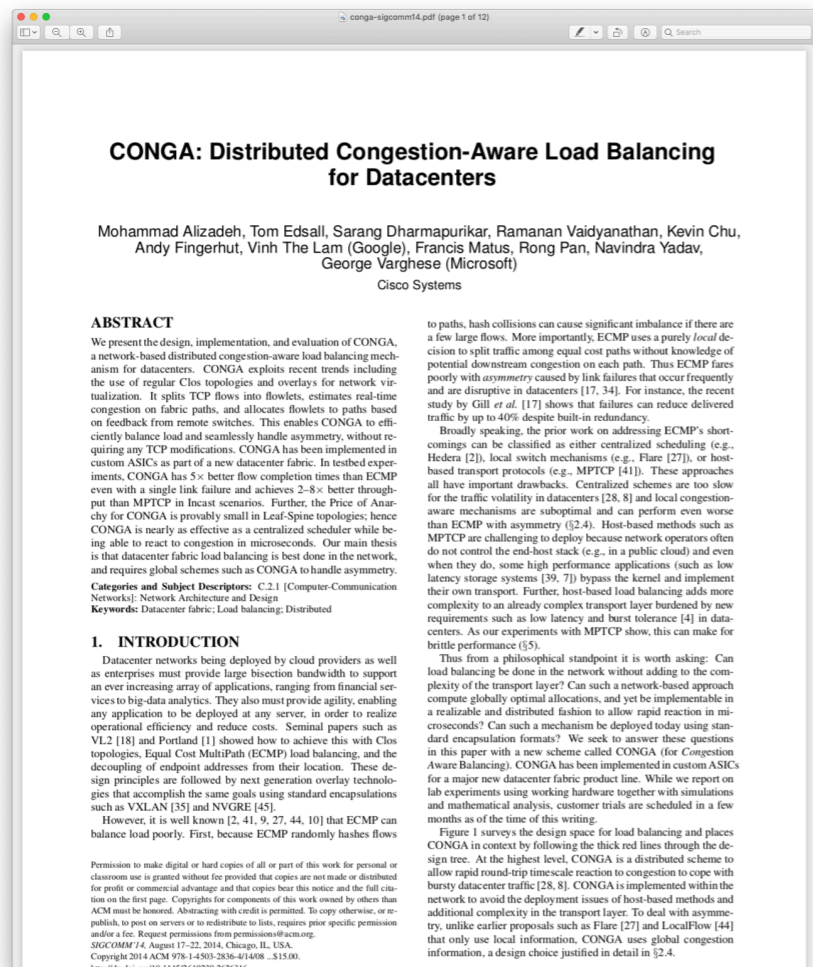


CONGA [SIGCOMM'14]



LetFlow [NSDI'17]

# A large set of papers on programmable data planes aim at improving performance, esp. load balancing



CONGA [SIGCOMM'14]



LetFlow [NSDI'17]

HULA [SOSR'16]

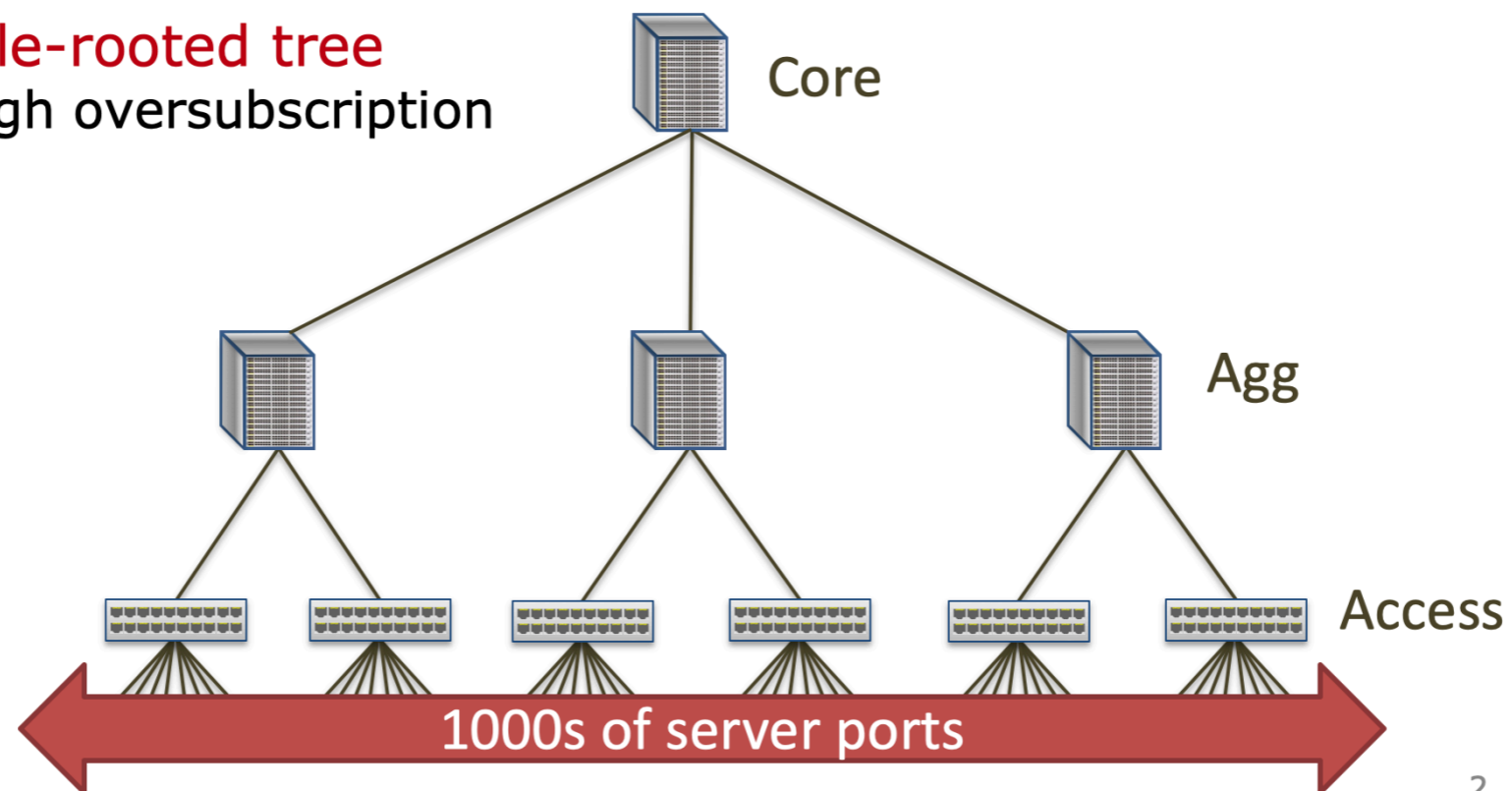
DRILL [SIGCOMM'17]

