Advanced Topics in Communication Networks

Laurent Vanbever

nsg.ee.ethz.ch

ETH Zürich

Tue 11 Oct 2022

Last week on

Advanced Topics in Communication Networks

We finished our exploration of the P4 language then looked at our first technique: load balancing

stateful objects

How do we maintain state in P4?

load balancing

How do we balance traffic in a network?

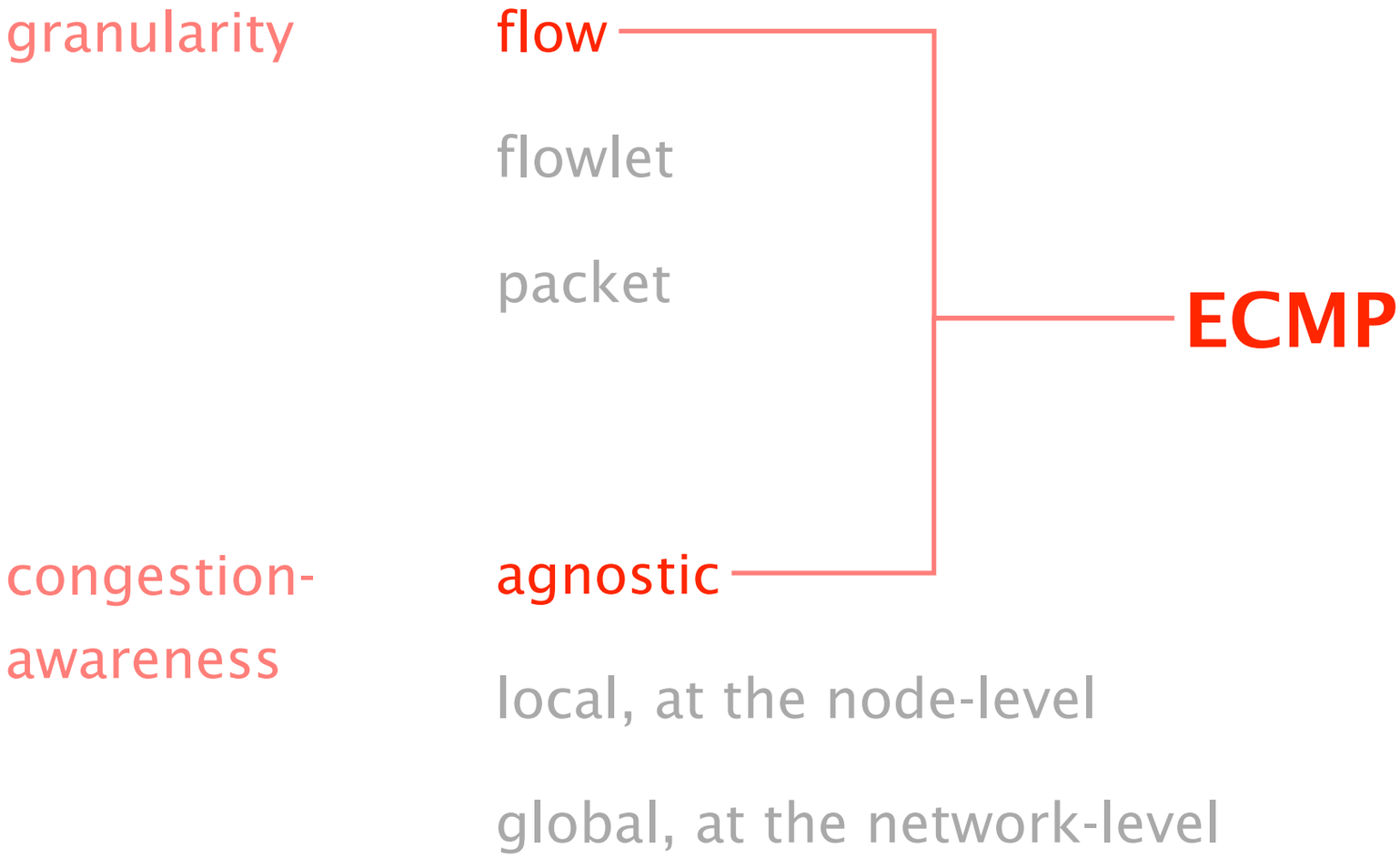
stateful objects

load balancing

How do we maintain state in P4?

How do we balance traffic in a network?

ECMP load-balancing decisions are suboptiomal because they are too coarse-grained and congestion-agnostic



load-balancing decision…

granularity

~~flow~~

flowlet

packet

congestion-awareness

~~agnostic~~

local, at the node-level

global, at the network-level

load-balancing decision…

granularity

~~flow~~

flowlet

packet —— finest granularity… but *bad* for TCP because of reordering

congestion-awareness

~~agnostic~~

local, at the node-level

global, at the network-level

load-balancing decision…

granularity

~~flow~~

flowlet

~~packet~~

congestion-awareness

~~agnostic~~

local, at the node-level

global, at the network-level

load-balancing decision…

granularity

~~flow~~

flowlet

~~packet~~

congestion-awareness

~~agnostic~~

~~local~~, at the node-level

handling asymmetry mandates non-local knowledge

global, at the network-level

load-balancing decision…

granularity

~~flow~~

flowlet **best pick!**

~~packet~~

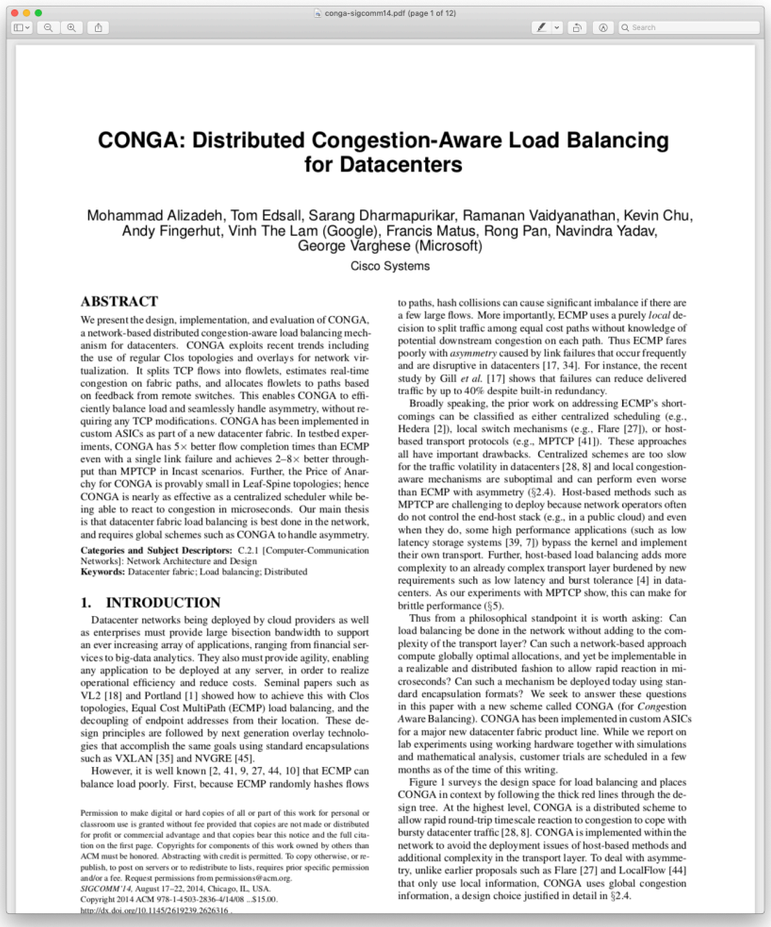
congestion-awareness

~~agnostic~~

~~local, at the node-level~~

global **best pick!**

We started to look at how to implement flowlet-based, congestion-aware load balancing



conga-sigcomm14.pdf (page 1 of 12)

CONGA: Distributed Congestion-Aware Load Balancing for Datacenters

Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam (Google), Francis Matus, Rong Pan, Navindra Yadav, George Varghese (Microsoft)

Cisco Systems

ABSTRACT

We present the design, implementation, and evaluation of CONGA, a network-based distributed congestion-aware load balancing mechanism for datacenters. CONGA exploits recent trends including the use of regular Clos topologies and overlays for network virtualization. It splits TCP flows into flowlets, estimates real-time congestion on fabric paths, and allocates flowlets to paths based on feedback from remote switches. This enables CONGA to efficiently balance load and seamlessly handle asymmetry, without requiring any TCP modifications. CONGA has been implemented in custom ASICs as part of a new datacenter fabric. In testbed experiments, CONGA has 5× better flow completion times than ECMP even with a single link failure and achieves 2–8× better throughput than MPTCP in Incast scenarios. Further, the Price of Anarchy for CONGA is provably small in Leaf-Spine topologies; hence CONGA is nearly as effective as a centralized scheduler while being able to react to congestion in microseconds. Our main thesis is that datacenter fabric load balancing is best done in the network, and requires global schemes such as CONGA to handle asymmetry.

Categories and Subject Descriptors: C.2.1 [Computer-Communication Networks]: Network Architecture and Design

Keywords: Datacenter fabric; Load balancing; Distributed

1. INTRODUCTION

Datacenter networks being deployed by cloud providers as well as enterprises must provide large bisection bandwidth to support an ever increasing array of applications, ranging from financial services to big-data analytics. They also must provide agility, enabling any application to be deployed at any server, in order to realize operational efficiency and reduce costs. Seminal papers such as VL2 [18] and Portland [1] showed how to achieve this with Clos topologies, Equal Cost MultiPath (ECMP) load balancing, and the decoupling of endpoint addresses from their location. These design principles are followed by next generation overlay technologies that accomplish the same goals using standard encapsulations such as VXLAN [35] and NVGRE [45].

However, it is well known [2, 41, 9, 27, 44, 10] that ECMP can balance load poorly. First, because ECMP randomly hashes flows to paths, hash collisions can cause significant imbalance if there are a few large flows. More importantly, ECMP uses a purely *local* decision to split traffic among equal cost paths without knowledge of potential downstream congestion on each path. Thus ECMP fares poorly with *asymmetry* caused by link failures that occur frequently and are disruptive in datacenters [17, 34]. For instance, the recent study by Gill *et al.* [17] shows that failures can reduce delivered traffic by up to 40% despite built-in redundancy.

Broadly speaking, the prior work on addressing ECMP's shortcomings can be classified as either centralized scheduling (e.g., Hedera [2]), local switch mechanisms (e.g., Flare [27]), or host-based transport protocols (e.g., MPTCP [41]). These approaches all have important drawbacks. Centralized schemes are too slow for the traffic volatility in datacenters [28, 8] and local congestion-aware mechanisms are suboptimal and can perform even worse than ECMP with asymmetry (§2.4). Host-based methods such as MPTCP are challenging to deploy because network operators often do not control the end-host stack (e.g., in a public cloud) and even when they do, some high performance applications (such as low latency storage systems [39, 7]) bypass the kernel and implement their own transport. Further, host-based load balancing adds more complexity to an already complex transport layer burdened by new requirements such as low latency and burst tolerance [4] in datacenters. As our experiments with MPTCP show, this can make for brittle performance (§5).

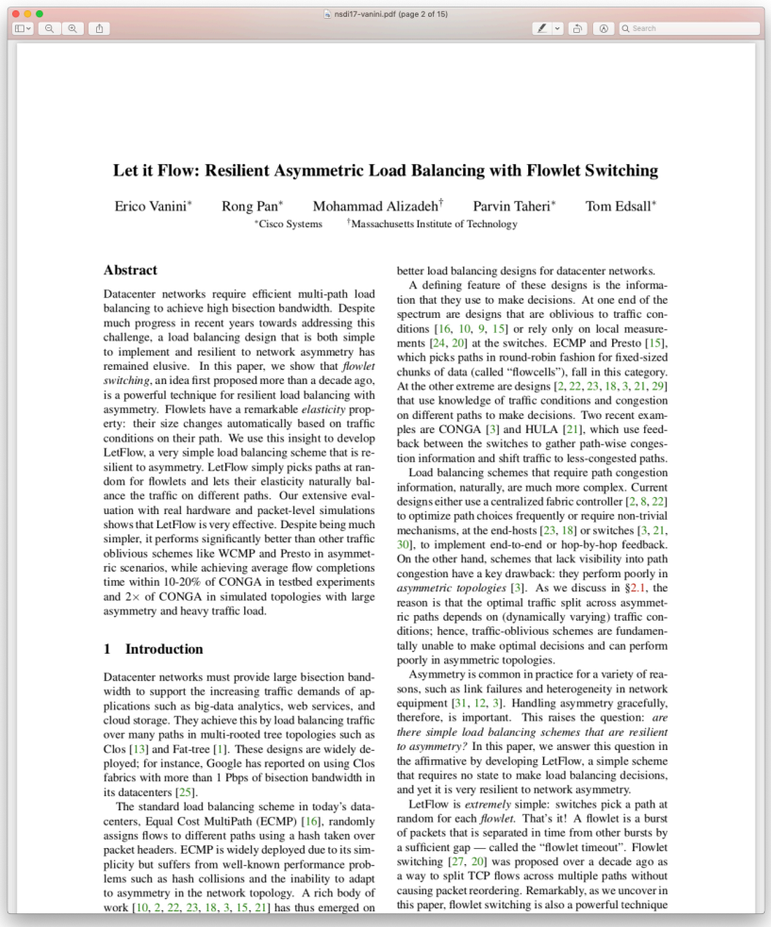
Thus from a philosophical standpoint it is worth asking: Can load balancing be done in the network without adding to the complexity of the transport layer? Can such a network-based approach compute globally optimal allocations, and yet be implementable in a realizable and distributed fashion to allow rapid reaction in microseconds? Can such a mechanism be deployed today using standard encapsulation formats? We seek to answer these questions in this paper with a new scheme called CONGA (for *Con*gestion Aware Balancing). CONGA has been implemented in custom ASICs for a major new datacenter fabric product line. While we report on lab experiments using working hardware together with simulations and mathematical analysis, customer trials are scheduled in a few months as of the time of this writing.

Figure 1 surveys the design space for load balancing and places CONGA in context by following the thick red lines through the design tree. At the highest level, CONGA is a distributed scheme to allow rapid round-trip timescale reaction to congestion to cope with bursty datacenter traffic [28, 8]. CONGA is implemented within the network to avoid the deployment issues of host-based methods and additional complexity in the transport layer. To deal with asymmetry, unlike earlier proposals such as Flare [27] and LocalFlow [44] that only use local information, CONGA uses global congestion information, a design choice justified in detail in §2.4.

SIGCOMM'14, August 17–22, 2014, Chicago, IL, USA.

http://dx.doi.org/10.1145/2619239.2626316

CONGA [SIGCOMM'14]



nsdi17-vanini.pdf (page 2 of 15)

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 simulations shows that LetFlow is very effective. Despite being much simpler, it performs significantly better than other traffic oblivious schemes like WCMP and Presto in asymmetric scenarios, while achieving average flow completions time within 10-20% of CONGA in testbed experiments and 2× 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 Pbps 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 Presto [15], which picks paths in round-robin fashion for fixed-sized chunks of data (called "flowcells"), 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

LetFlow [NSDI'17]

CONGA: Distributed Congestion-Aware Load Balancing for Datacenters

Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam (Google), Francis Matus, Rong Pan, Navindra Yadav, George Varghese (Microsoft)

Cisco Systems

ABSTRACT

We present the design, implementation, and evaluation of CONGA, a network-based distributed congestion-aware load balancing mechanism for datacenters. CONGA exploits recent trends including the use of regular Clos topologies and overlays for network virtualization. It splits TCP flows into flowlets, estimates real-time congestion on fabric paths, and allocates flowlets to paths based on feedback from remote switches. This enables CONGA to efficiently balance load and seamlessly handle asymmetry, without requiring any TCP modifications. CONGA has been implemented in custom ASICs as part of a new datacenter fabric. In testbed experiments, CONGA has 5× better flow completion times than ECMP even with a single link failure and achieves 2–8× better throughput than MPTCP in Incast scenarios. Further, the Price of Anarchy for CONGA is provably small in Leaf-Spine topologies; hence CONGA is nearly as effective as a centralized scheduler while being able to react to congestion in microseconds. Our main thesis is that datacenter fabric load balancing is best done in the network, and requires global schemes such as CONGA to handle asymmetry.

Categories and Subject Descriptors: C.2.1 [Computer-Communication Networks]: Network Architecture and Design
Keywords: Datacenter fabric; Load balancing; Distributed

1. INTRODUCTION

Datacenter networks being deployed by cloud providers as well as enterprises must provide large bisection bandwidth to support an ever increasing array of applications, ranging from financial services to big-data analytics. They also must provide agility, enabling any application to be deployed at any server, in order to realize operational efficiency and reduce costs. Seminal papers such as VL2 [18] and Portland [1] showed how to achieve this with Clos topologies, Equal Cost MultiPath (ECMP) load balancing, and the decoupling of endpoint addresses from their location. These design principles are followed by next generation overlay technologies that accomplish the same goals using standard encapsulations such as VXLAN [35] and NVGRE [45].

However, it is well known [2, 41, 9, 27, 44, 10] that ECMP can balance load poorly. First, because ECMP randomly hashes flows to paths, hash collisions can cause significant imbalance if there are a few large flows. More importantly, ECMP uses a purely *local* decision to split traffic among equal cost paths without knowledge of potential downstream congestion on each path. Thus ECMP fares poorly with *asymmetry* caused by link failures that occur frequently and are disruptive in datacenters [17, 34]. For instance, the recent study by Gill *et al.* [17] shows that failures can reduce delivered traffic by up to 40% despite built-in redundancy.

Broadly speaking, the prior work on addressing ECMP's shortcomings can be classified as either centralized scheduling (e.g., Hedera [2]), local switch mechanisms (e.g., Flare [27]), or host-based transport protocols (e.g., MPTCP [41]). These approaches all have important drawbacks. Centralized schemes are too slow for the traffic volatility in datacenters [28, 8] and local congestion-aware mechanisms are suboptimal and can perform even worse than ECMP with asymmetry (§2.4). Host-based methods such as MPTCP are challenging to deploy because network operators often do not control the end-host stack (e.g., in a public cloud) and even when they do, some high performance applications (such as low latency storage systems [39, 7]) bypass the kernel and implement their own transport. Further, host-based load balancing adds more complexity to an already complex transport layer burdened by new requirements such as low latency and burst tolerance [4] in datacenters. As our experiments with MPTCP show, this can make for brittle performance (§5).

Thus from a philosophical standpoint it is worth asking: Can load balancing be done in the network without adding to the complexity of the transport layer? Can such a network-based approach compute globally optimal allocations, and yet be implementable in a realizable and distributed fashion to allow rapid reaction in microseconds? Can such a mechanism be deployed today using standard encapsulation formats? We seek to answer these questions in this paper with a new scheme called CONGA (for *Con*gestion *A*ware Balancing). CONGA has been implemented in custom ASICs for a major new datacenter fabric product line. While we report on lab experiments using working hardware together with simulations and mathematical analysis, customer trials are scheduled in a few months as of the time of this writing.

Figure 1 surveys the design space for load balancing and places CONGA in context by following the thick red lines through the design tree. At the highest level, CONGA is a distributed scheme to allow rapid round-trip timescale reaction to congestion to cope with bursty datacenter traffic [28, 8]. CONGA is implemented within the network to avoid the deployment issues of host-based methods and additional complexity in the transport layer. To deal with asymmetry, unlike earlier proposals such as Flare [27] and LocalFlow [44] that only use local information, CONGA uses global congestion information, a design choice justified in detail in §2.4.

SIGCOMM'14, August 17–22, 2014, Chicago, IL, USA.

http://dx.doi.org/10.1145/2619239.2626316 .

CONGA [SIGCOMM'14]

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 simulations shows that LetFlow is very effective. Despite being much simpler, it performs significantly better than other traffic oblivious schemes like WCMP and Presto in asymmetric scenarios, while achieving average flow completions time within 10-20% of CONGA in testbed experiments and 2× 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 Pbps 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 Presto [15], which picks paths in round-robin fashion for fixed-sized chunks of data (called "flowcells"), 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.