# Motivation

DC networks need large bisection bandwidth for **distributed** apps (big data, HPC, web services, etc)

## Single-rooted tree

- High oversubscription



2

Source: CONGA: Distributed Congestion-Aware Load Balancing for Datacenters, Mohammad Alizadeh et al., 2014

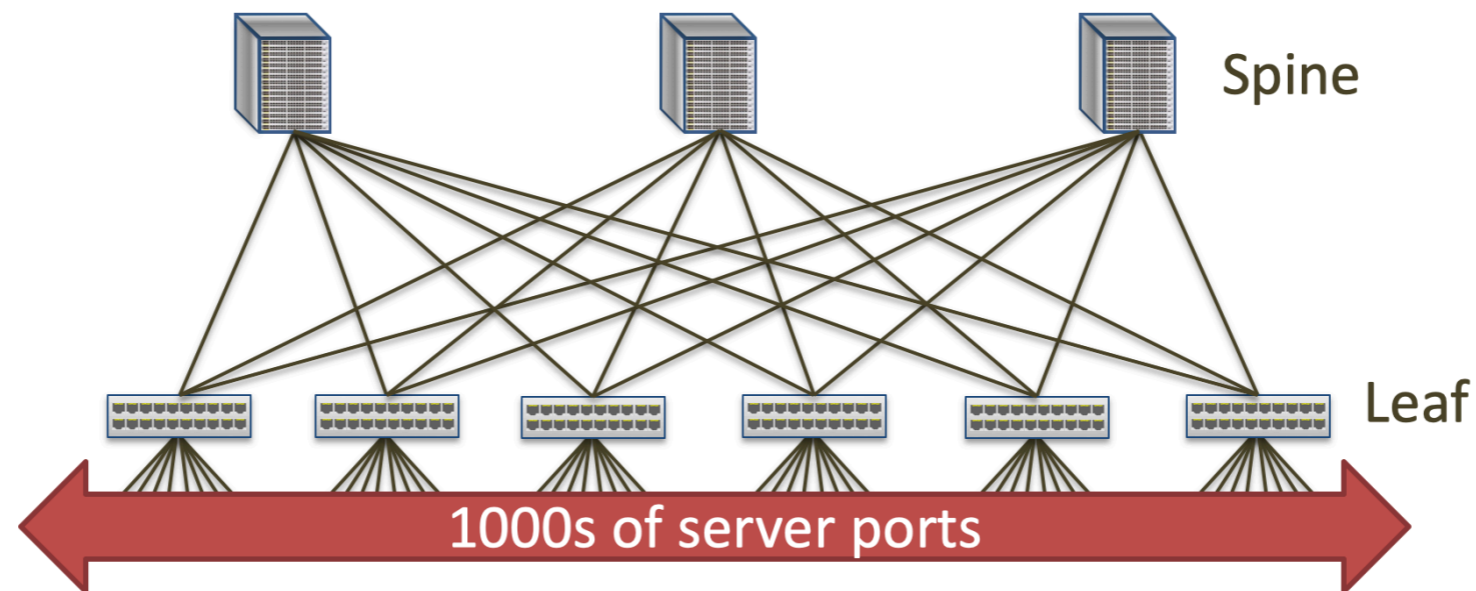
# Motivation

---

DC networks need large bisection bandwidth for **distributed** apps (big data, HPC, web services, etc)

**Multi-rooted tree [Fat-tree, Leaf-Spine, ...]**

- Full bisection bandwidth, achieved via multipathing

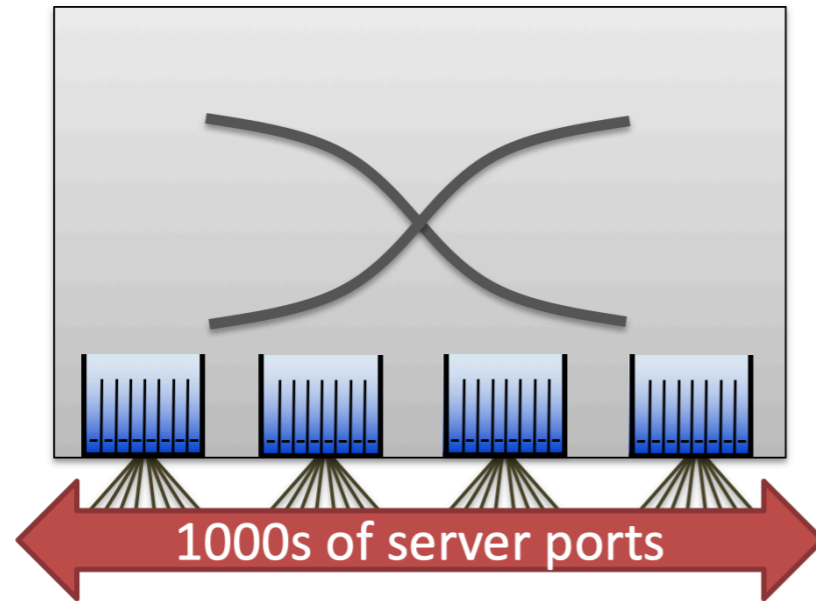


2

Source: CONGA: Distributed Congestion-Aware Load Balancing for Datacenters, Mohammad Alizadeh et al., 2014