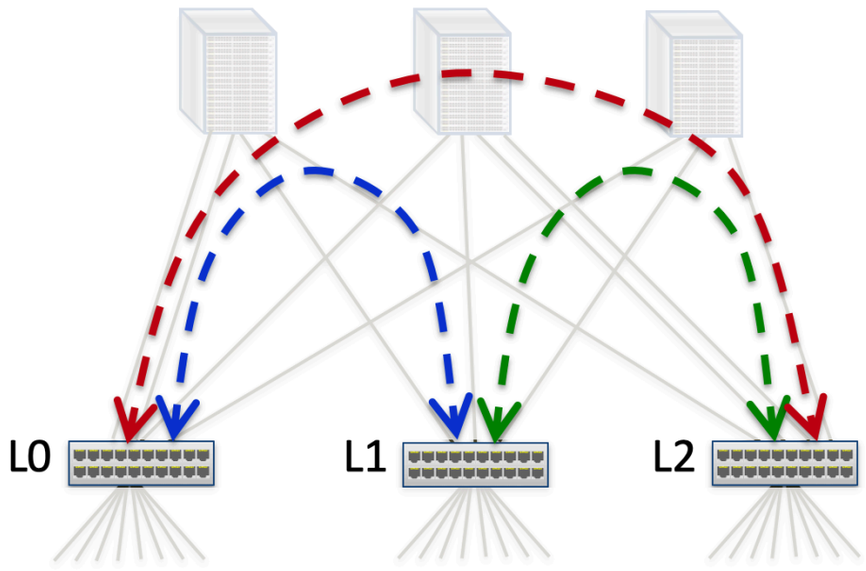
LetFlow is *extremely* simple: switches pick a path at random for each *flowlet*. That's it! A flowlet is a burst of packets that is separated in time from other bursts by a sufficient gap — called the "flowlet timeout". Flowlet switching [27, 20] was proposed over a decade ago as a way to split TCP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

LetFlow [NSDI'17]

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

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

This week on

Advanced Topics in Communication Networks

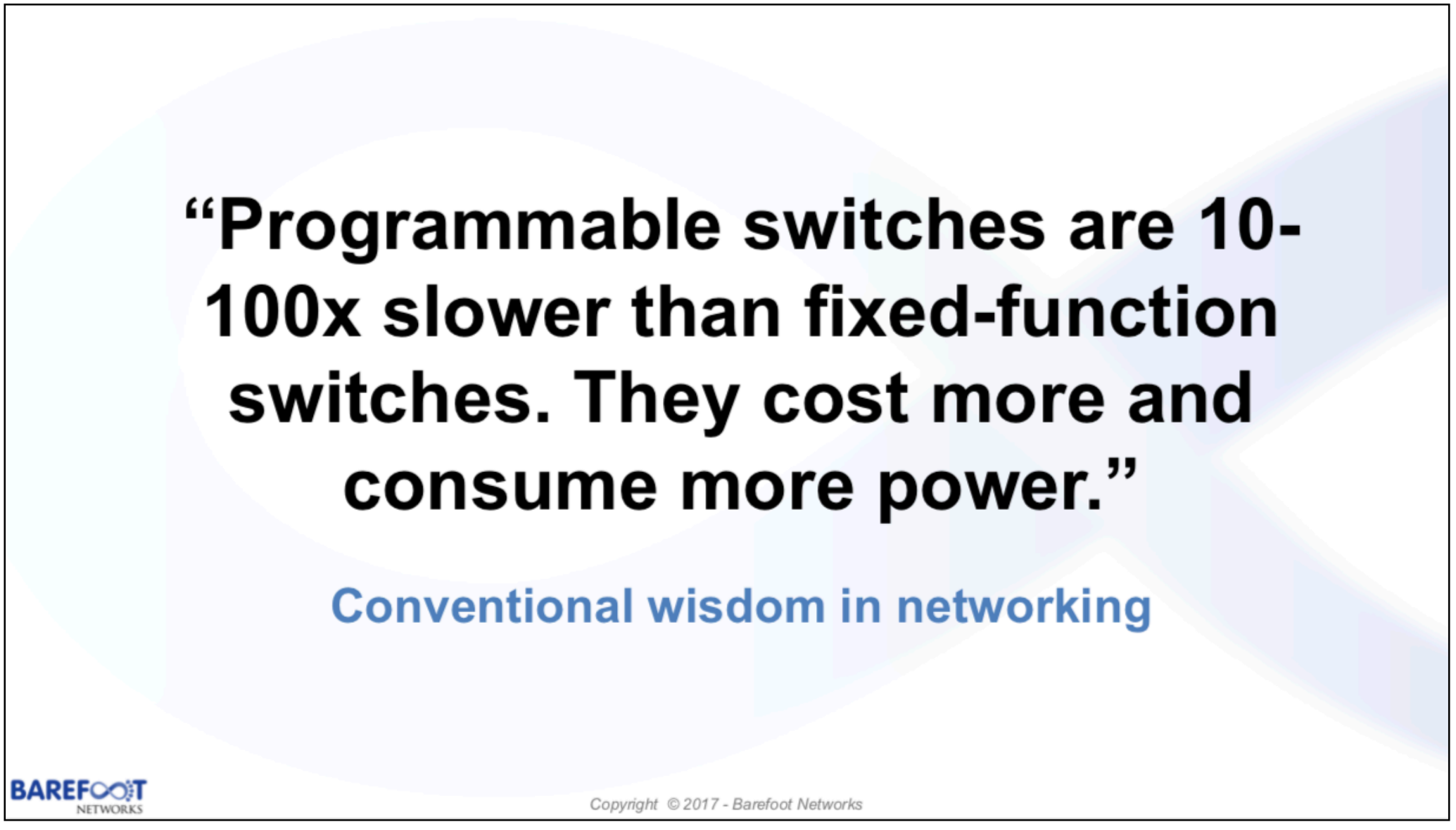
P4 hardware target

Probabilistic data structures

P4 hardware target

Probabilistic data structures

How do we build a *fast* reprogrammable switch?



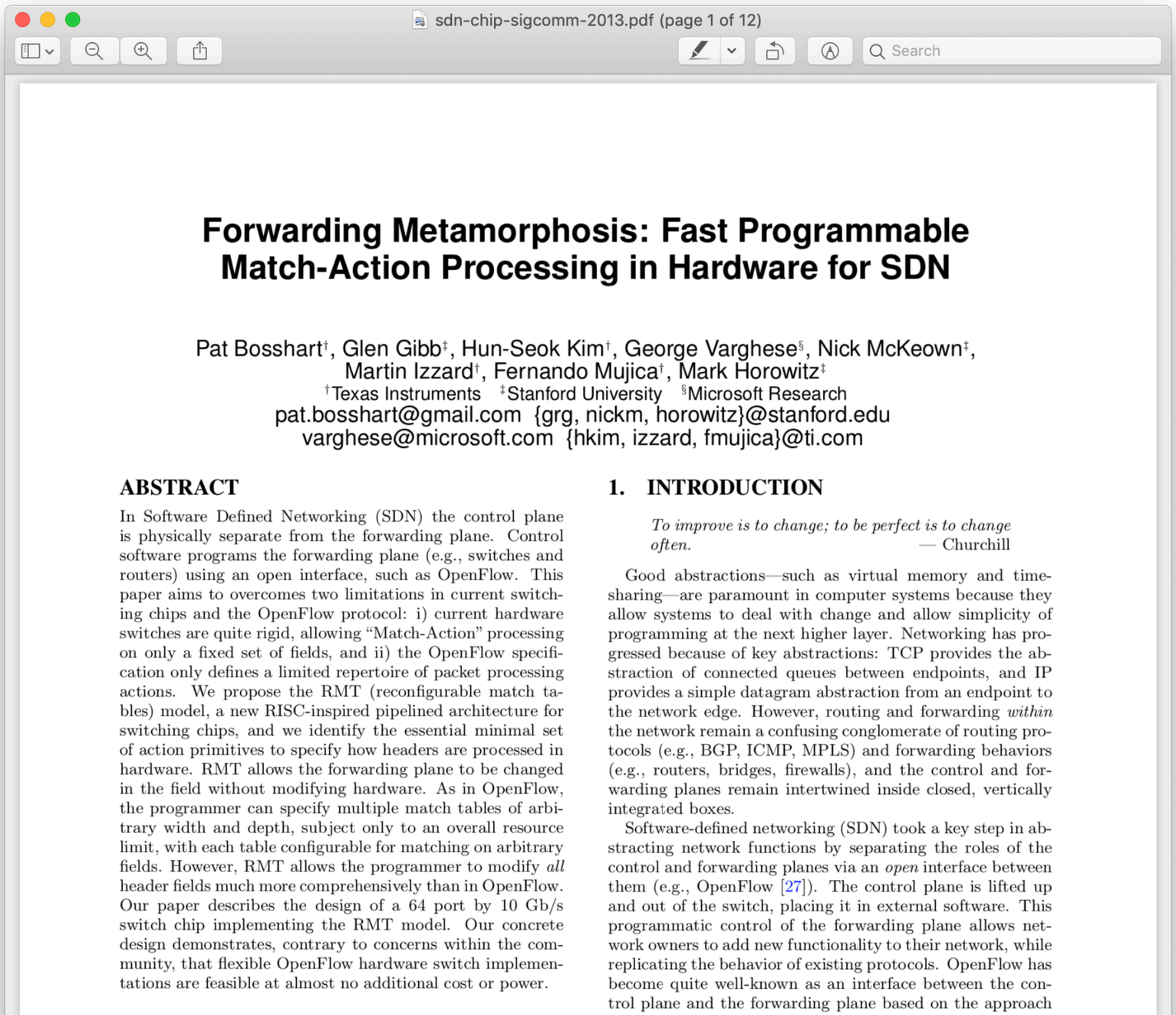
Source: Programmable Data Planes at Terabit Speeds, Vladimir Gurevich, 2017

How can we enable 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)



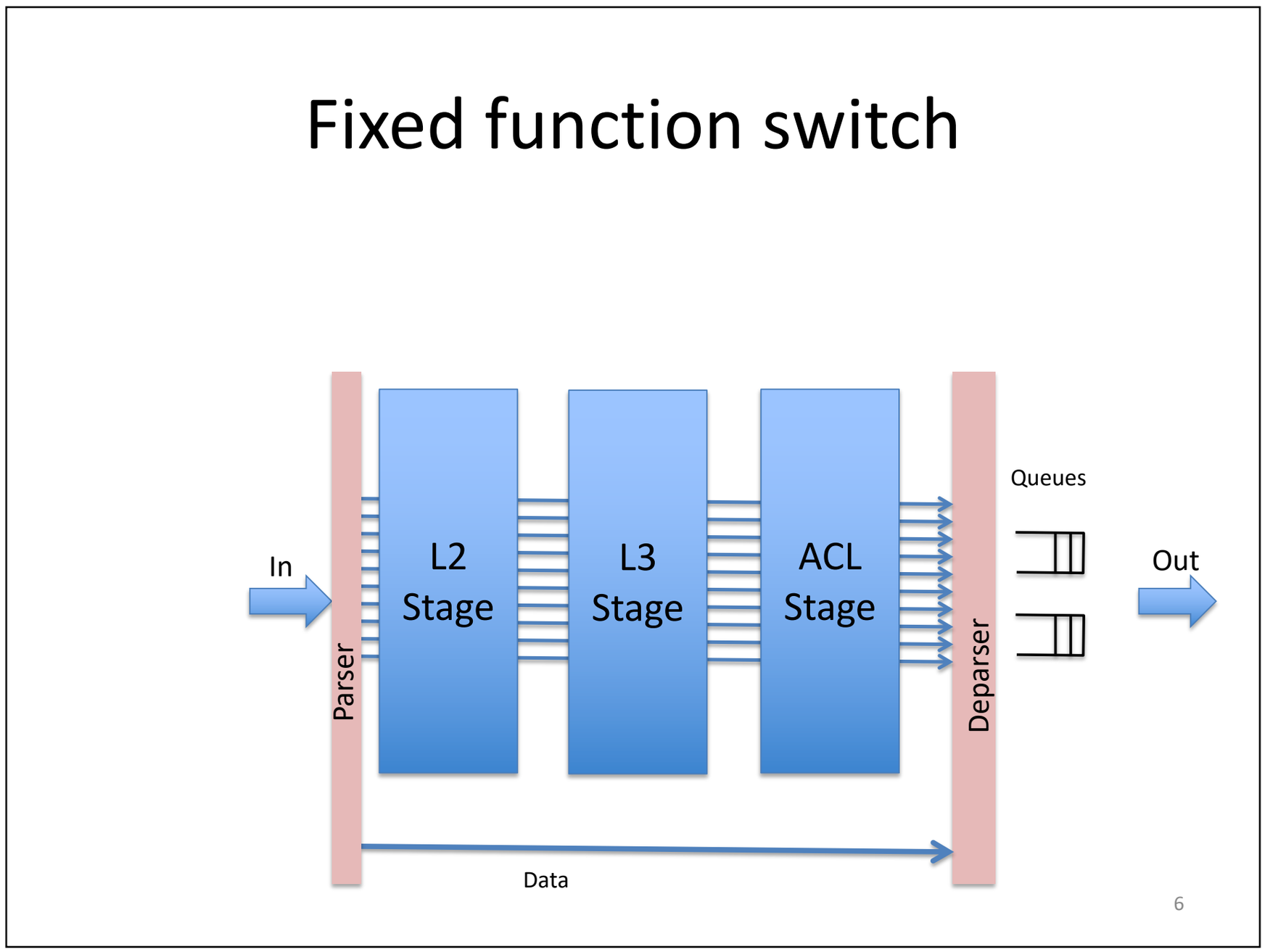
[SIGCOMM'13]

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

Outline

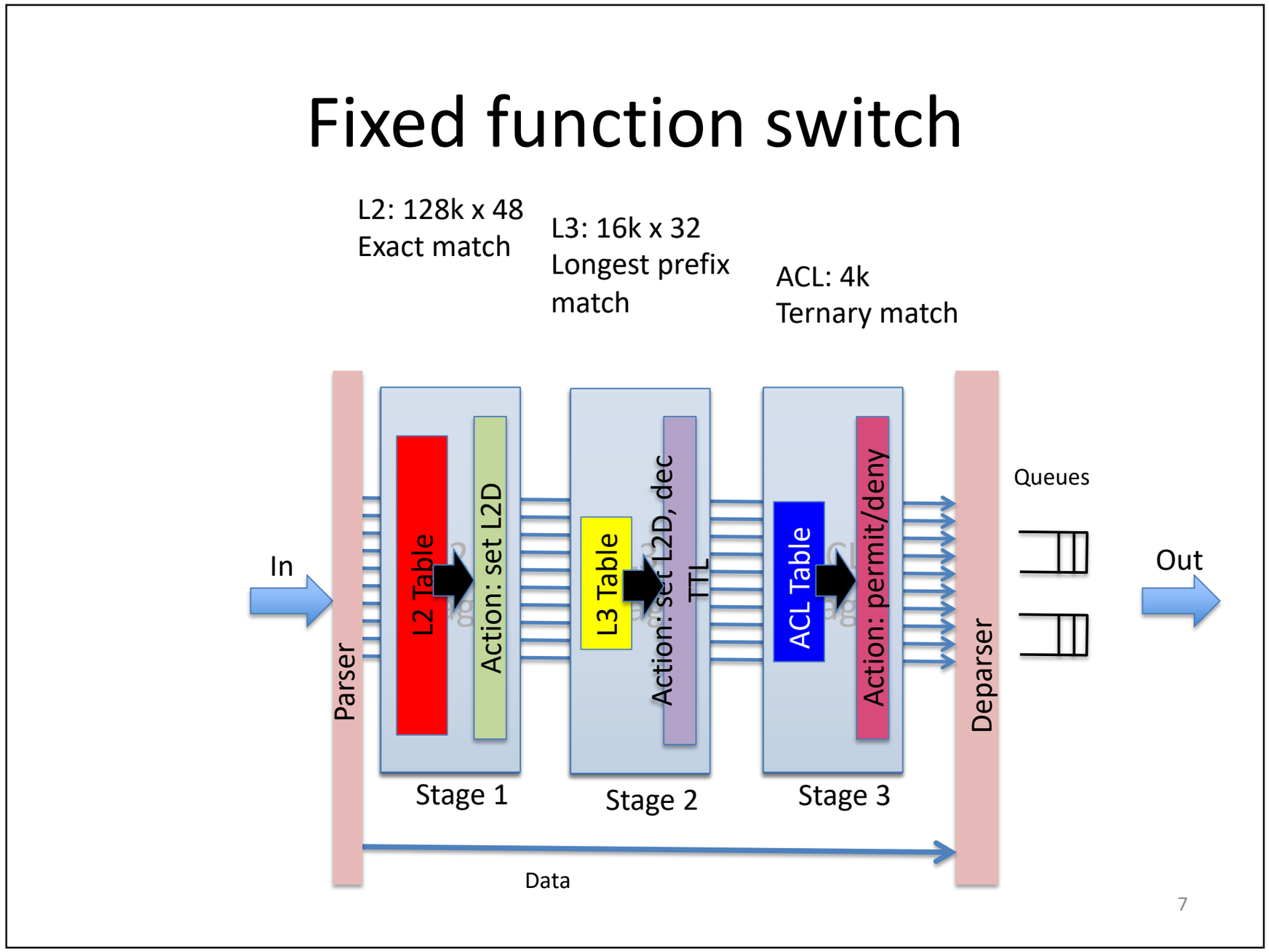
- Conventional switch chips are inflexible
- SDN demands flexibility…sounds expensive…
- How do we do it: The RMT switch model
- Flexibility costs less than 15%

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



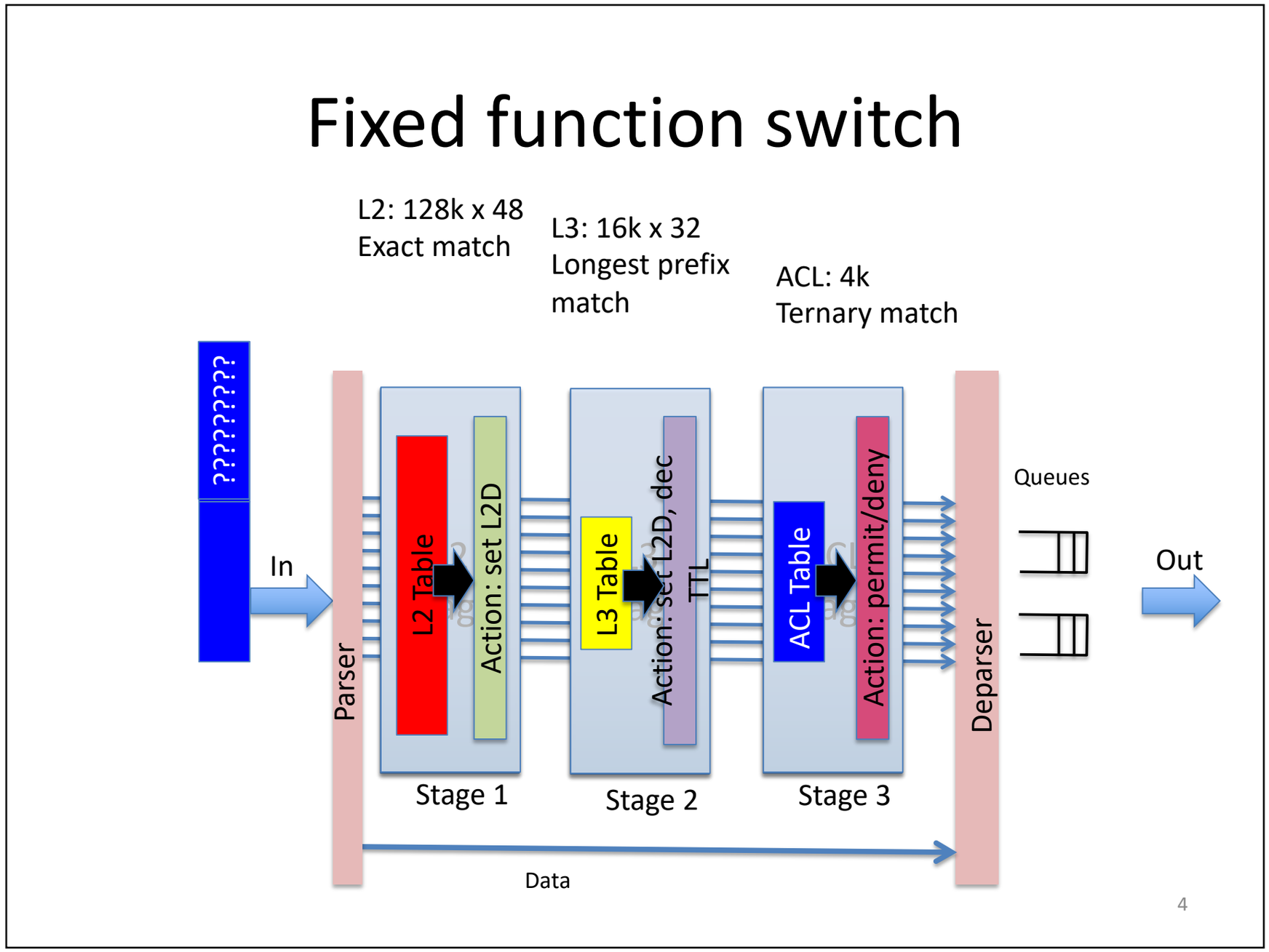
In such a switch,

each stage is particularized to its usage



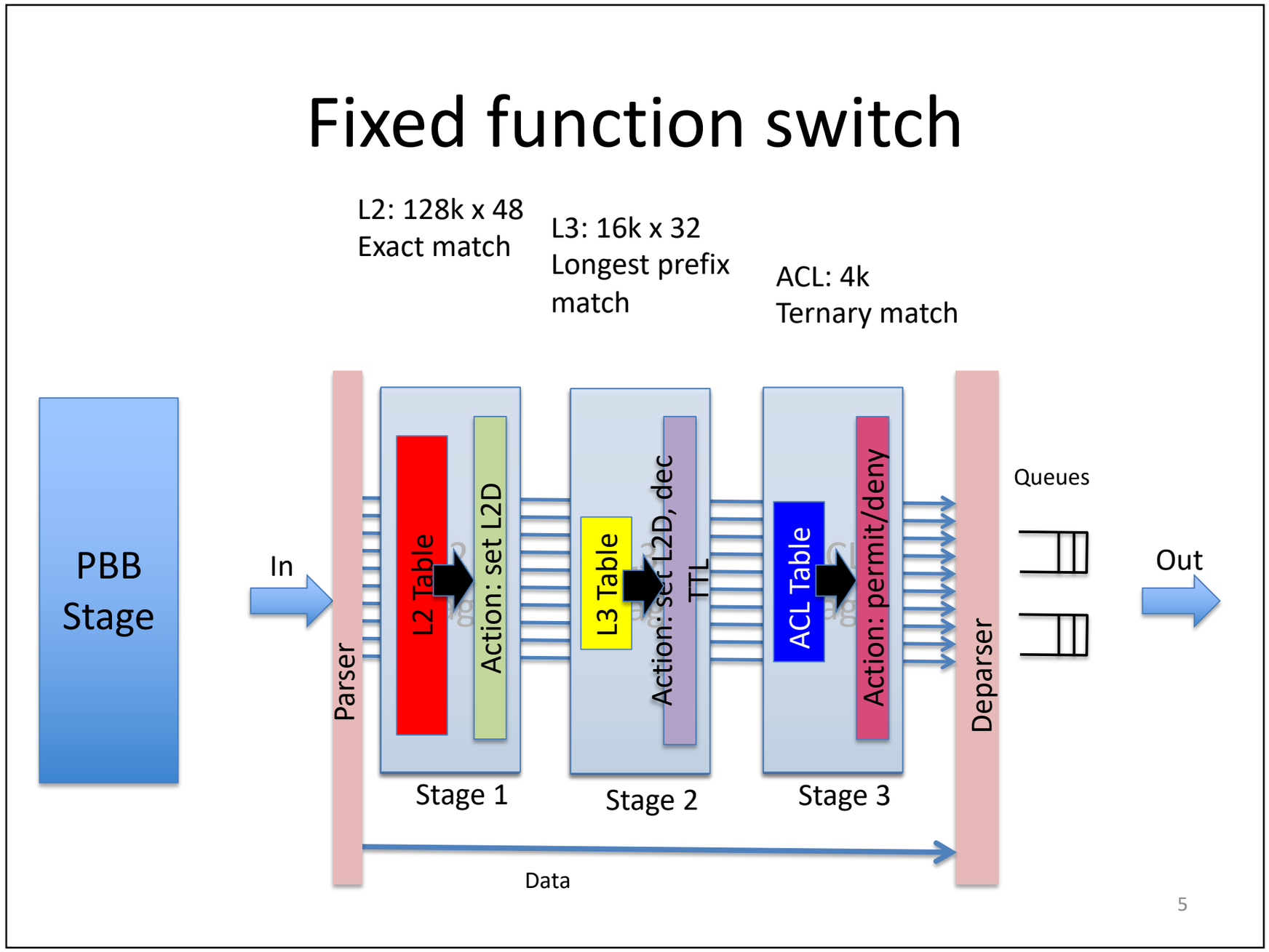
This specificity makes it impossible to…