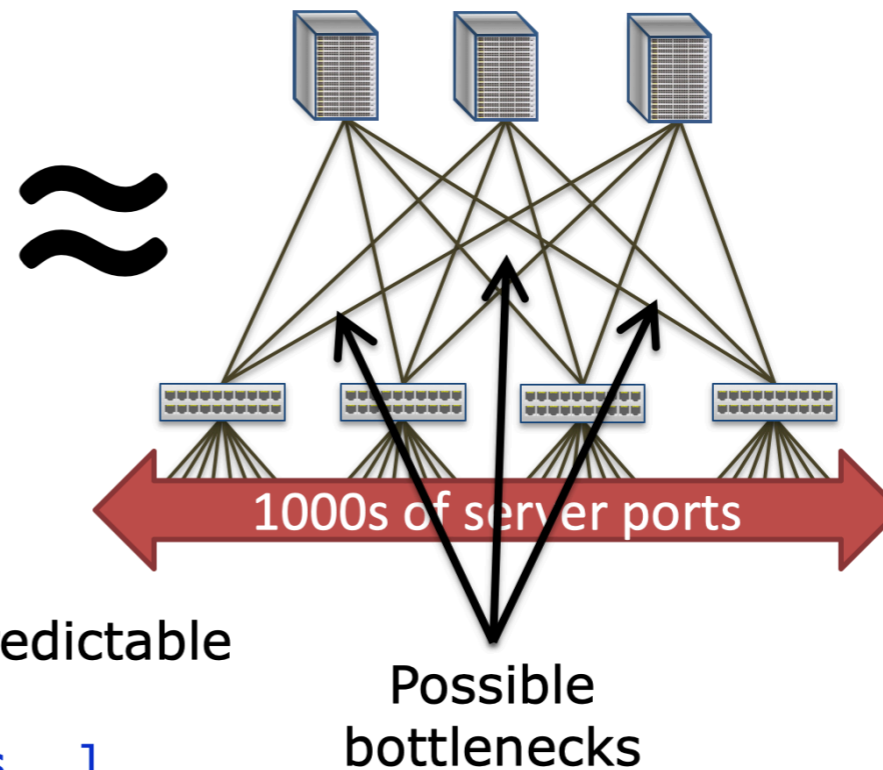
# Multi-rooted != Ideal DC Network

Ideal DC network:  
Big output-queued switch



- No internal bottlenecks → predictable
- Simplifies BW management  
[EyeQ, FairCloud, pFabric, Varys, ...]