re-arrange the tables' sizes



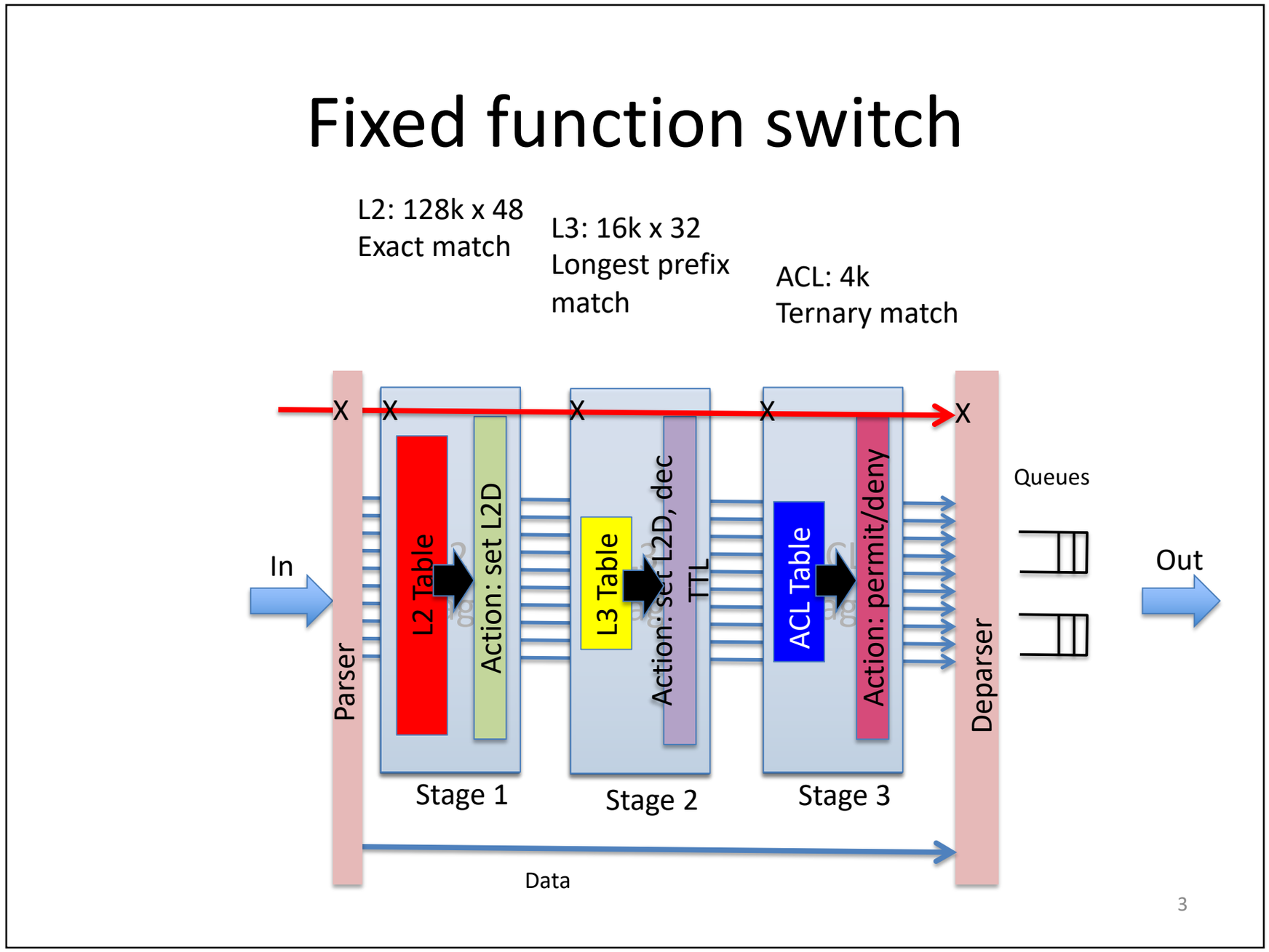
This specificity makes it impossible to…

add a new table



This specificity makes it impossible to…

support new headers or new actions



What We Set Out To Learn

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

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

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%

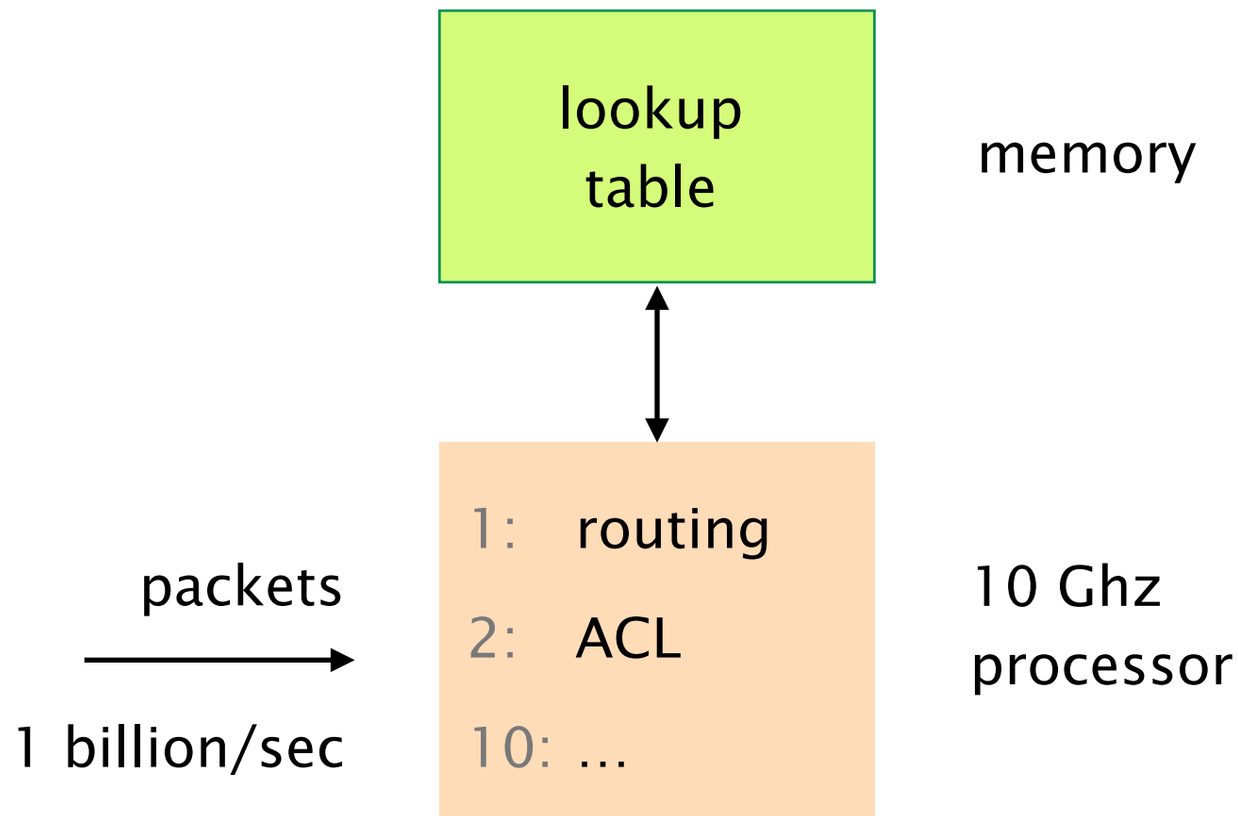
What kind of switch architecture could support flexibility and yet run at Terabits per second?

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

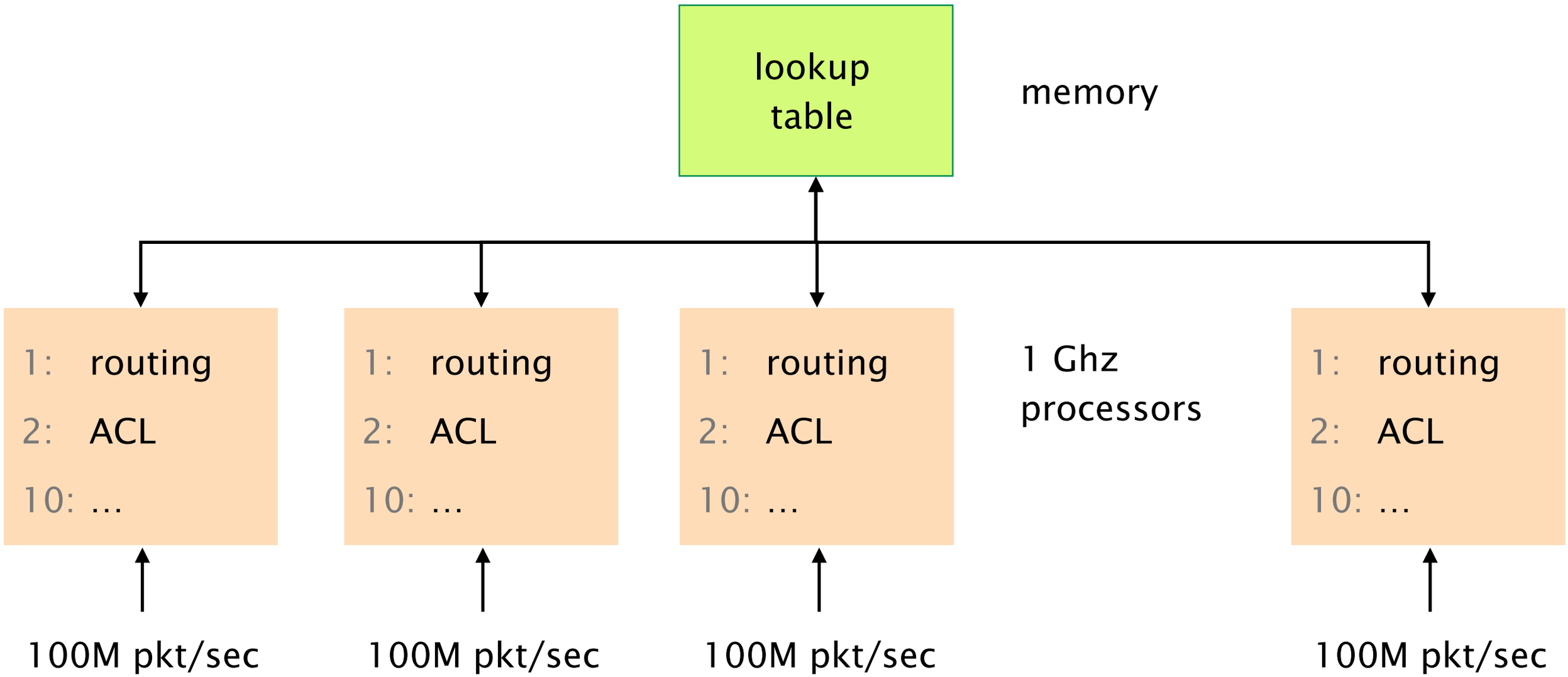


Let's parallelize things with a

packet-parallel architecture

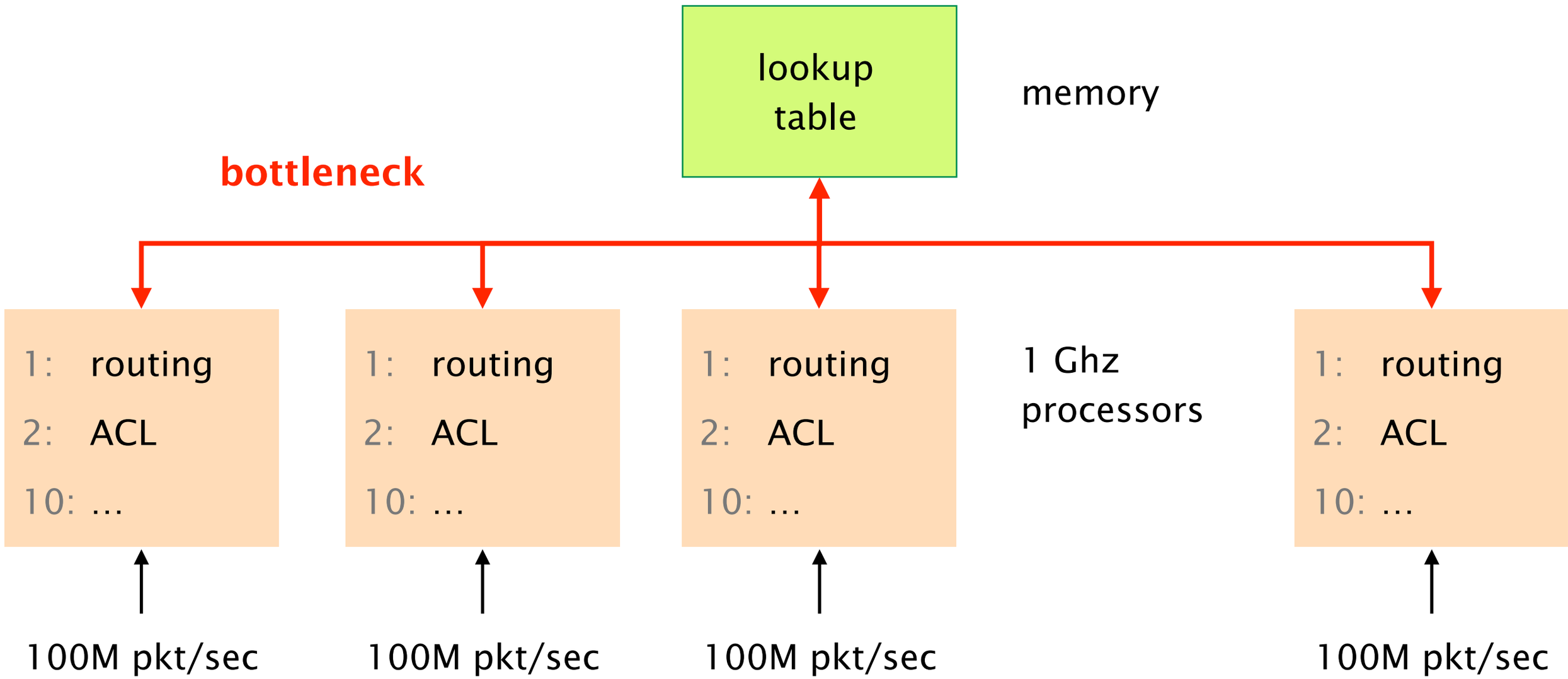
What about we duplicate the processing units?

Each of which clocked at 1 Ghz



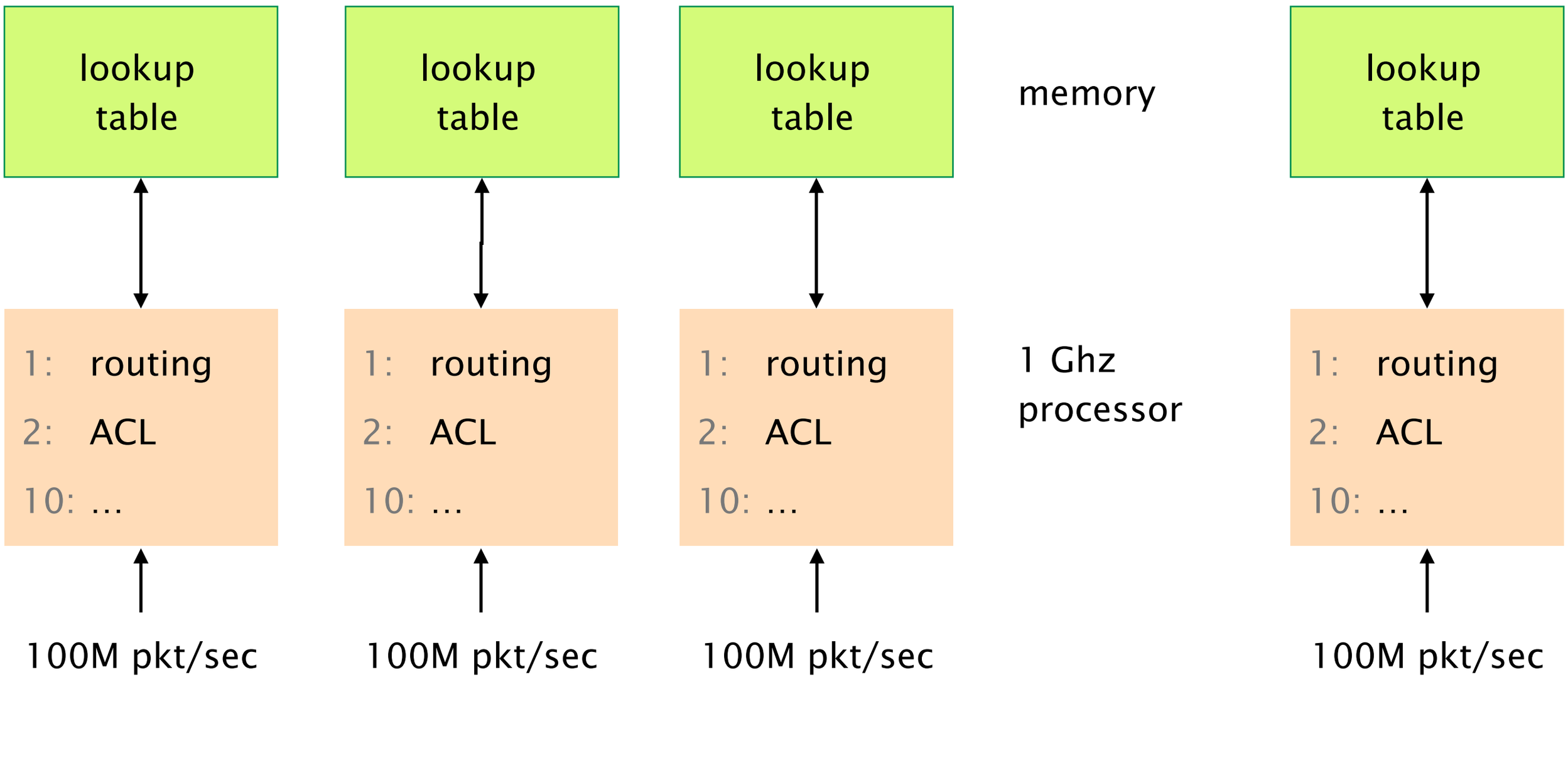
The next issue is to scale the

the memory-to-CPU bandwidth



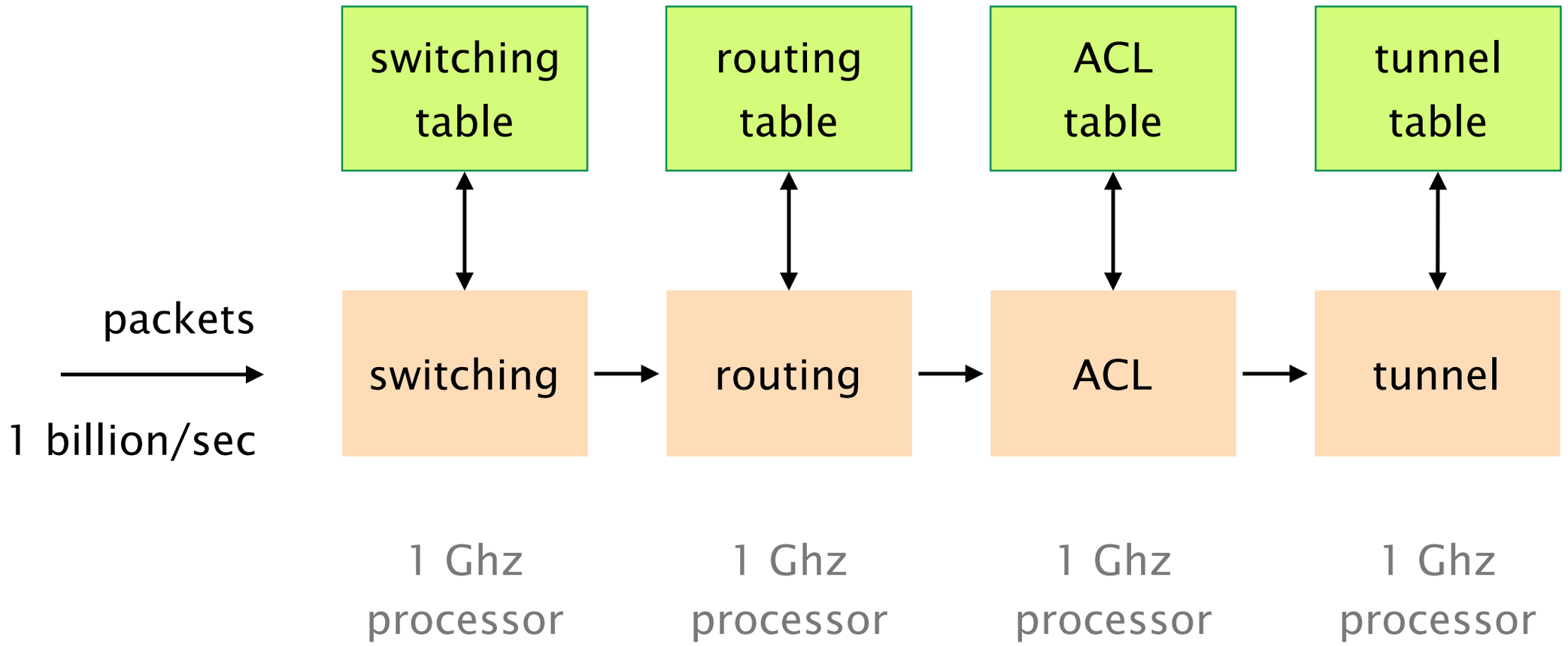
We could replicate the memory of course…

but that comes at a huge cost in die area



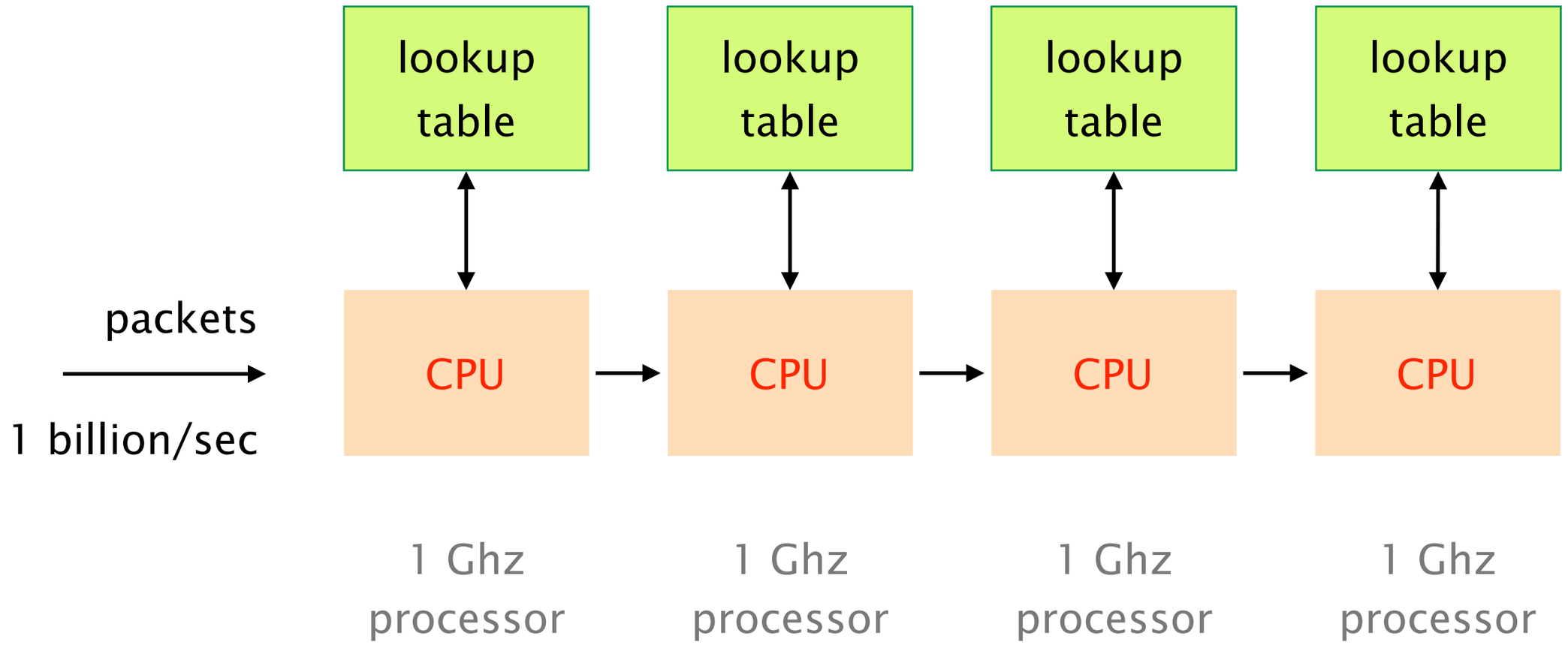
What if we organize the processing as a pipeline instead?

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

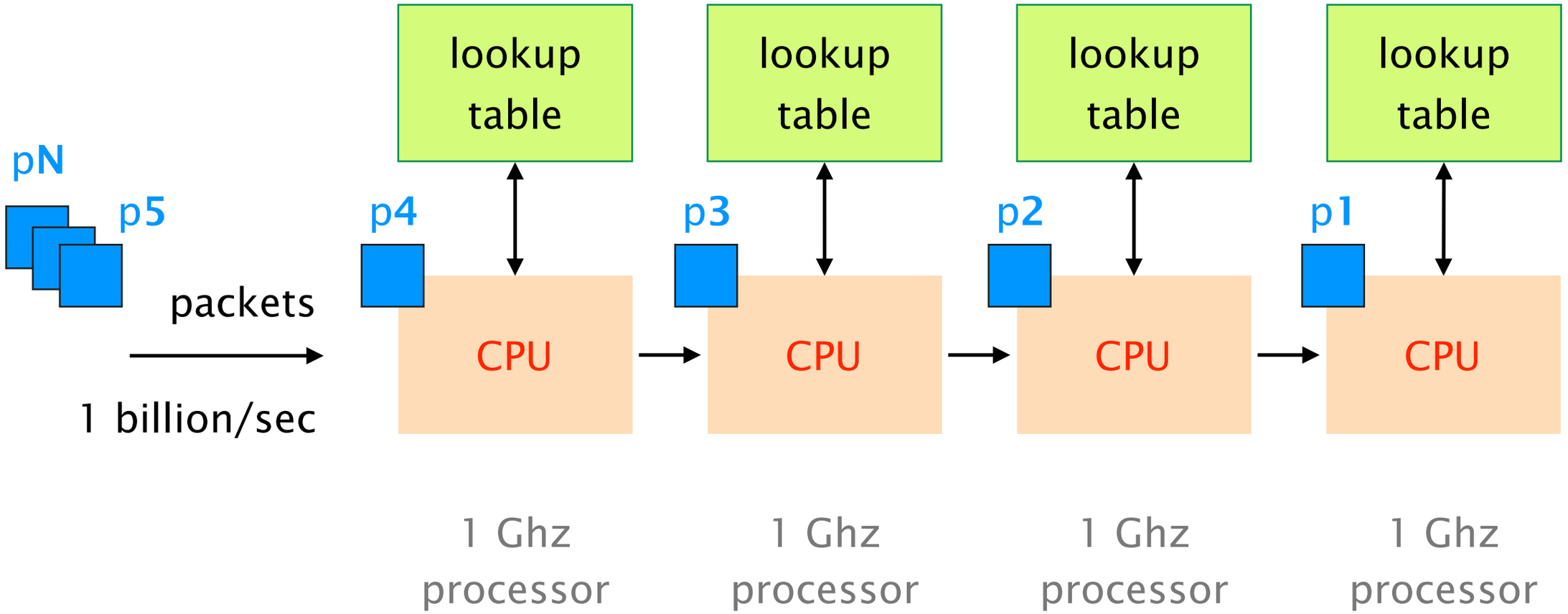


For flexibility,

each processing unit/memory can be made generic



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



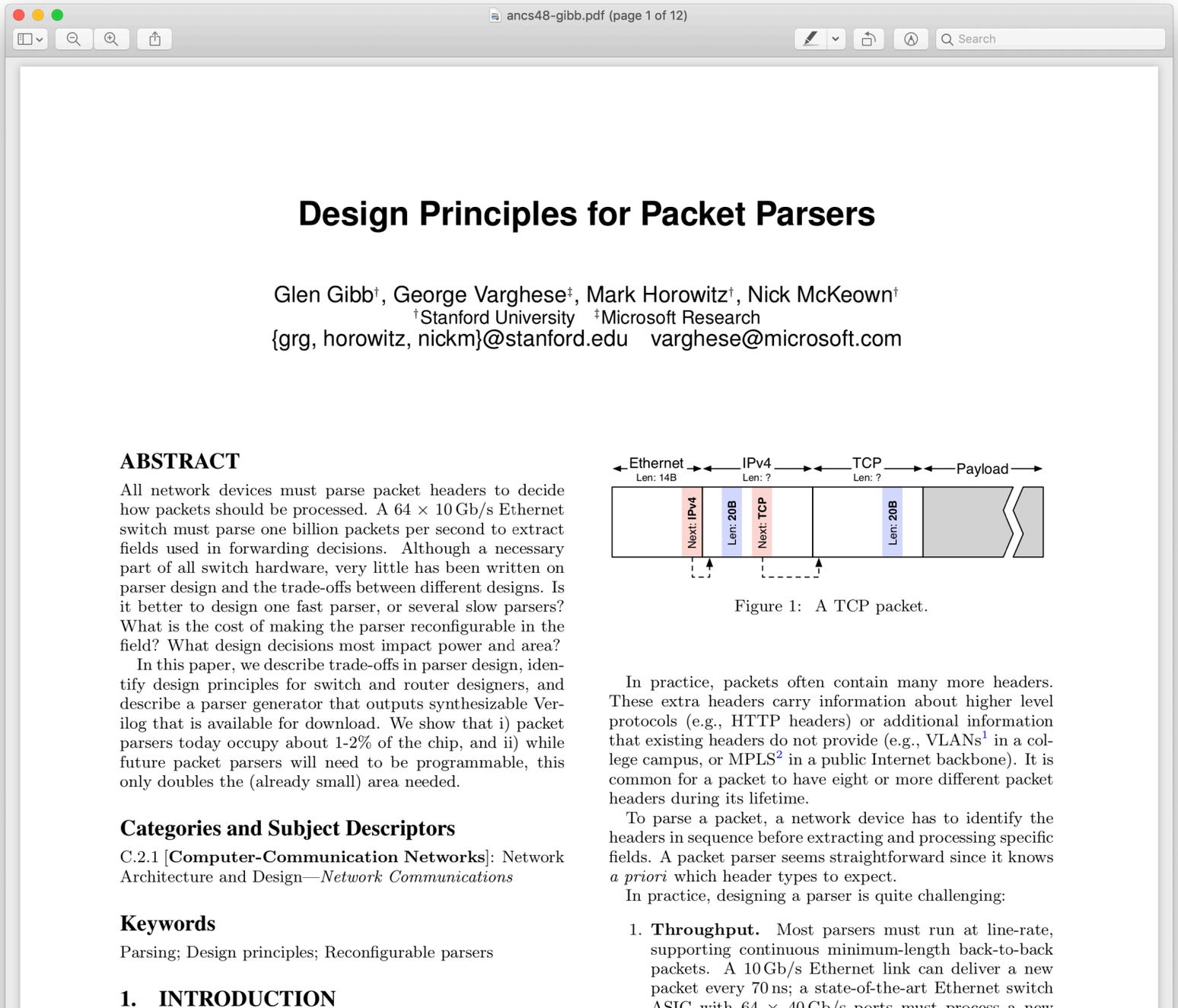
The runtime behavior of the parser & the match stages is defined through the RMT abstract model

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?



ancs48-gibb.pdf (page 1 of 12)

Search

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

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.

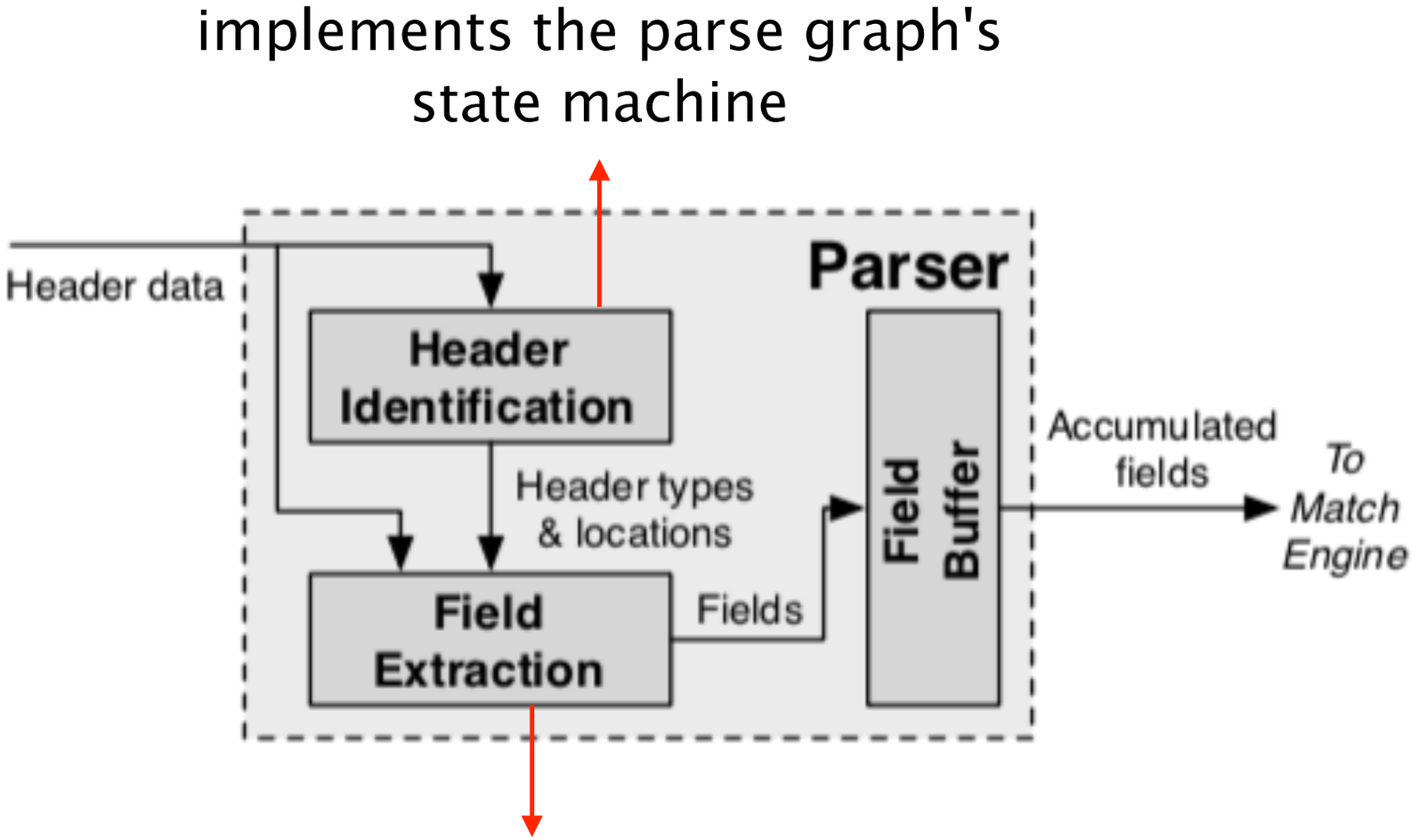
[ANCS'13]

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:

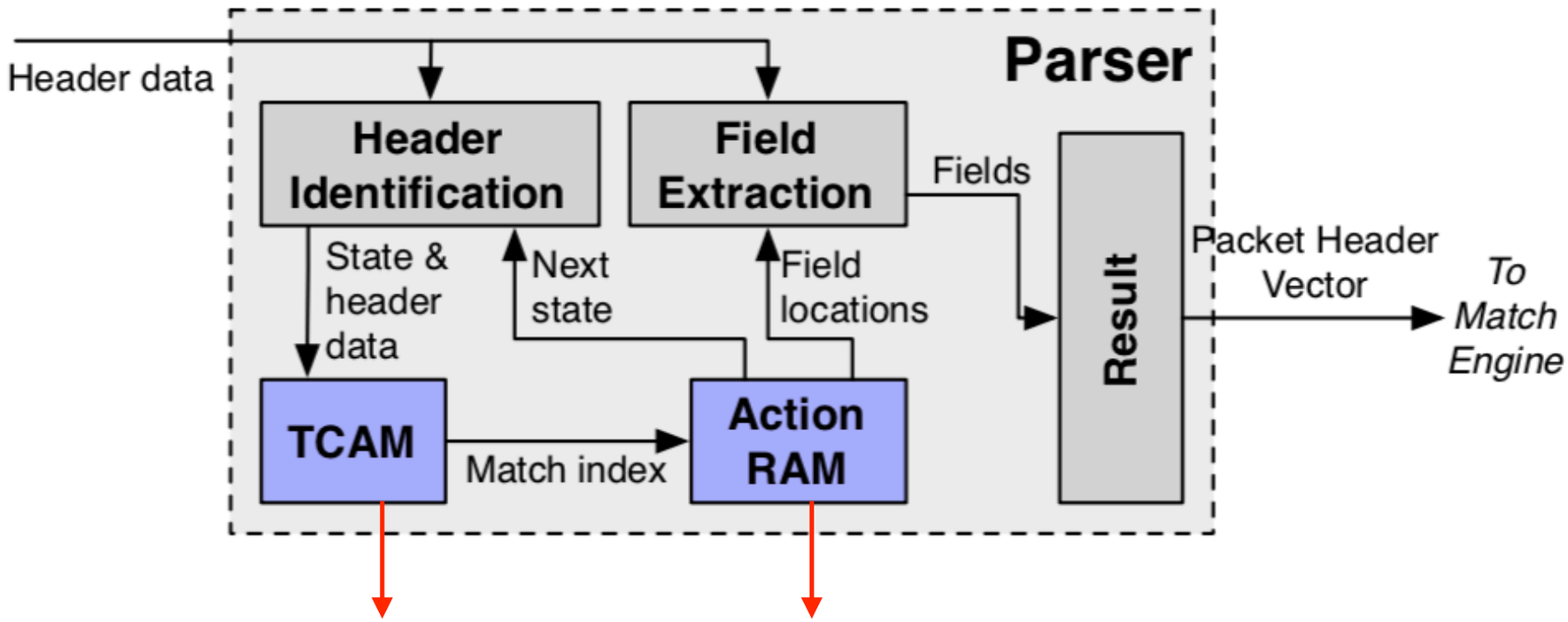
header identification and field extraction



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 SRAM and/or TCAM

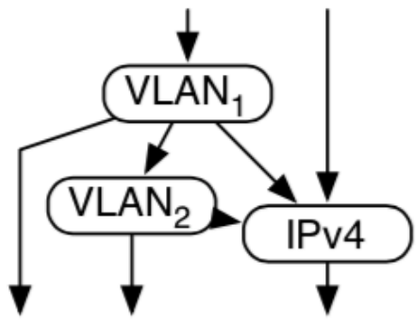


stores the bit sequences
that identify the headers

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

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

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



(a) Graph fragment.

TCAM

Curr. Hdr.	Lookup Val.
$VLAN_1$	0x0800
$VLAN_1$	0x8100
$VLAN_2$	0x0800

SRAM

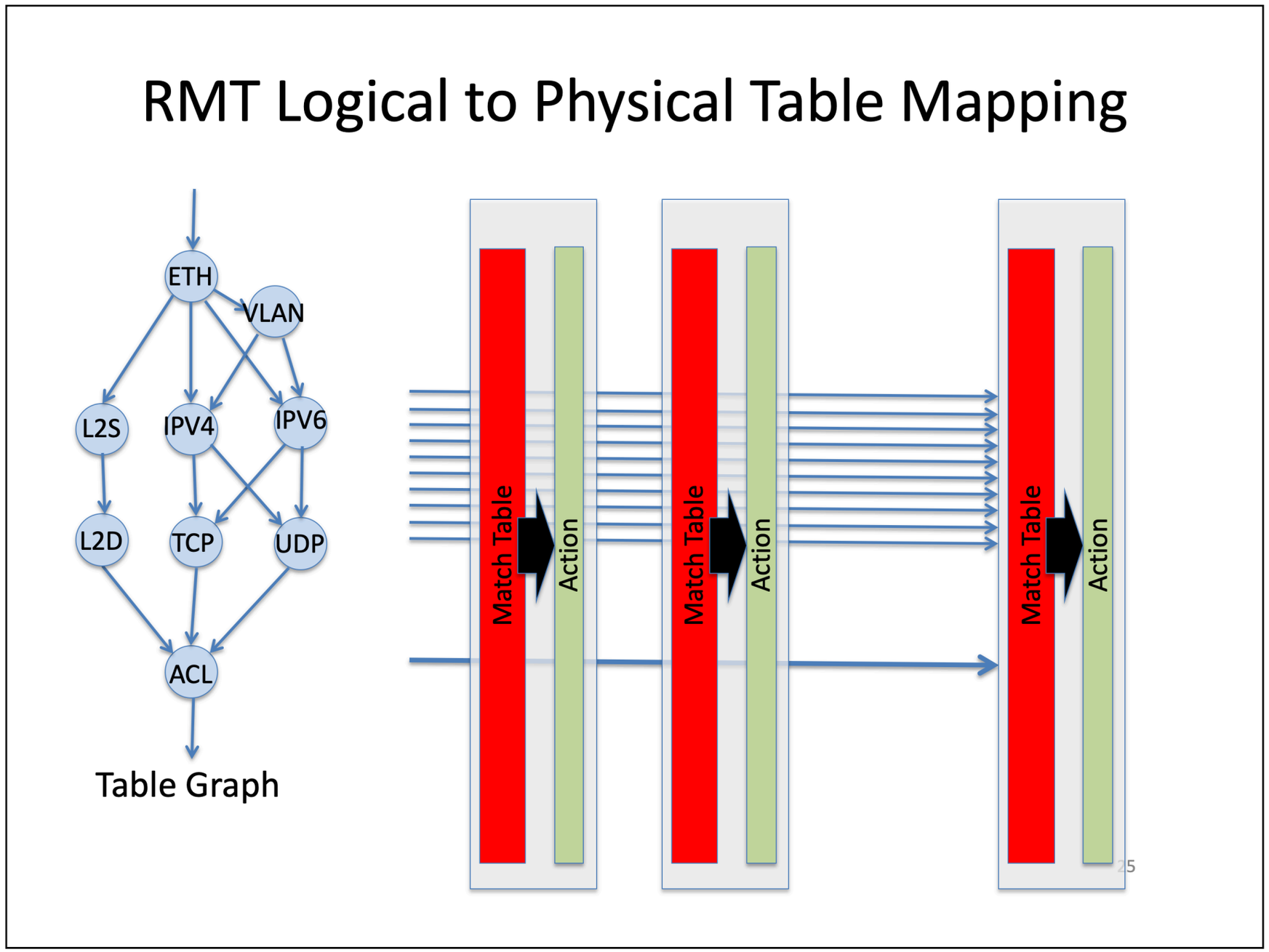
Next Hdr.	Hdr. Len	Next lookup offset
IPv4	4	0
$VLAN_2$	4	2
IPv4	4	0

(b) Corresponding parse table entries.