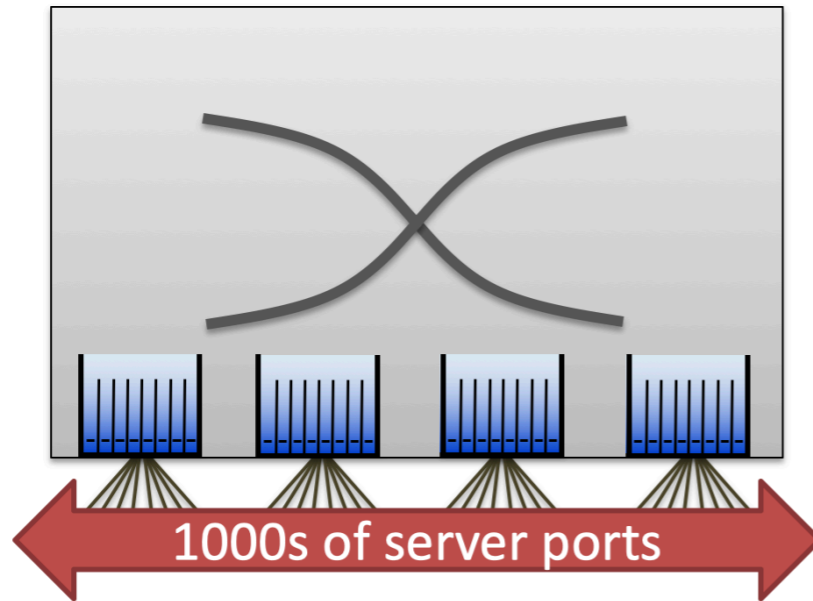
Multi-rooted tree



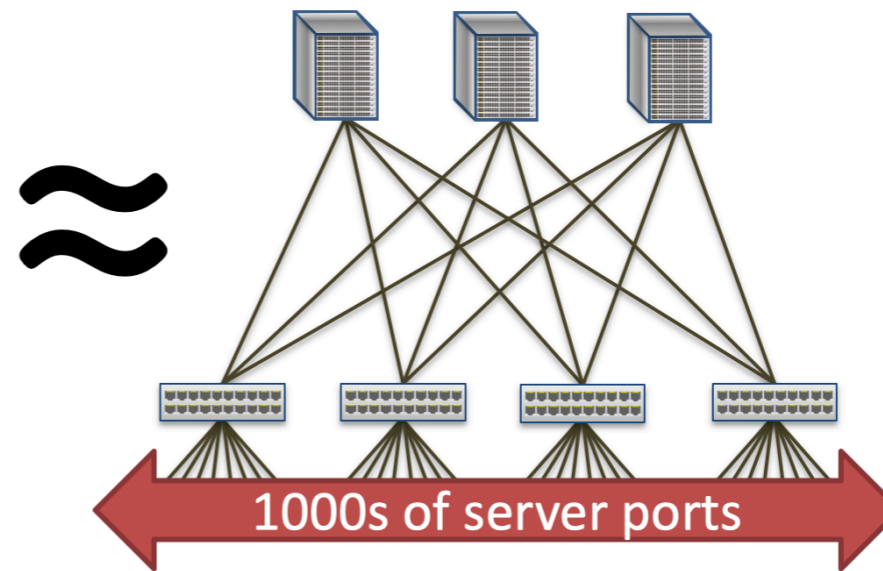


# Multi-rooted $\neq$ Ideal DC Network

Ideal DC network:  
Big output-queued switch



Multi-rooted tree



Need precise load balancing

# Today: ECMP Load Balancing

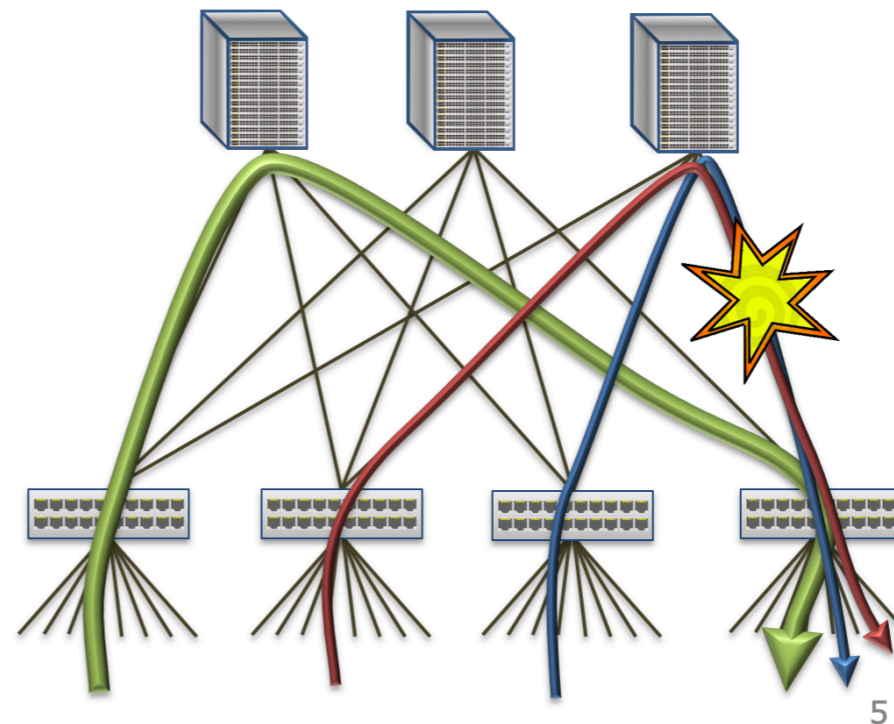
---

Pick among equal-cost paths by a **hash** of 5-tuple

- Approximates Valiant load balancing
- Preserves packet order

## Problems:

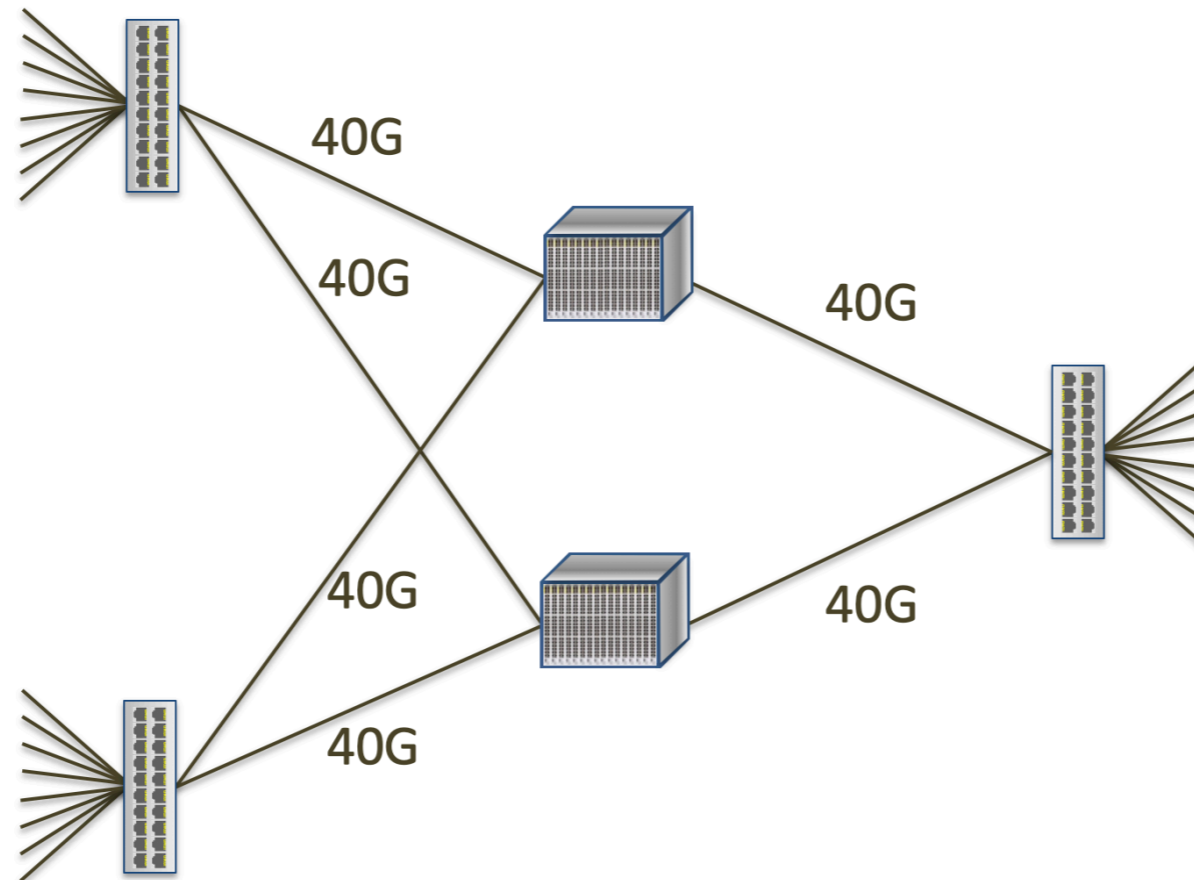
- Hash collisions  
(coarse granularity)
- Local & stateless  
(v. bad with asymmetry  
due to link failures)



# Dealing with Asymmetry

---

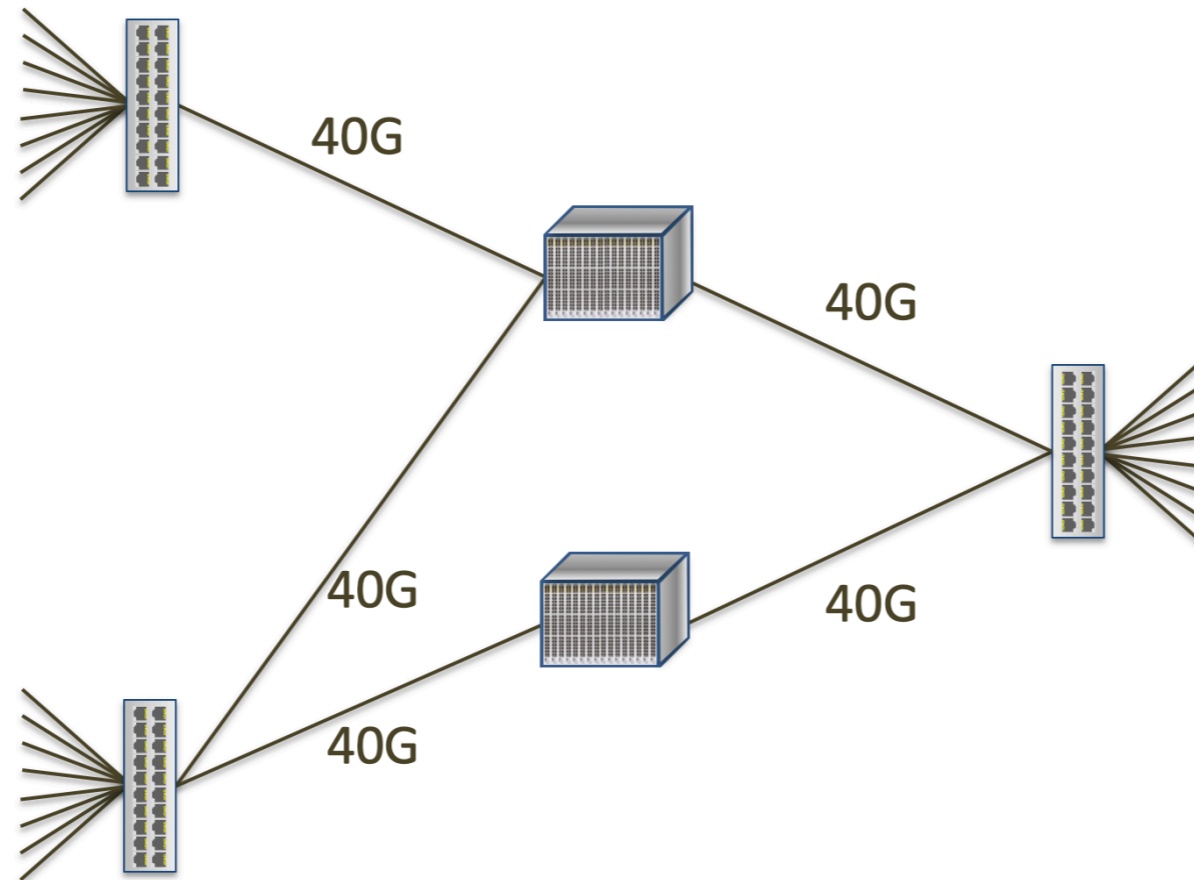
Handling asymmetry needs non-local knowledge