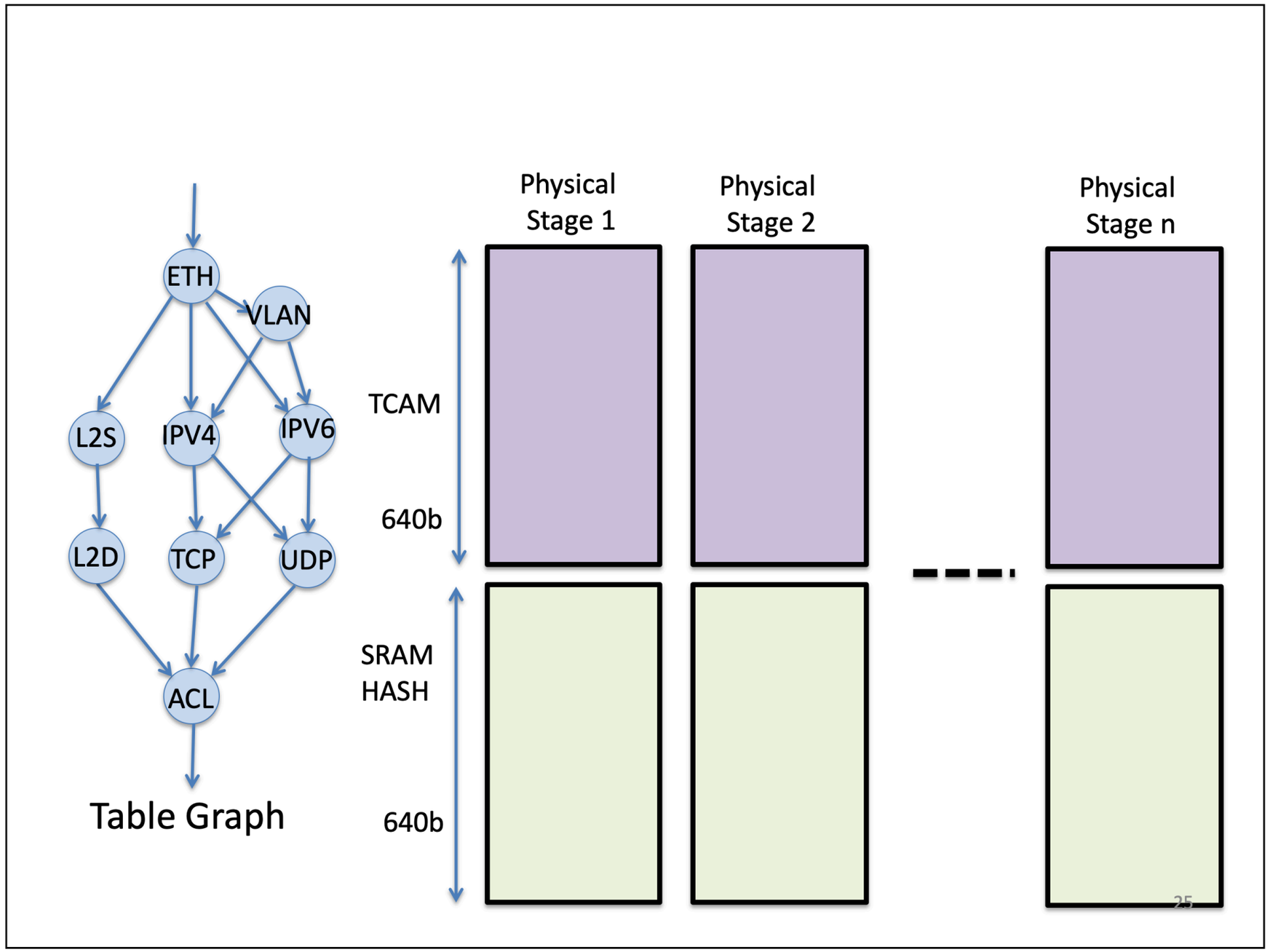
Source: Design Principles for Packet Parsers, Gibb et al.

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

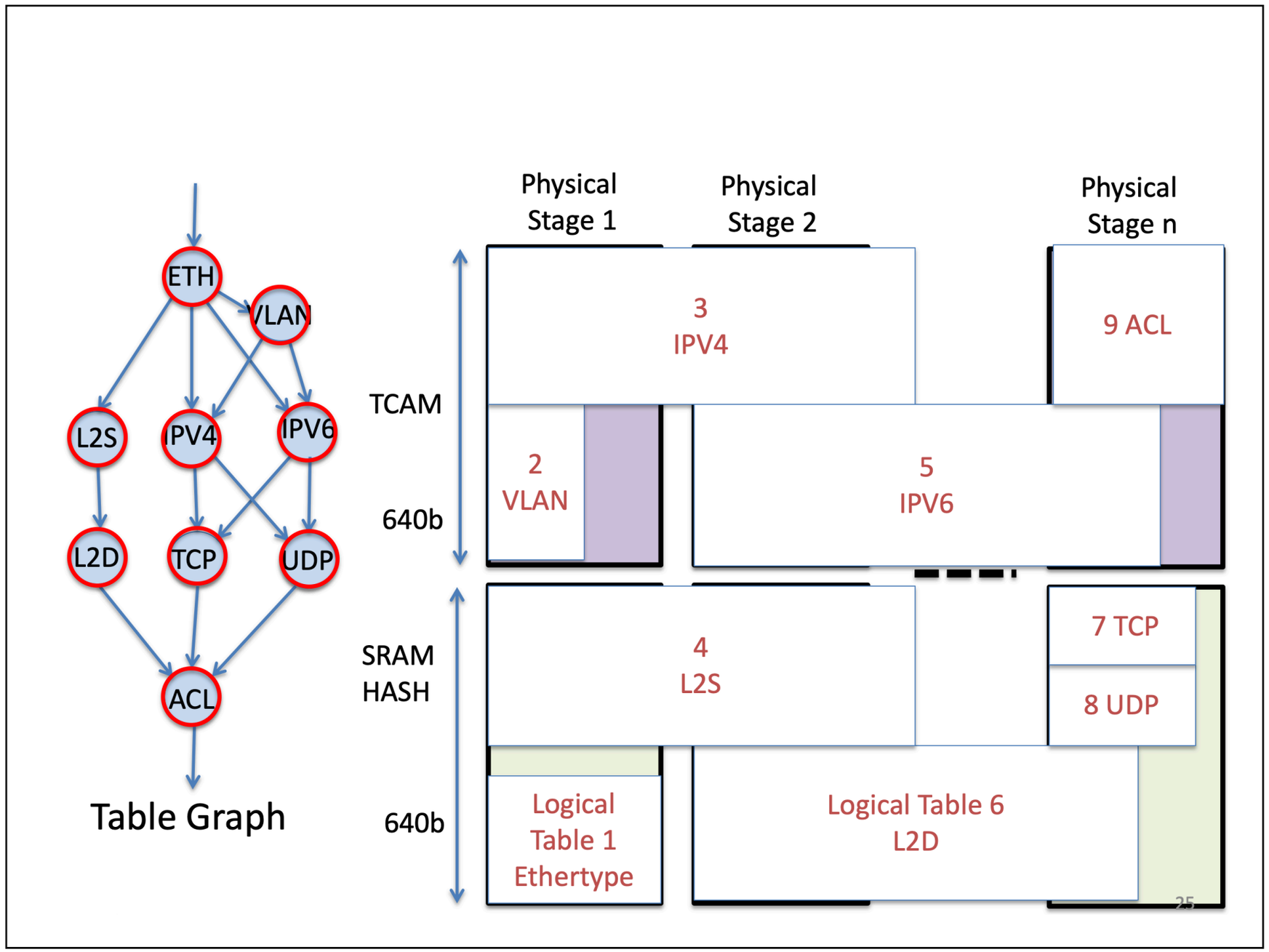
A compiler translates a given RMT logical pipeline (specified in P4) into a physical one



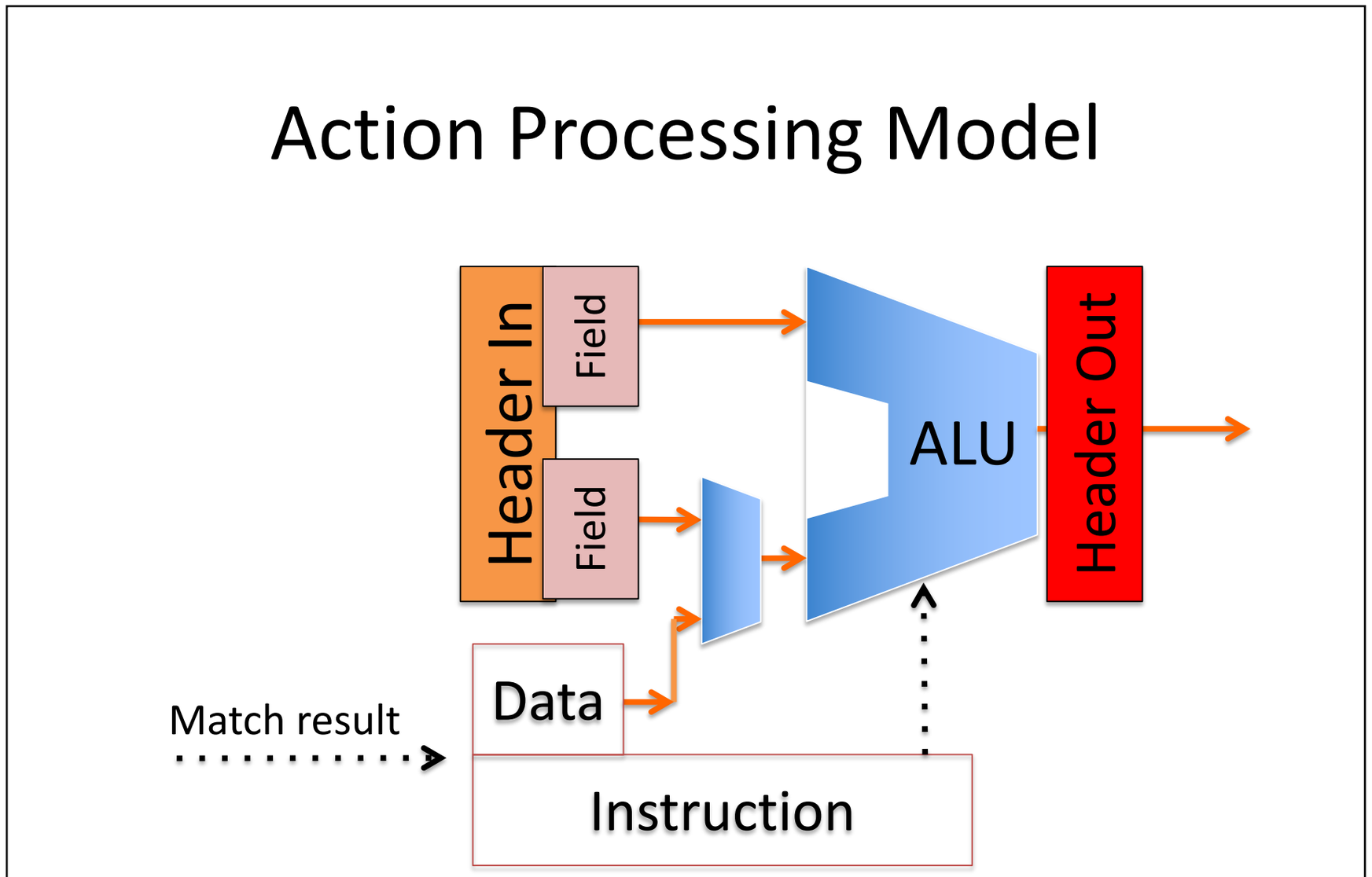
Each physical stage contains dedicated SRAM, for exact matches, and TCAM, for ternary matches



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



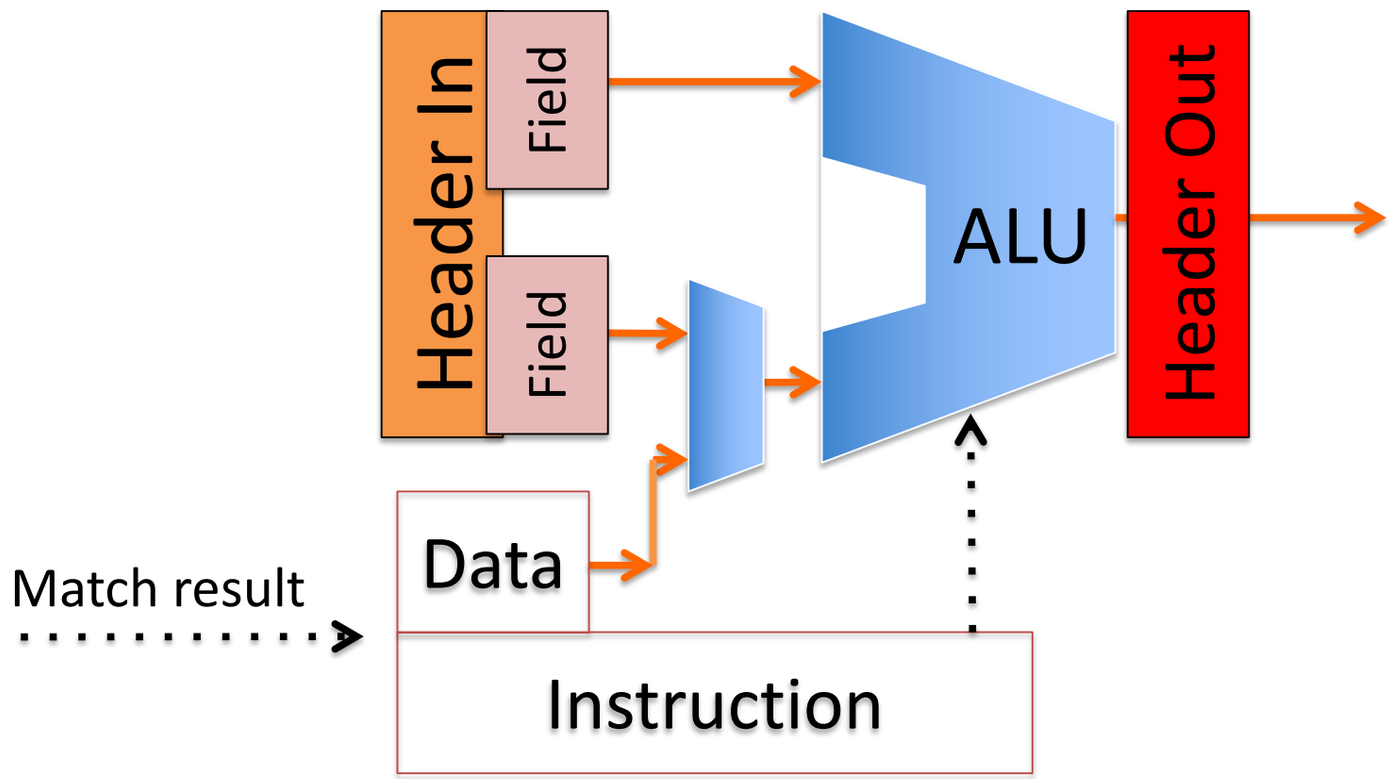
The RMT pipeline relies on many Arithmetic Logic Units (ALU) to perform actions on the result of a match



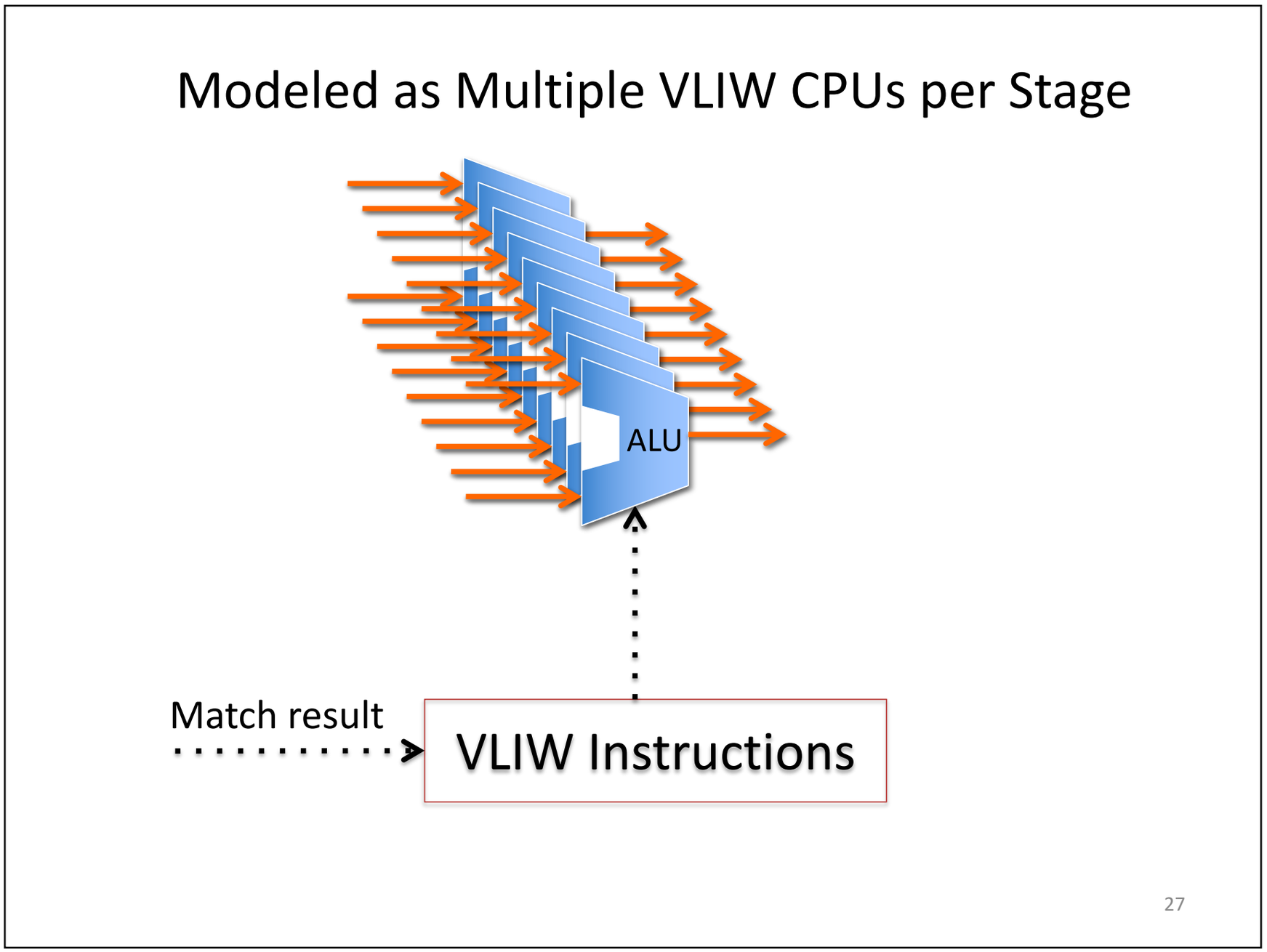
Each ALU modifies only one word of a header

(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)



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%

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%
Total extra area cost		**14.2%**

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%
Total extra area cost		**14.2%**

Power

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

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

P4 hardware target

Probabilistic data structures

Can we trade some accuracy for memory?

← You are looking at a stream of data.

In networking, we usually talk about **streams of packets**,

but these questions apply to other domains as well,

e.g. **search engines and databases**.

← You are looking at a stream of data (packets).

There are many questions you might ask:

Is a certain element (e.g. ip address) in the stream?

How frequently does an element appear?

How many distinct elements are in the stream?

What are the most frequent elements?

← You are looking at a stream of data (packets).

There are many questions you might ask:

Is a certain element (e.g. ip address) in the stream?

How frequently does an element appear?

How many distinct elements are in the stream?

What are the most frequent elements?

In P4, these questions are difficult to answer.

← You are looking at a stream of data (packets).

Today, I'll show you how set membership and frequency queries can be realized in P4.

PART 1

Is a certain element (e.g. ip address) in the stream?

→ Bloom filter

PART 2

How frequently does an element appear?

→ CountMin Sketch, Count Sketch, ...

part 1: **set membership queries with Bloom filters**

Is a certain element (e.g. ip address) in the stream?

(slides by Thomas Holterbach)

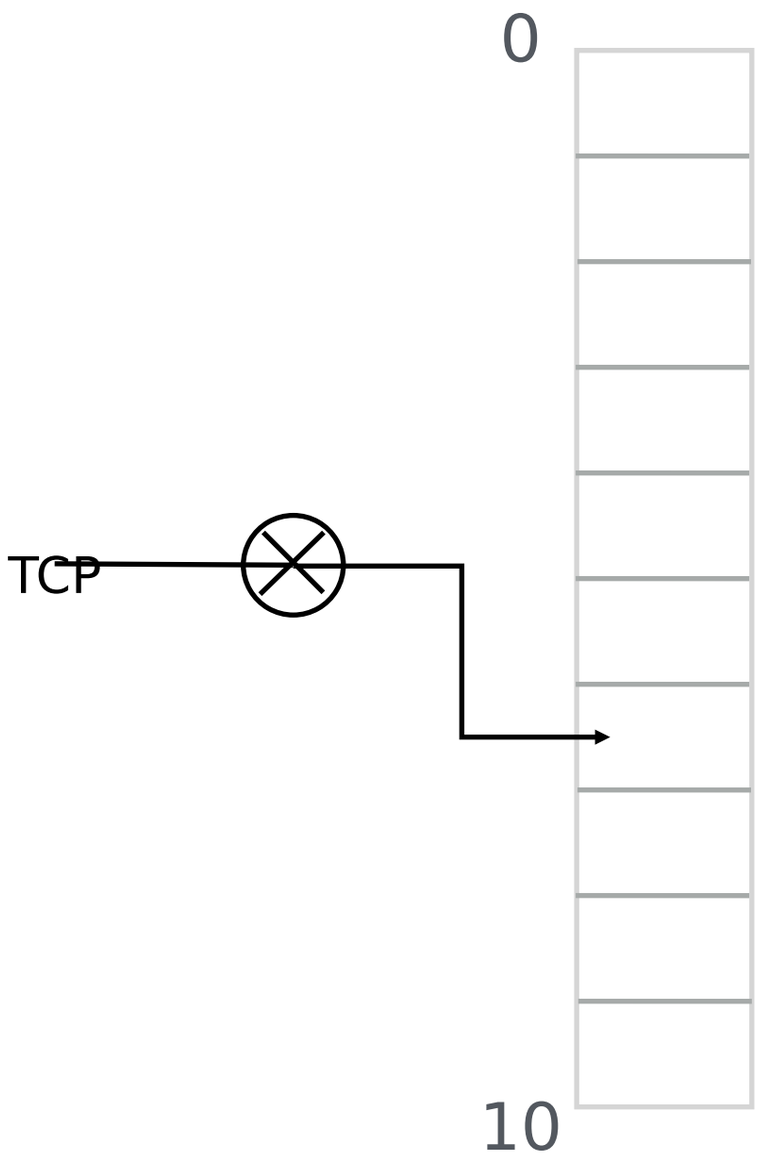
There are two common strategies to implement a set

	strategy #1	strategy #2
output	Deterministic	
number of required operations	Probabilistic	

Intuitive implementation of a set

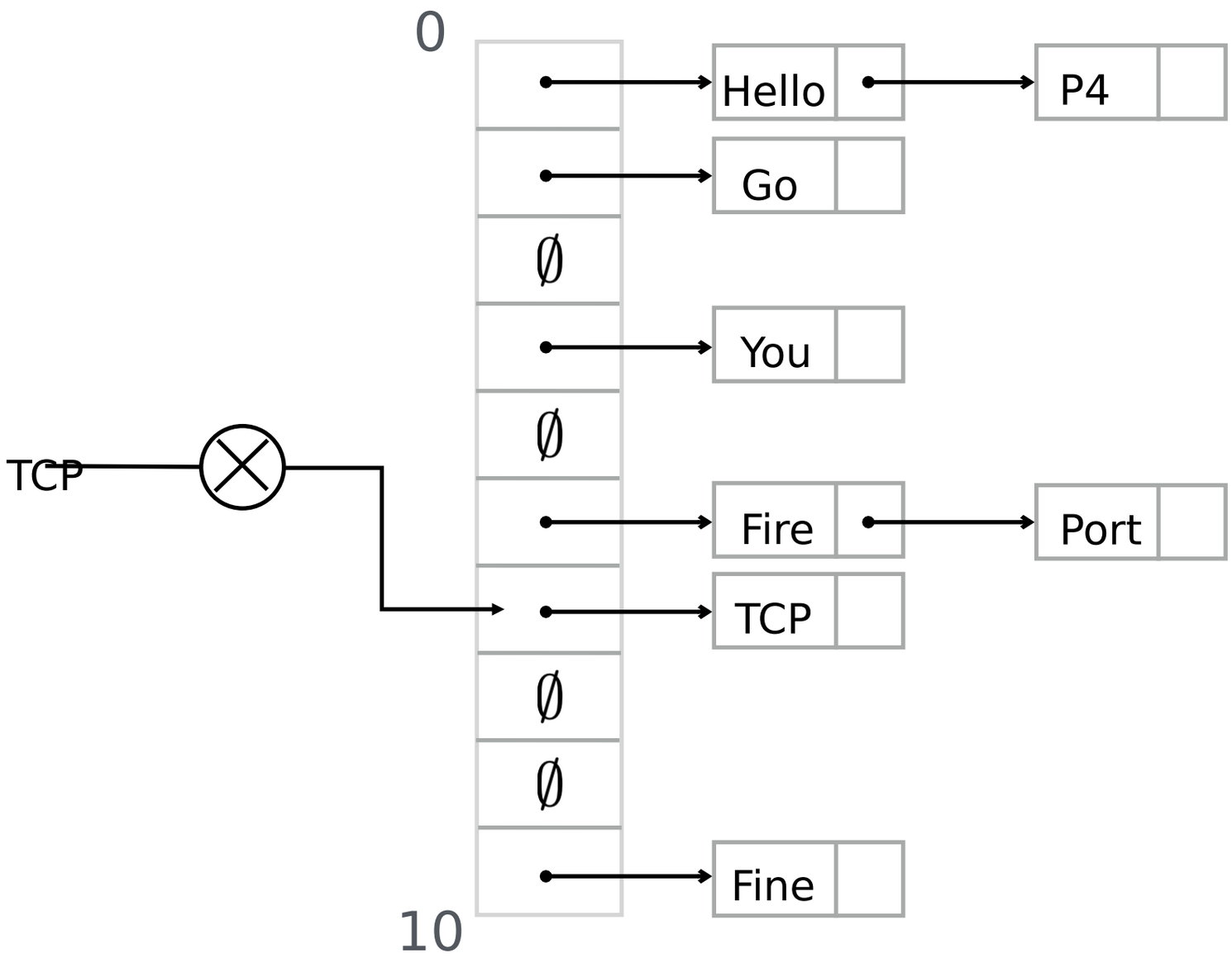
Intuitive implementation of a set

Separate-chaining



Intuitive implementation of a set

Separate-chaining



Intuitive implementation of a set

Separate-chaining

N elements and M cells

	list size
average	N/M
worse-case	N

Intuitive implementation of a set

Separate-chaining

Pros: accurate and fast in the average case

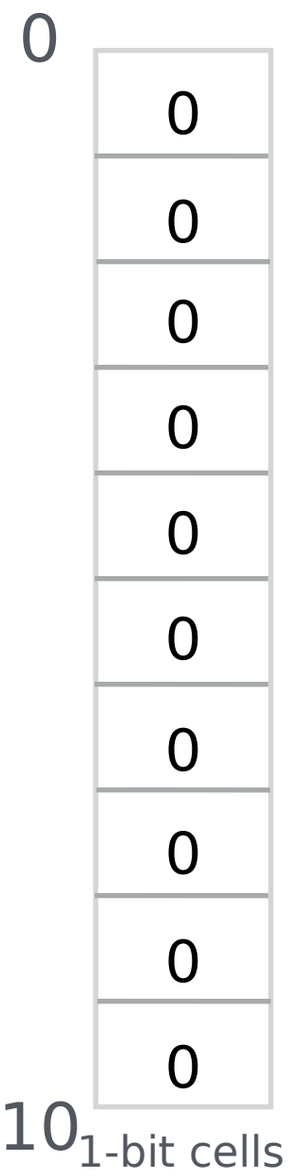
Con: only works in hardware if there is a low number of elements (e.g. < 100)

There are two common strategies to implement a set

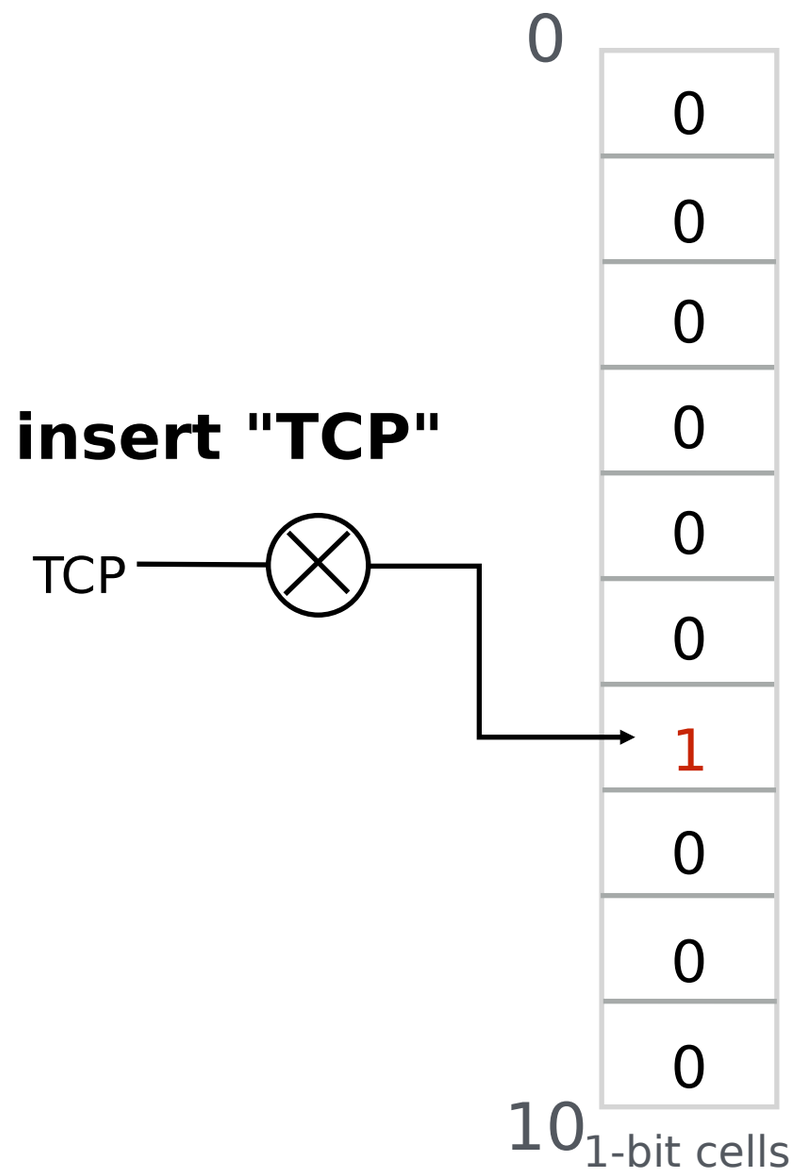
	strategy #1	strategy #2
output	Deterministic	Probabilistic
number of required operations	Probabilistic	Deterministic

‘probabilistic data structures’

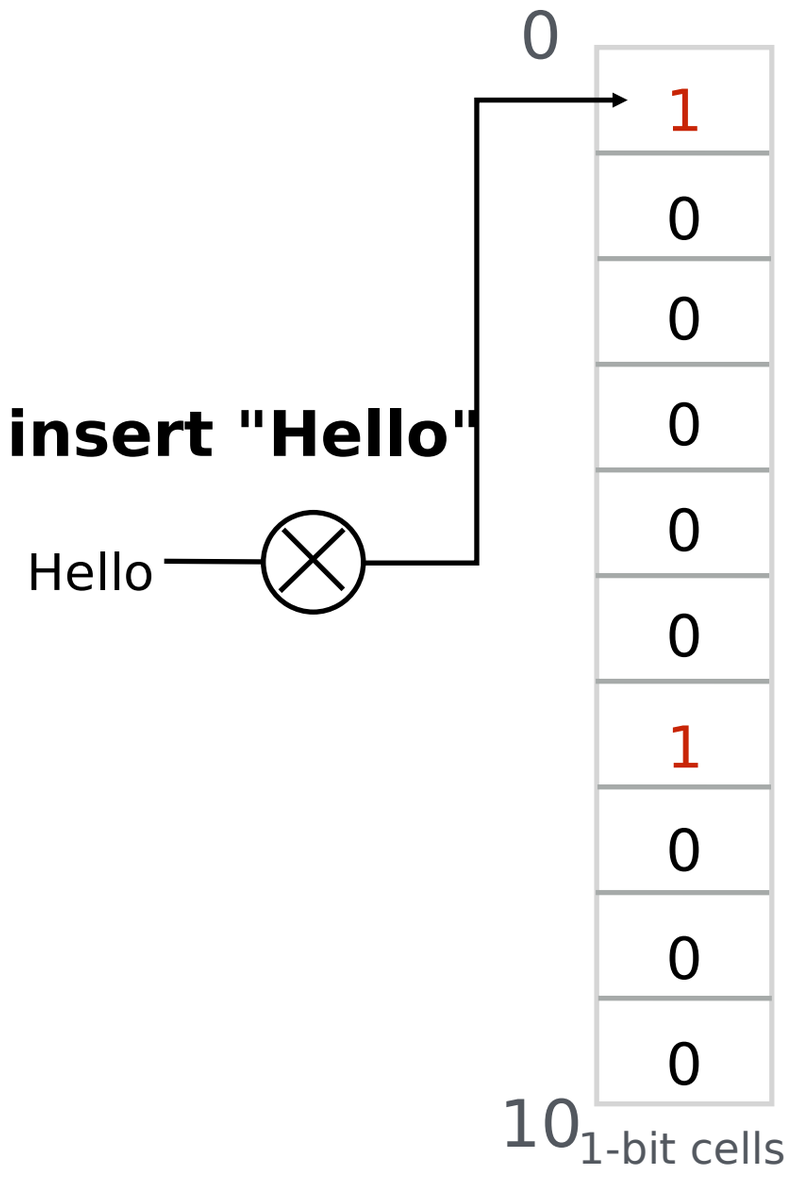
A simple approach for insertions and membership queries



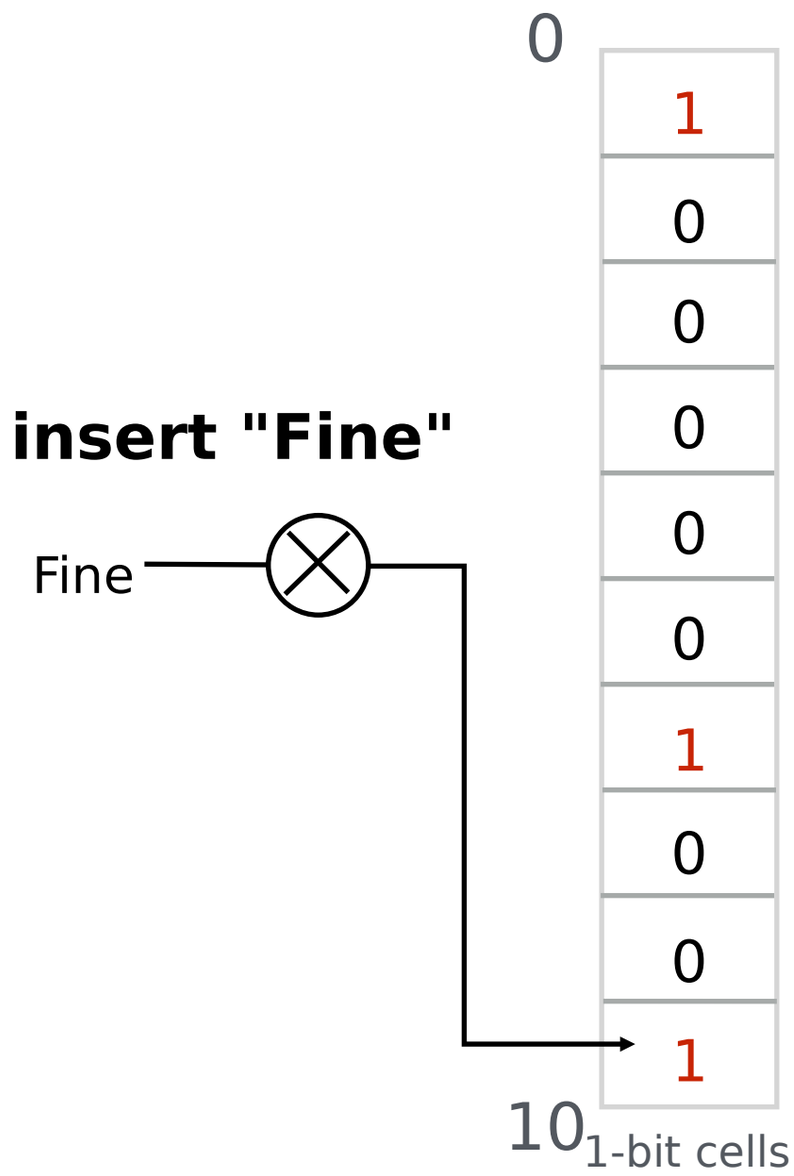
A simple approach for insertions and membership queries



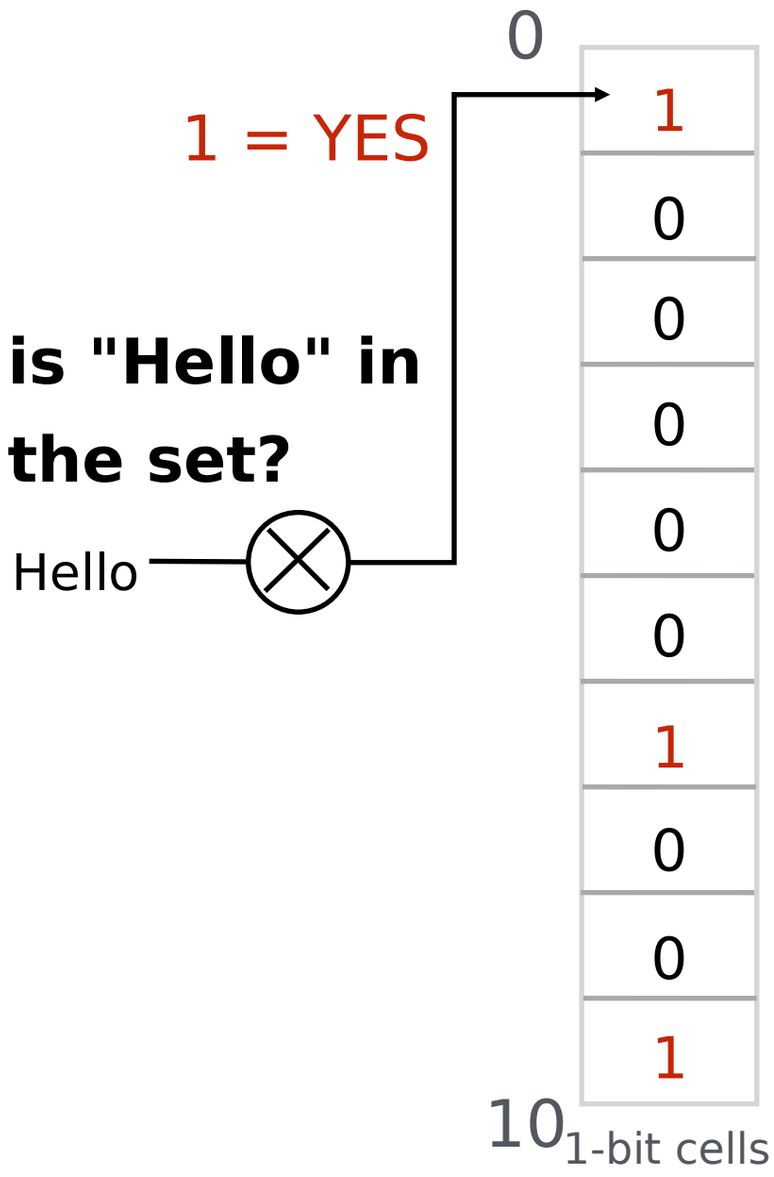
A simple approach for insertions and membership queries



A simple approach for insertions and membership queries

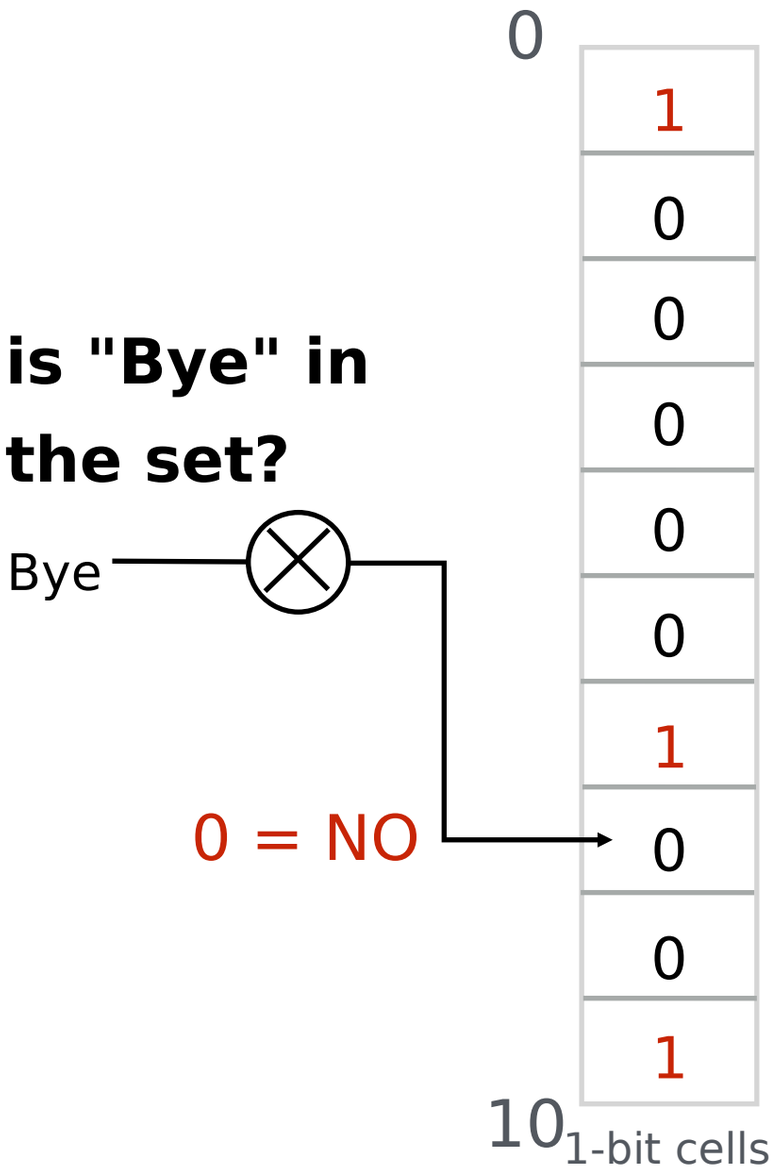


A simple approach for insertions and membership queries

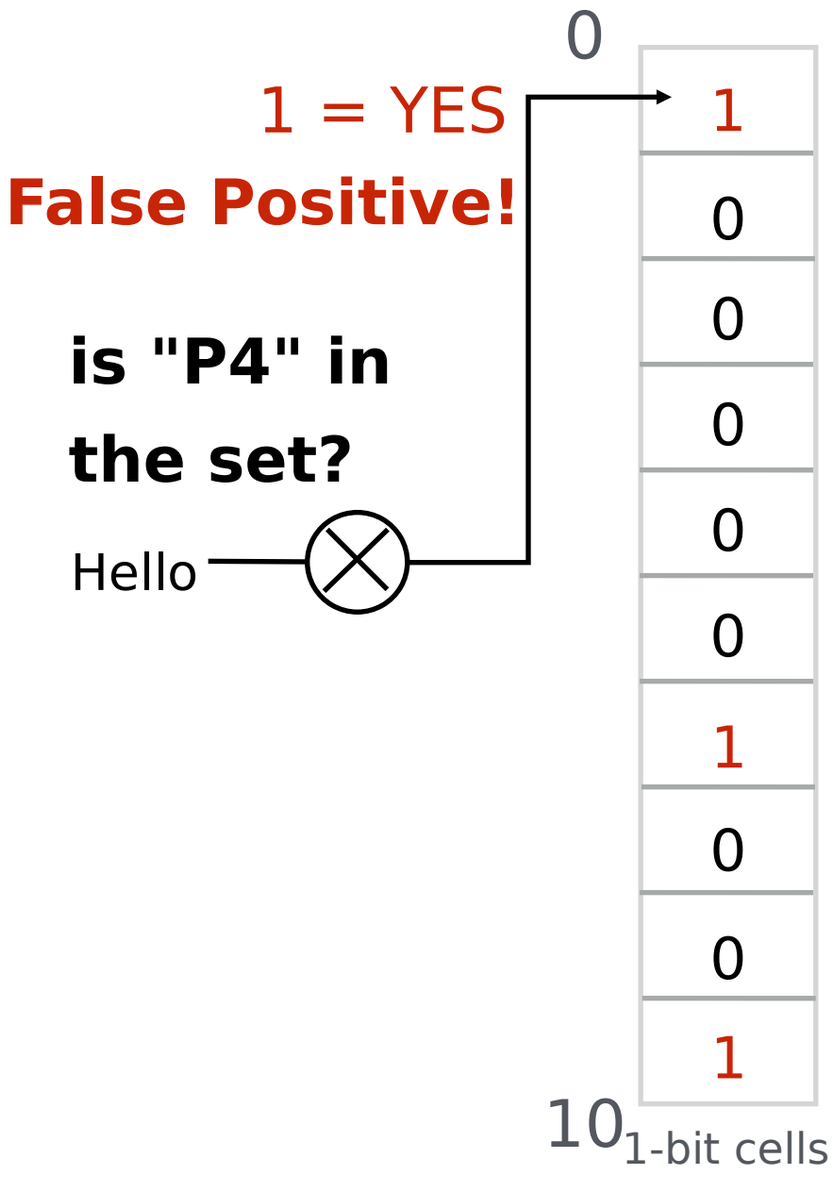


A simple approach for insertions and membership queries

is "Bye" in the set?