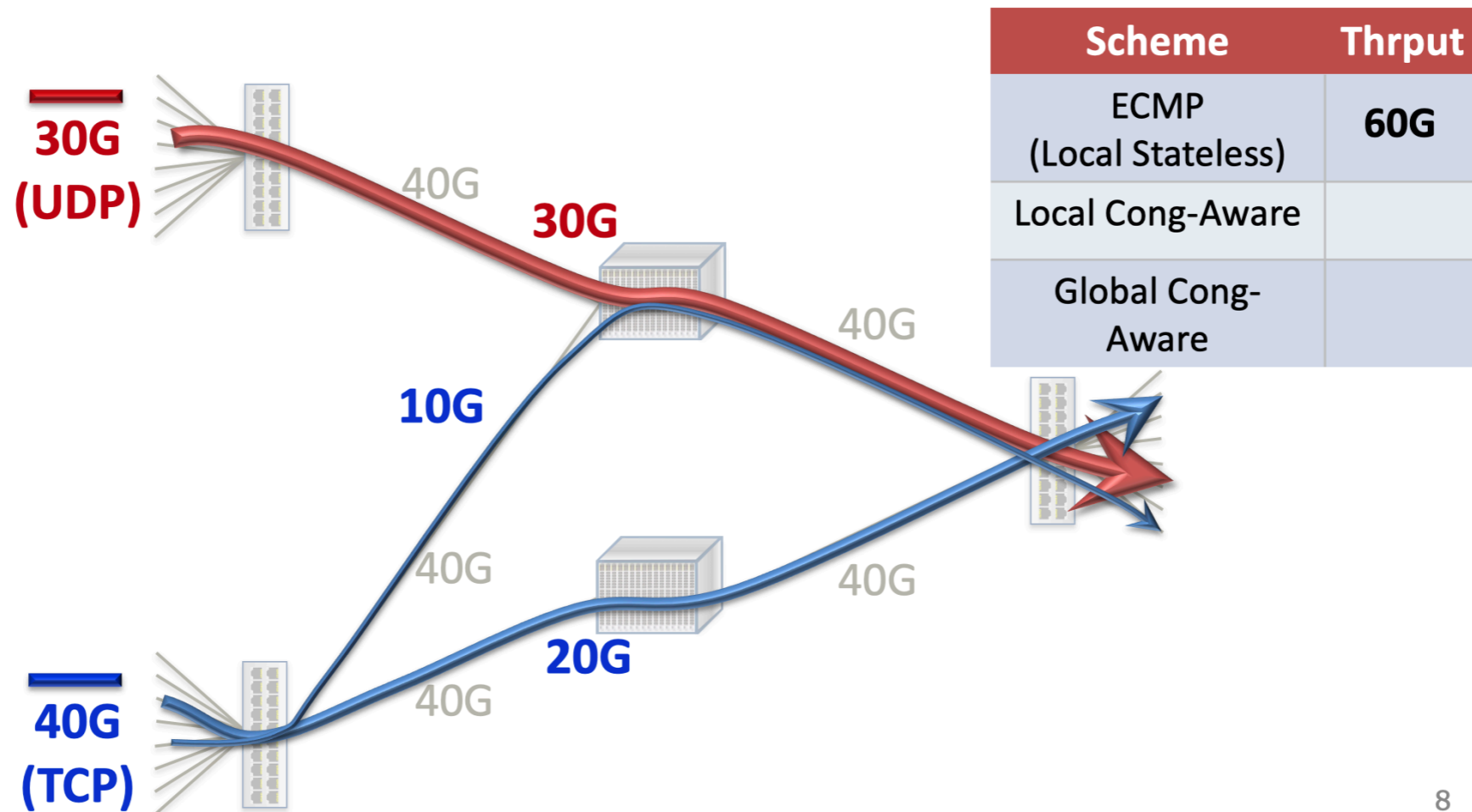
# Dealing with Asymmetry

---

Handling asymmetry needs non-local knowledge

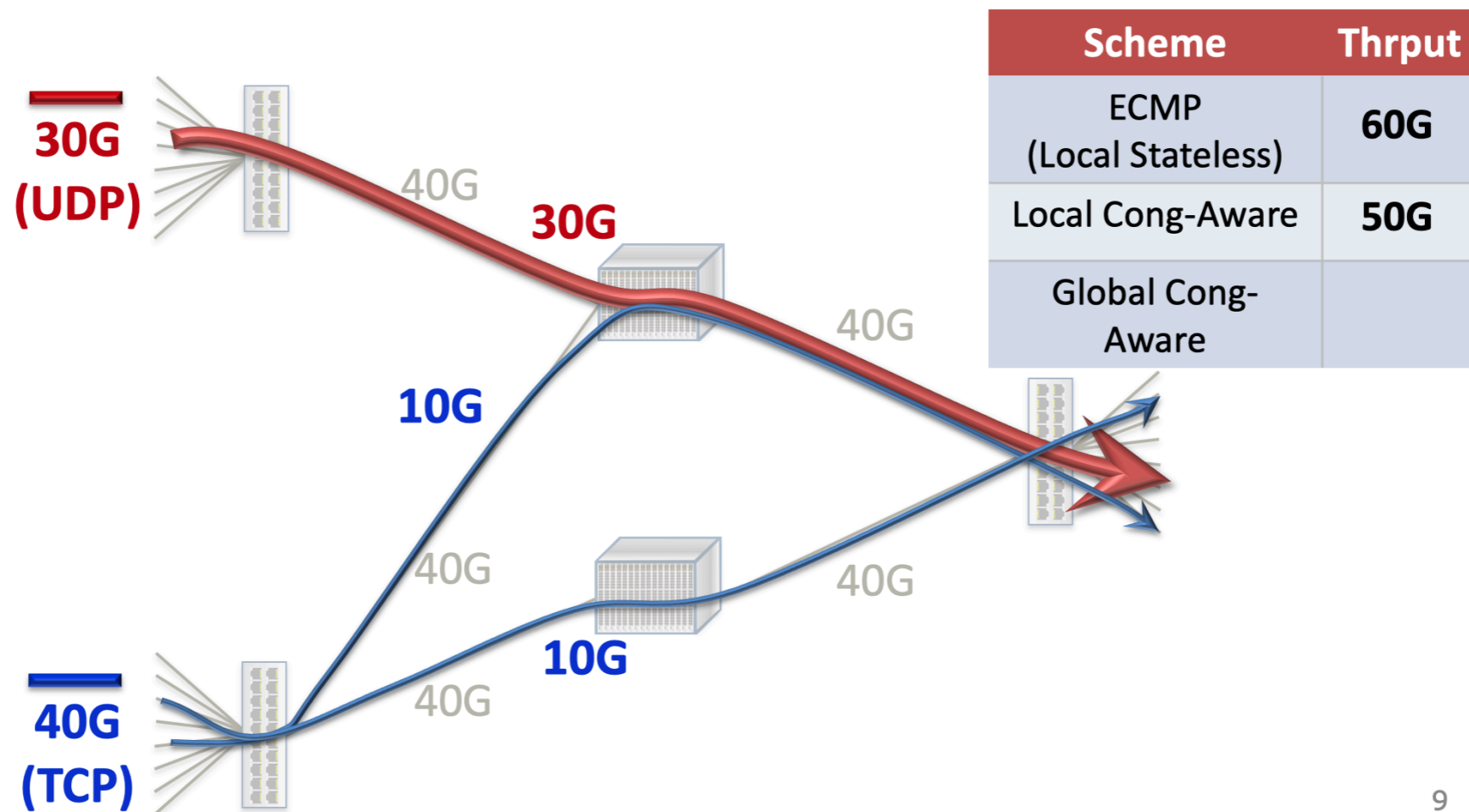


# Dealing with Asymmetry: ECMP



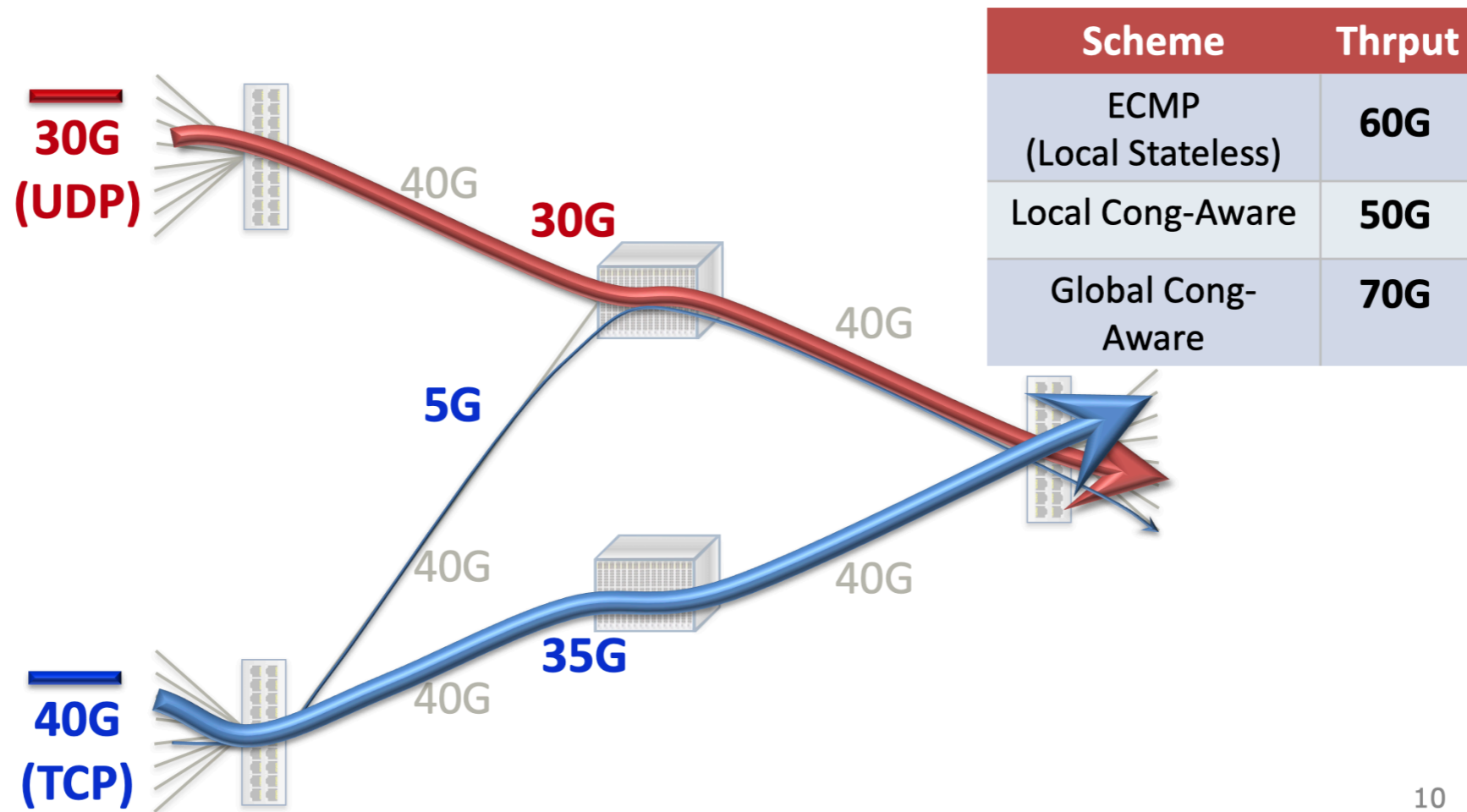
Source: CONGA: Distributed Congestion-Aware Load Balancing for Datacenters, Mohammad Alizadeh et al., 2014

# Dealing with Asymmetry: Local Congestion-Aware



Source: CONGA: Distributed Congestion-Aware Load Balancing for Datacenters,  
Mohammad Alizadeh et al., 2014

# Dealing with Asymmetry: Global Congestion-Aware



Source: CONGA: Distributed Congestion-Aware Load Balancing for Datacenters, Mohammad Alizadeh et al., 2014

# Global Congestion-Awareness (in Datacenters)

---

		Datacenter
Opportunity →	{ Latency Topology	microseconds simple, regular
Challenge →	Traffic	volatile, bursty



# Global Congestion-Awareness (in Datacenters)

---

		Datacenter
Opportunity →	{ Latency Topology	microseconds simple, regular
Challenge →	Traffic	volatile, bursty

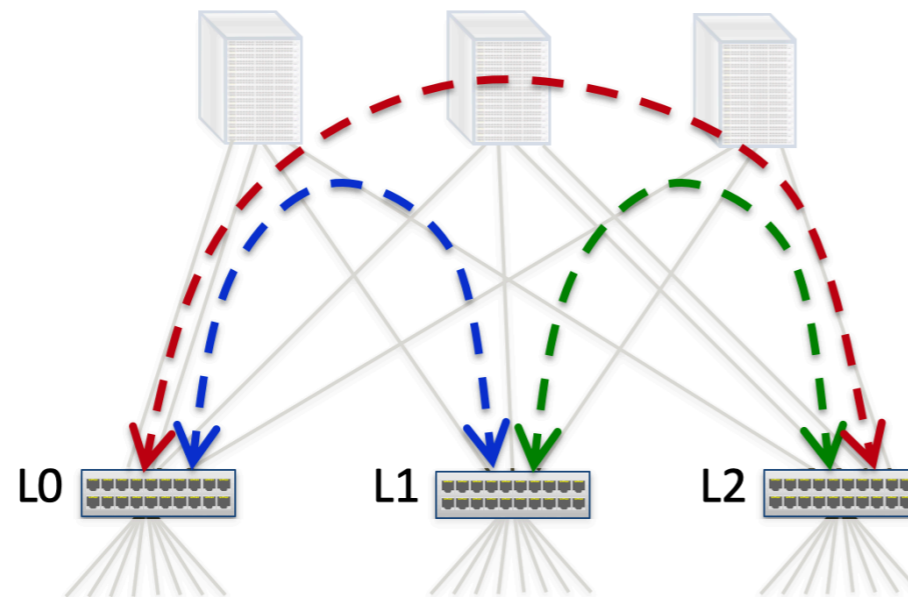
## Key Insight:

Use *extremely* fast, low latency distributed control

# CONGA in 1 Slide

---