A simple approach for insertions and membership queries



A simple approach for insertions and membership queries

N elements and M cells

probability of an element to be mapped into a particular cell	$\frac{1}{M}$
probability of an element not to be mapped into a particular cell	$1 - \frac{1}{M}$
probability of a cell to be 0	$(1 - \frac{1}{M})^N$
false positive rate (FPR)	$1 - (1 - \frac{1}{M})^N$
false negative rate	0

A simple approach for insertions and membership queries

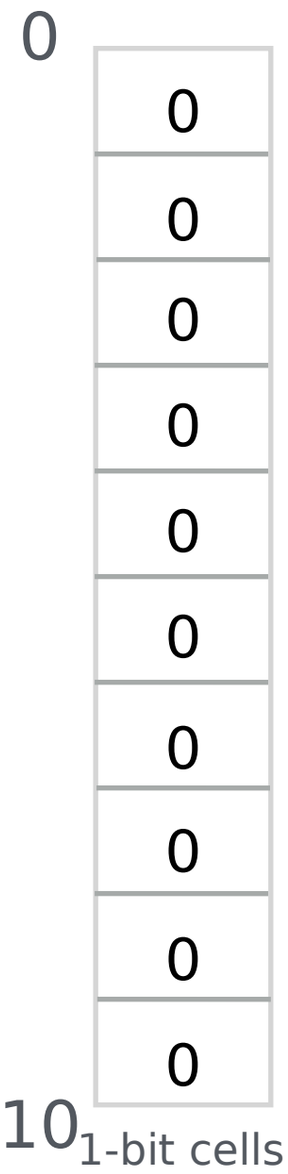
# of elements	# of cells	FPR
1000	10000	9.5%
1000	100000	1%

A simple approach for insertions and membership queries

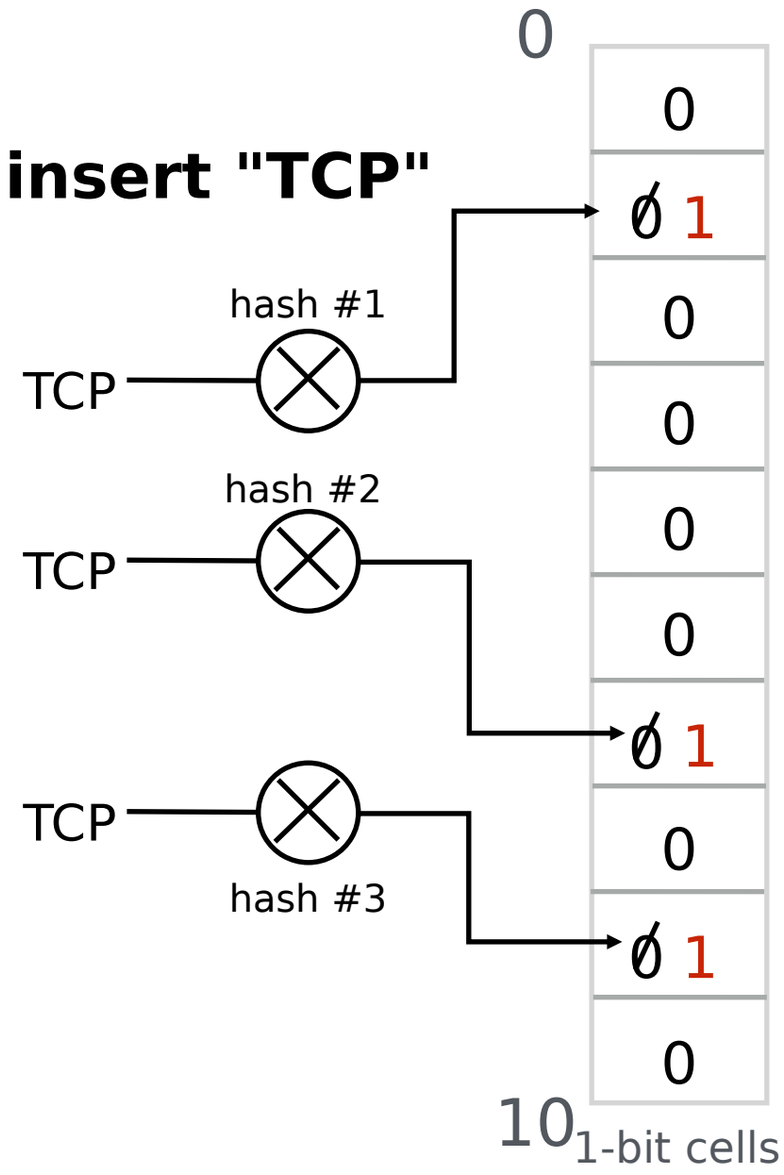
Pros: simple and only one operation per insertion or query

Con: roughly 100x more cells are required than the number of element we want to store for a 1% false positive rate

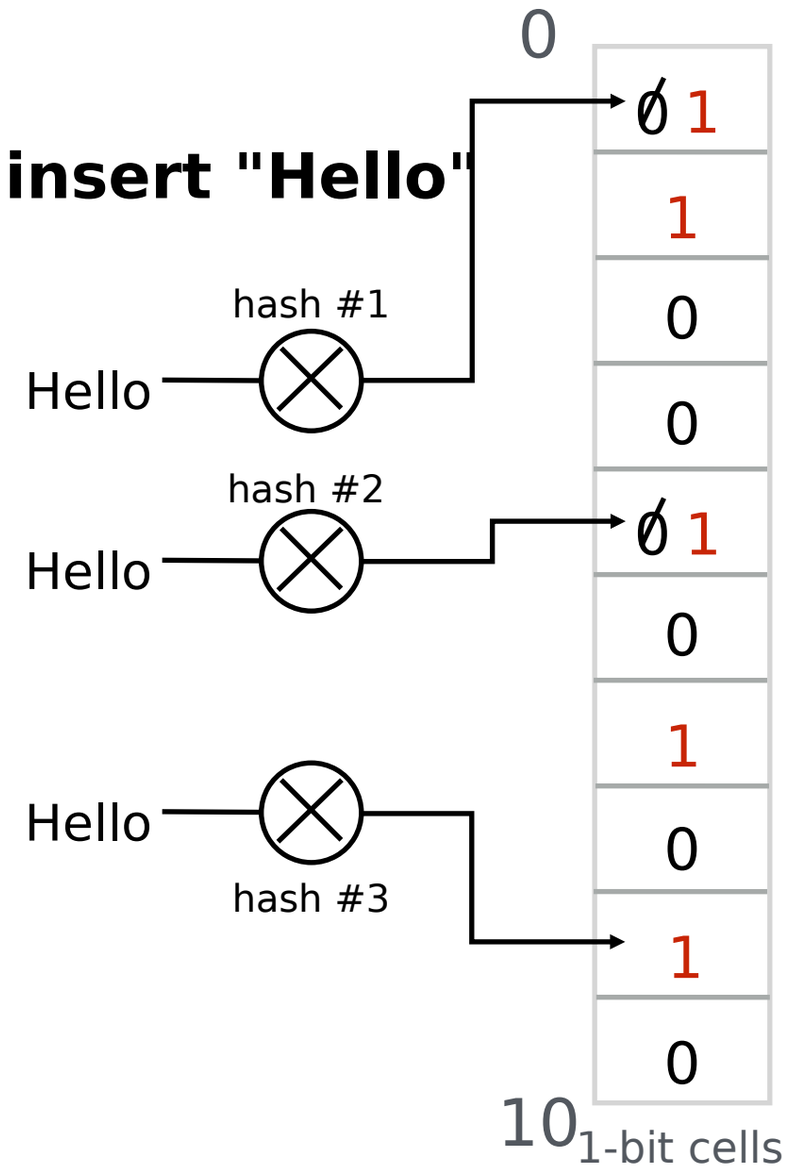
Bloom Filters: a more memory-efficient approach for insertions and membership queries



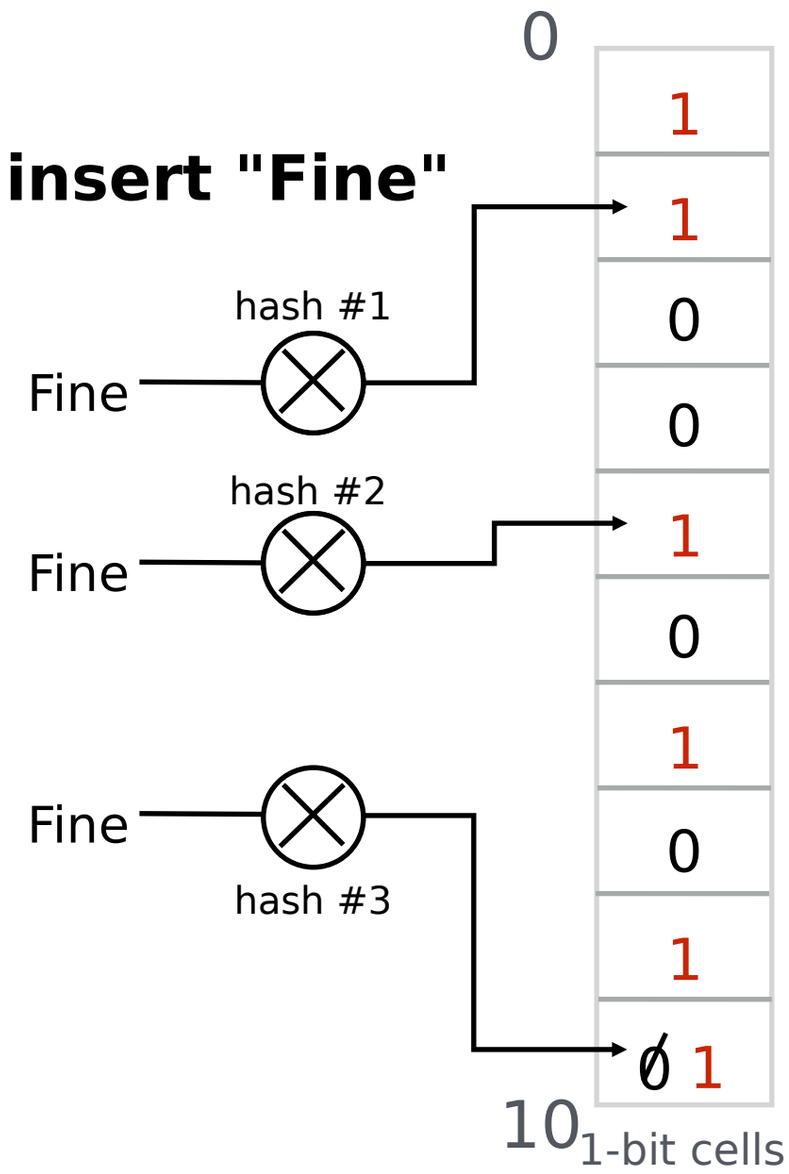
Bloom Filters: a more memory-efficient approach for insertions and membership queries



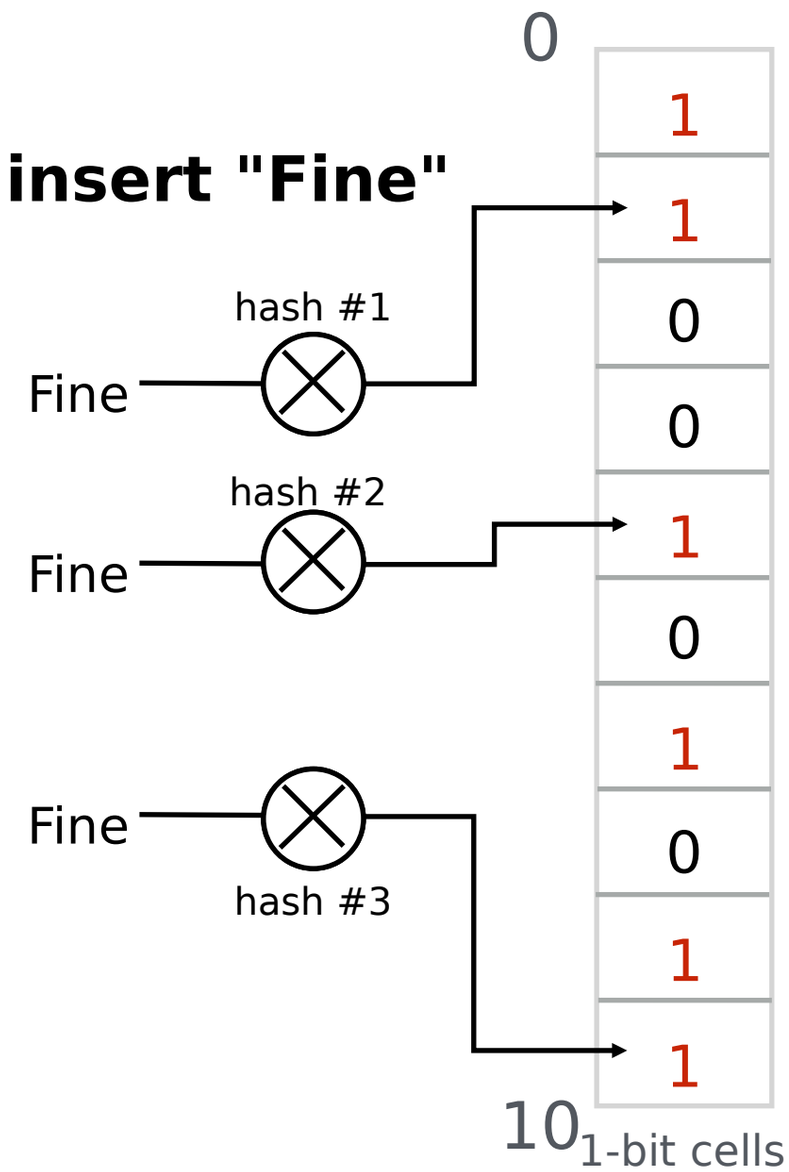
Bloom Filters: a more memory-efficient approach for insertions and membership queries



Bloom Filters: a more memory-efficient approach for insertions and membership queries



Bloom Filters: a more memory-efficient approach for insertions and membership queries



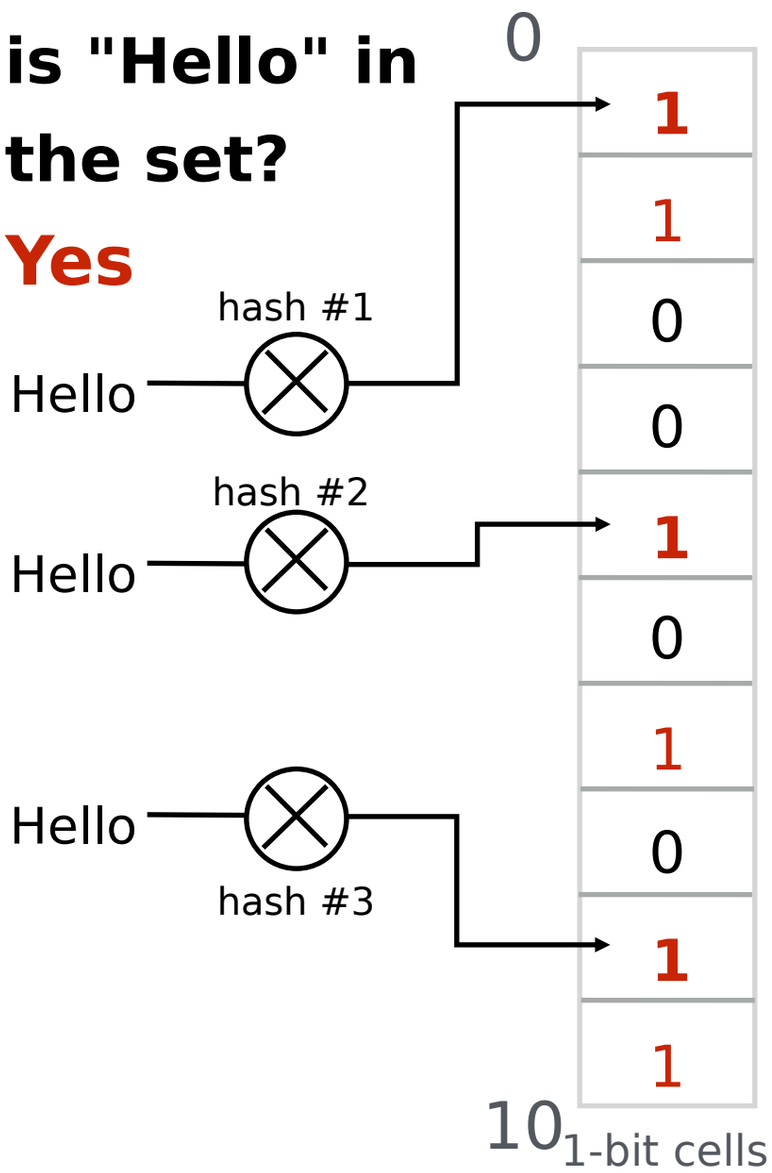
An element is considered in the set if **all** the hash values map to a cell with 1

An element is not in the set if **at least** one hash value maps to a cell with 0

Bloom Filters: a more memory-efficient approach for insertions and membership queries

is "Hello" in the set?

Yes

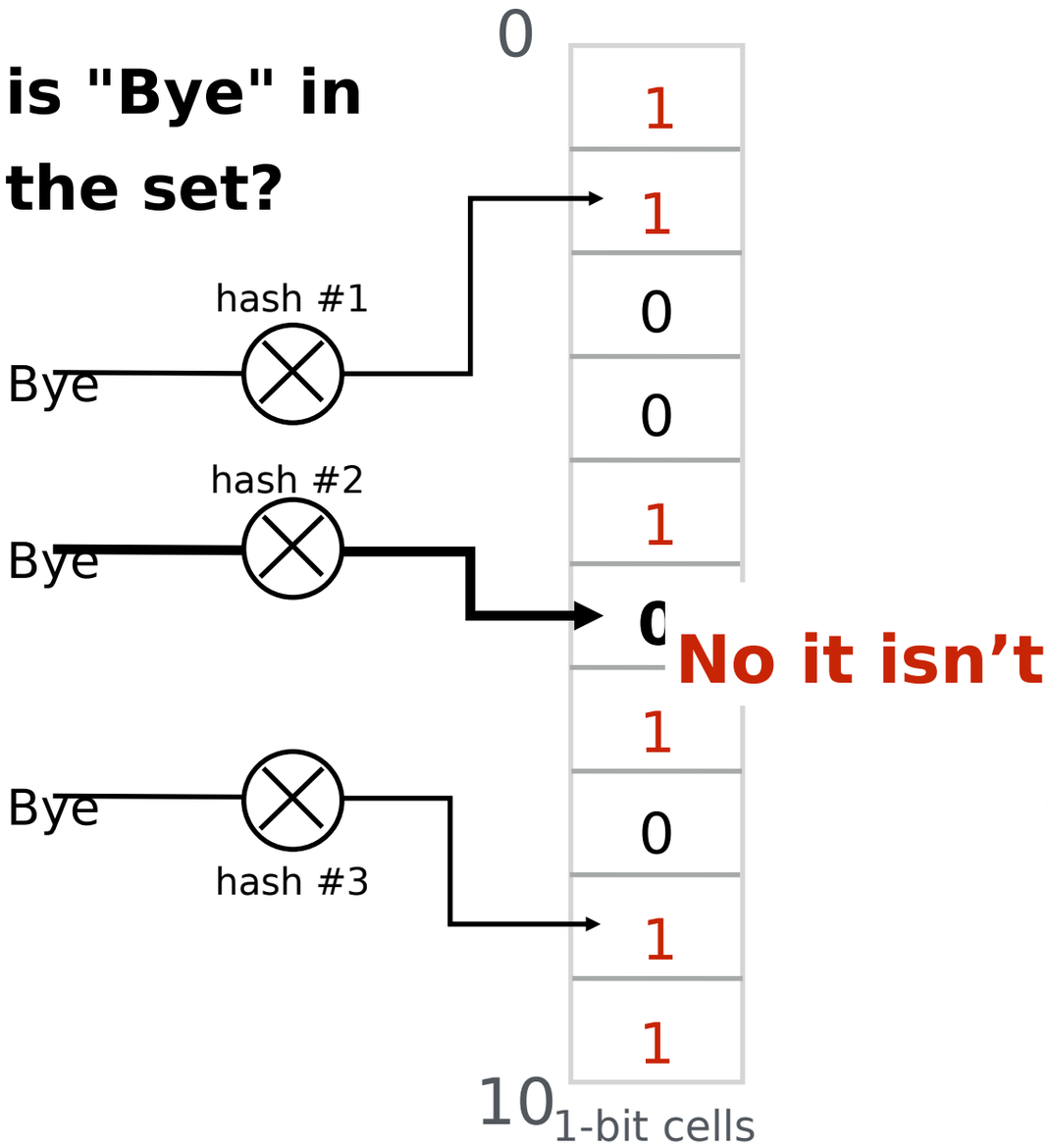


An element is considered in the set if **all** the hash values map to a cell with 1

An element is not in the set if **at least** one hash value maps to a cell with 0

Bloom Filters: a more memory-efficient approach for insertions and membership queries

is "Bye" in the set?