1. Leaf switches (top-of-rack) track congestion to other leaves on different paths **in near real-time**
1. Use greedy decisions to minimize bottleneck util



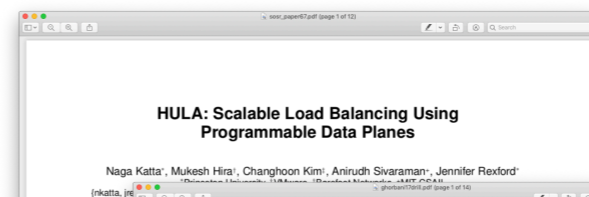
Fast feedback loops  
between leaf switches,  
**directly in dataplane**

# A large set of papers on programmable data planes aim at improving performance, esp. load balancing

P4-based data-plane load-balancing with better scalability than CONGA

"micro" load balancing, packet-by-packet, can deal with micro-bursts

HULA [SOSR'16]



ABSTRACT

Previous research on P4-based load balancing has focused on coarse-grained, per-flow load balancing. These techniques are limited by the tracking state at the leaf switches. In this paper, we propose HULA, a P4-based load balancing scheme that achieves per-packet load balancing. HULA uses a novel P4 program that tracks the state of each packet at the leaf switches. This allows HULA to balance traffic on a per-packet basis, which is essential for handling micro-bursts. HULA achieves a 3x speedup over CONGA and a 10x speedup over P4-based load balancing schemes that use per-flow load balancing.

CCS CONCEPTS

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

DRILL: Micro Load Balancing for Low-latency Data Center Networks

Soudesh Ghorbani, Zhibin Yang, P. Brighten Godfrey

University of Wisconsin-Madison, University of Illinois at Urbana-Champaign, University of Illinois at Urbana-Champaign

ABSTRACT

The trend towards multi-tenant data centers has led to a new class of applications that require low-latency, high-throughput data center networks. These applications are often sensitive to network congestion and require fine-grained load balancing. In this paper, we propose DRILL, a micro load balancing scheme for low-latency data center networks. DRILL uses a novel P4 program that tracks the state of each packet at the leaf switches. This allows DRILL to balance traffic on a per-packet basis, which is essential for handling micro-bursts. DRILL achieves a 3x speedup over CONGA and a 10x speedup over P4-based load balancing schemes that use per-flow load balancing.

CCS CONCEPTS

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

Networks

DRILL [SIGCOMM'17]

Let it Flow: Resilient Asymmetric Load Balancing with Flowlet Switching

Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, Tom Edsall

Cisco Systems, Massachusetts Institute of Technology

ABSTRACT

Datacenter networks require efficient multi-path load balancing to achieve high bisection bandwidth. Despite much progress in recent years towards addressing this challenge, a load balancing design that is both simple to implement and resilient to network asymmetry has remained elusive. In this paper, we show that flowlet switching, an idea first proposed more than a decade ago, is a powerful technique for resilient load balancing with asymmetry. Flowlets have a remarkable elasticity property: their size changes automatically based on traffic conditions on their path. We use this insight to develop LetFlow, a very simple load balancing scheme that is resilient to asymmetry. LetFlow simply picks paths at random for flowlets and lets their elasticity naturally balance the traffic on different paths. Our extensive evaluation with real hardware and packet-level simulation shows that LetFlow is very effective. Despite being much simpler, it performs significantly better than other traffic-oblivious schemes like WCMP and Preso in asymmetric scenarios, while achieving average flow completion time within 10-20% of CONGA in tested experiments and 2x of CONGA in simulated topologies with large asymmetry and heavy traffic load.

1 Introduction

Datacenter networks must provide large bisection bandwidth to support the increasing traffic demands of applications such as big-data analytics, web services, and cloud storage. They achieve this by load balancing traffic over many paths in multi-rooted tree topologies such as Clos [13] and Fat-tree [1]. These designs are widely deployed; for instance, Google has reported on using Clos fabrics with more than 1 Pops of bisection bandwidth in its datacenters [25].

The standard load balancing scheme in today's datacenters, Equal Cost MultiPath (ECMP) [16], randomly assigns flows to different paths using a hash taken over packet headers. ECMP is widely deployed due to its simplicity but suffers from well-known performance problems such as hash collisions and the inability to adapt to asymmetry in the network topology. A rich body of work [10, 2, 22, 23, 18, 3, 15, 21] has thus emerged on better load balancing designs for datacenter networks.

A defining feature of these designs is the information that they use to make decisions. At one end of the spectrum are designs that are oblivious to traffic conditions [16, 10, 9, 15] or rely only on local measurements [24, 20] at the switches. ECMP and Preso [15], which picks paths in round-robin fashion for fixed-sized chunks of data (called "flowlets"), fall in this category. At the other extreme are designs [2, 22, 23, 18, 3, 21, 29] that use knowledge of traffic conditions and congestion on different paths to make decisions. Two recent examples are CONGA [3] and HULA [21], which use feedback between the switches to gather path-wise congestion information and shift traffic to less-congested paths. Load balancing schemes that require path congestion information, naturally, are much more complex. Current designs either use a centralized fabric controller [2, 8, 22] to optimize path choices frequently or require non-trivial mechanisms, at the end-hosts [23, 18] or switches [3, 21, 30], to implement end-to-end or hop-by-hop feedback. On the other hand, schemes that lack visibility into path congestion have a key drawback: they perform poorly in asymmetric topologies [3]. As we discuss in §2.1, the reason is that the optimal traffic split across asymmetric paths depends on (dynamically varying) traffic conditions; hence, traffic-oblivious schemes are fundamentally unable to make optimal decisions and can perform poorly in asymmetric topologies.

Asymmetry is common in practice for a variety of reasons, such as link failures and heterogeneity in network equipment [31, 12, 3]. Handling asymmetry gracefully, therefore, is important. This raises the question: are there simple load balancing schemes that are resilient to asymmetry? In this paper, we answer this question in the affirmative by developing LetFlow, a simple scheme that requires no state to make load balancing decisions, and yet it is very resilient to network asymmetry.

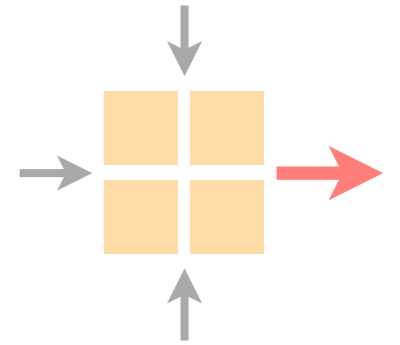
LetFlow is extremely simple: switches pick a path at random for each flowlet. That's it! A flowlet is a burst of packets that is separated in time from other bursts by a sufficient gap—called the "flowlet timeout". Flowlet switching [27, 20] was proposed over a decade ago as a way to split TCP flows across multiple paths without causing packet reordering. Remarkably, as we uncover in this paper, flowlet switching is also a powerful technique

stateless, yet congestion-aware load-balancing decision

LetFlow [NSDI'17]

# Advanced Topics in Communication Networks

## Programming Network Data Planes



Laurent Vanbever

[nsg.ee.ethz.ch](http://nsg.ee.ethz.ch)

ETH Zürich

Oct 22 2019