No it isn’t

An element is considered in the set if **all** the hash values map to a cell with 1

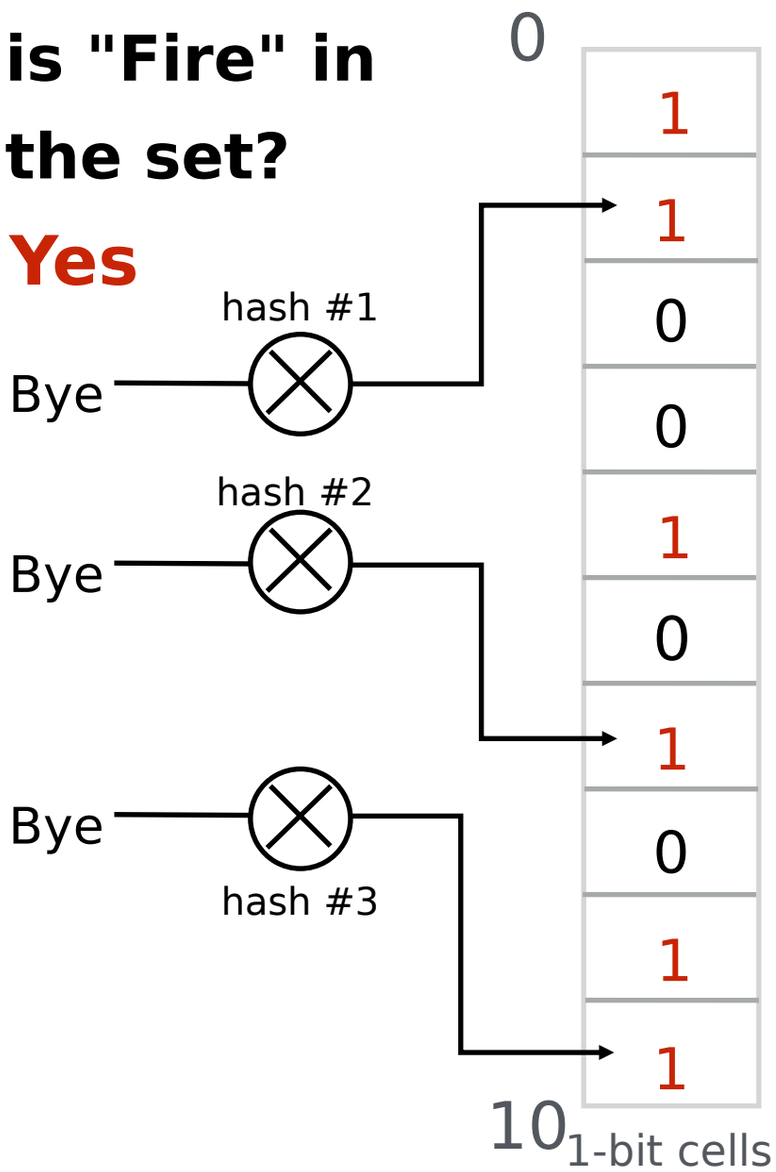
An element is not in the set if **at least** one hash value maps to a cell with 0

Bloom Filters: a more memory-efficient approach for insertions and membership queries

False Positive!

is "Fire" in the set?

Yes



An element is considered in the set if **all** the hash values map to a cell with 1

An element is not in the set if **at least** one hash value maps to a cell with 0

Bloom Filters: a more memory-efficient approach for insertions and membership queries

N elements, M cells and K *independent* hash functions

probability that one hash function returns the index of a particular cell $\frac{1}{M}$

probability that one hash function does not return the index of a particular cell $1 - \frac{1}{M}$

probability of a cell to be 0 $(1 - \frac{1}{M})^{KN}$

false positive rate $(1 - (1 - \frac{1}{M})^{KN})^{K}$

false negative rate 0

Bloom Filters: a more memory-efficient approach for insertions and membership queries

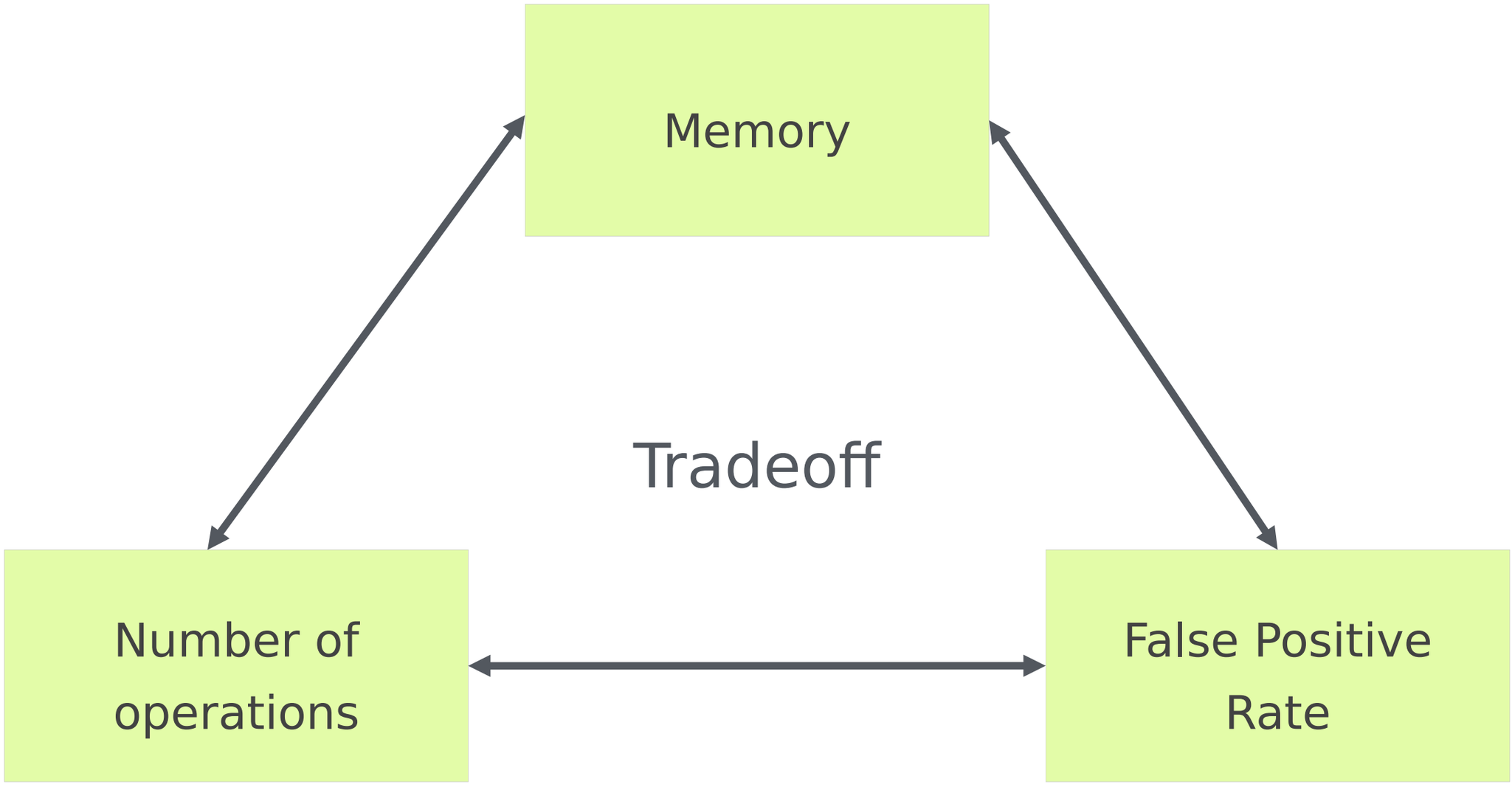
# of elements	# of cells	# hash functions	FPR
1000	10000	7	0.82%
1000	100000	7	≈0%

Bloom Filters: a more memory-efficient approach for insertions and membership queries

Pro: consumes roughly 10x less memory than the simple approach

Con: Requires slightly more operations than the simple approach (7 hashes instead of just 1)

Dimension your Bloom Filter



Dimension your Bloom Filter

N elements

M cells

K hash functions

FP false positive rate

Dimension your Bloom Filter

N elements

M cells

K hash functions

FP false positive rate

asymptotic approx.

$$FP = (1 - (1 - \frac{1}{M})^{KN})^K \approx (1 - e^{-KN/M})^K$$

with calculus you can dimension your bloom filter

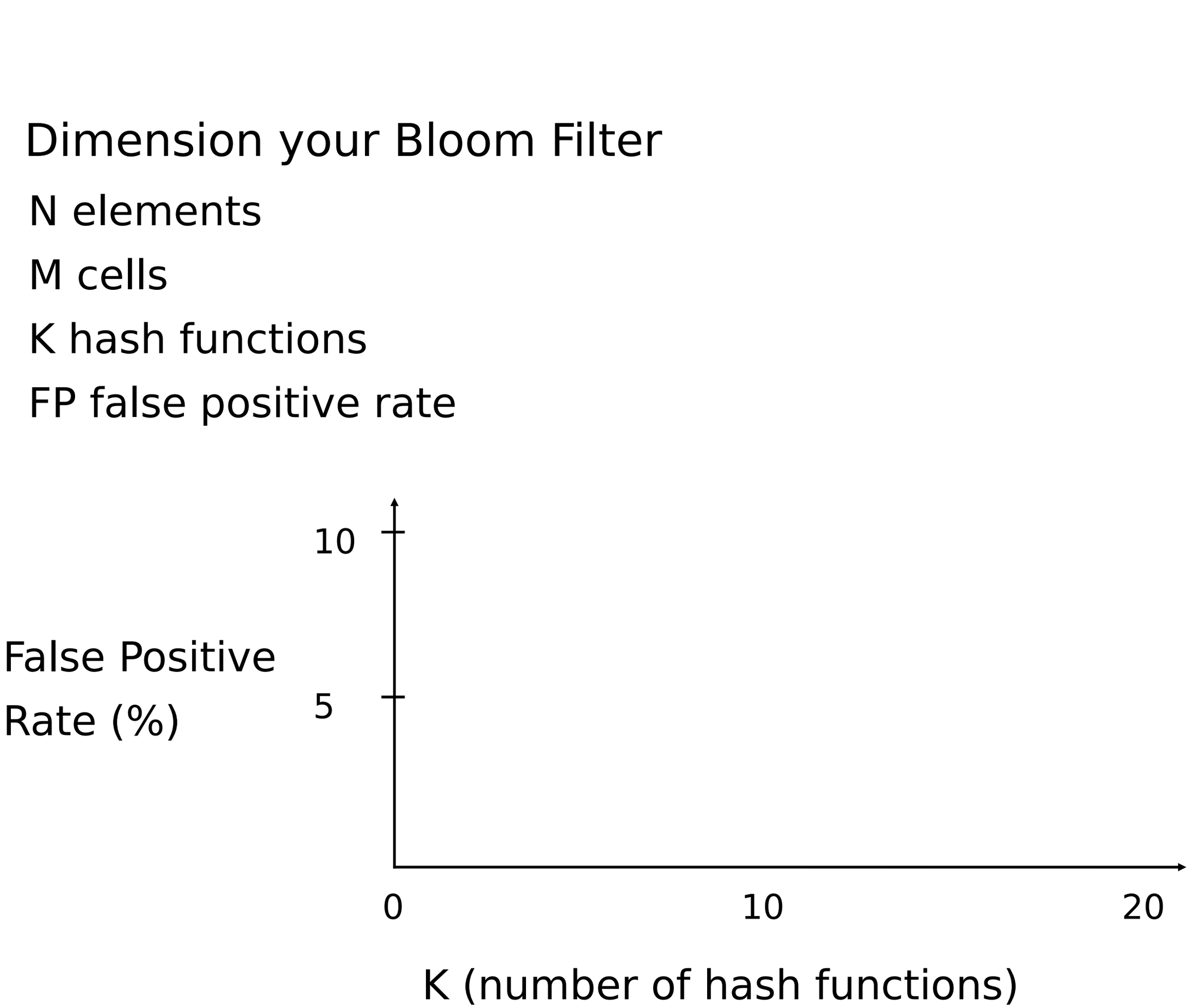
Dimension your Bloom Filter

N elements

M cells

K hash functions

FP false positive rate



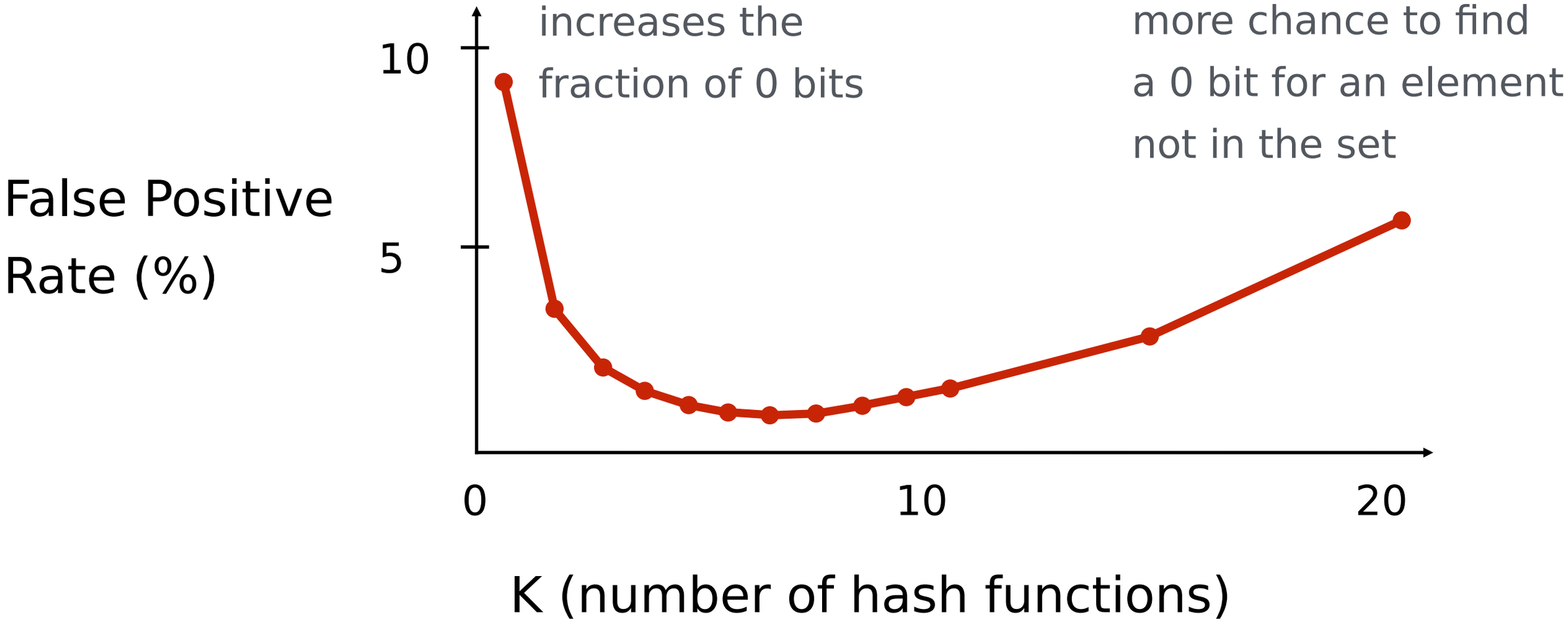
Dimension your Bloom Filter

N elements

M cells

K hash functions

FP false positive rate



Dimension your Bloom Filter

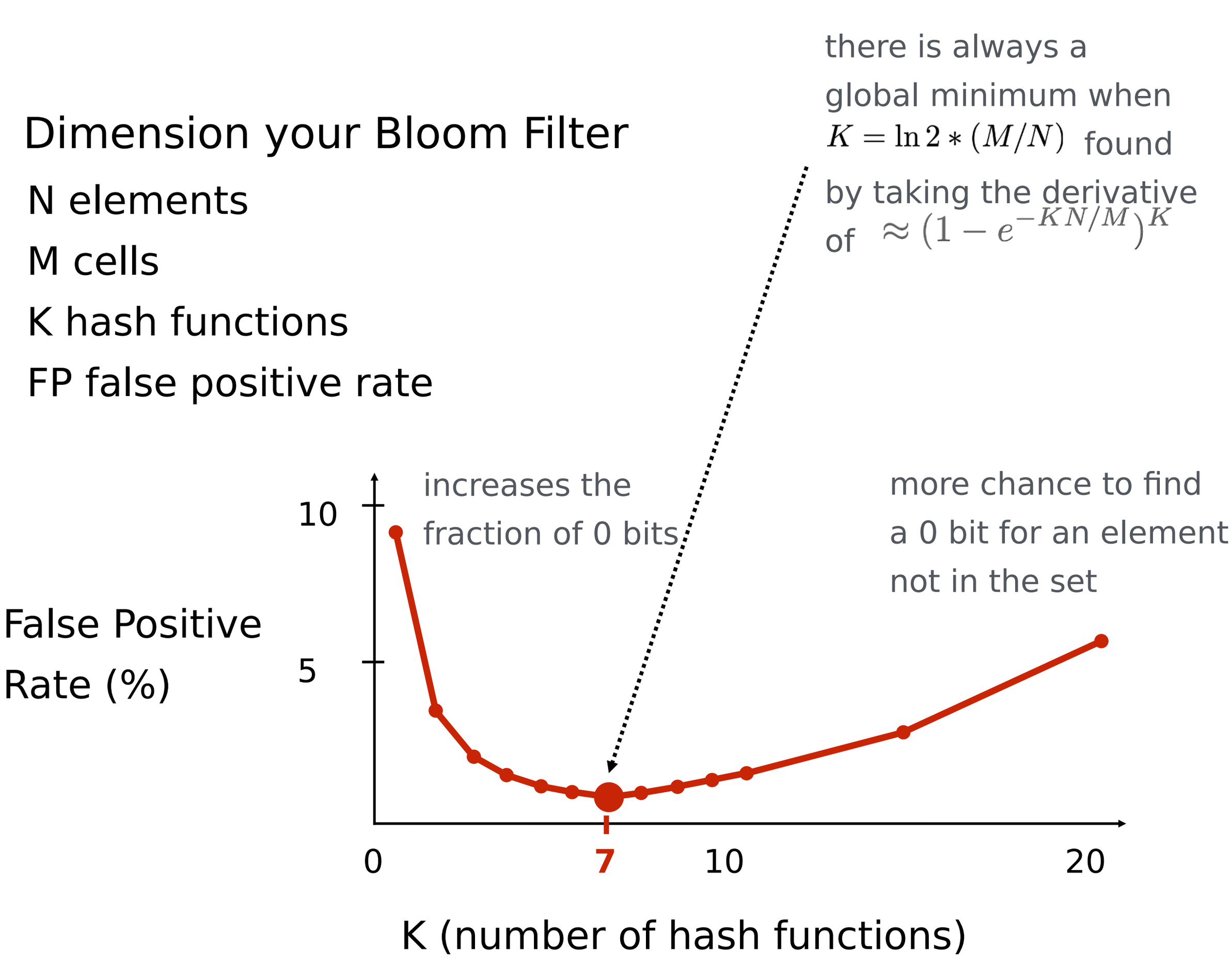
N elements

M cells

K hash functions

FP false positive rate

there is always a global minimum when $K = \ln 2 * (M/N)$ found by taking the derivative of $\approx (1 - e^{-KN/M})^K$



Implementation of a Bloom Filter in $P4_{16}$

You will have to use hash functions

v1model

```
enum HashAlgorithm {
    crc32,
    crc32_custom,
    crc16,
    s,
    random,
    identity,
    csum16,
    xor16
}
```

```
extern void hash<O, T, D, M>(out O result,
  in HashAlgorithm algo, in T base, in D data, in M
max);
```

more info https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4

Implementation of a Bloom Filter in $P4_{16}$

You will have to use hash functions, as well as registers

v1model

```
extern register<T> {

    register(bit<32> size);
    void read(out T result, in bit<32> index);
    void write(in bit<32> index, in T value);
}
```

more info https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4

Implementation of a Bloom Filter in $P4_{16}$

with 2 hash functions

```
control MyIngress(…) {
    register register<bit<1>>(NB_CELLS) bloom_filter;
```

Implementation of a Bloom Filter in $P4_{16}$

with 2 hash functions

```
control MyIngress(…) {
    register register<bit<1>>(NB_CELLS) bloom_filter;
    apply {
       hash(meta.index1, HashAlgorithm.my_hash1, 0,
         {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);
       hash(meta.index2, HashAlgorithm.my_hash2, 0,
         {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);
```

Implementation of a Bloom Filter in $P4_{16}$

with 2 hash functions

```
control MyIngress(…) {
    register register<bit<1>>(NB_CELLS) bloom_filter;
    apply {
       hash(meta.index1, HashAlgorithm.my_hash1, 0,
         {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);
       hash(meta.index2, HashAlgorithm.my_hash2, 0,
         {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);

       if (meta.to_insert == 1) {
         bloom_filter.write(meta.index1, 1);
         bloom_filter.write(meta.index2, 1);
       }
       if (meta.to_query == 1) {
         bloom_filter.read(meta.query1, meta.index1);
         bloom_filter.read(meta.query2, meta.index2);

         if (meta.query1 == 0 || meta.query2 == 0) {
           meta.is_stored = 0;
         }
         else {
           meta.is_stored = 1;
         }
       }
    }
}
```

Implementation of a Bloom Filter in $P4_{16}$

with 2 hash functions

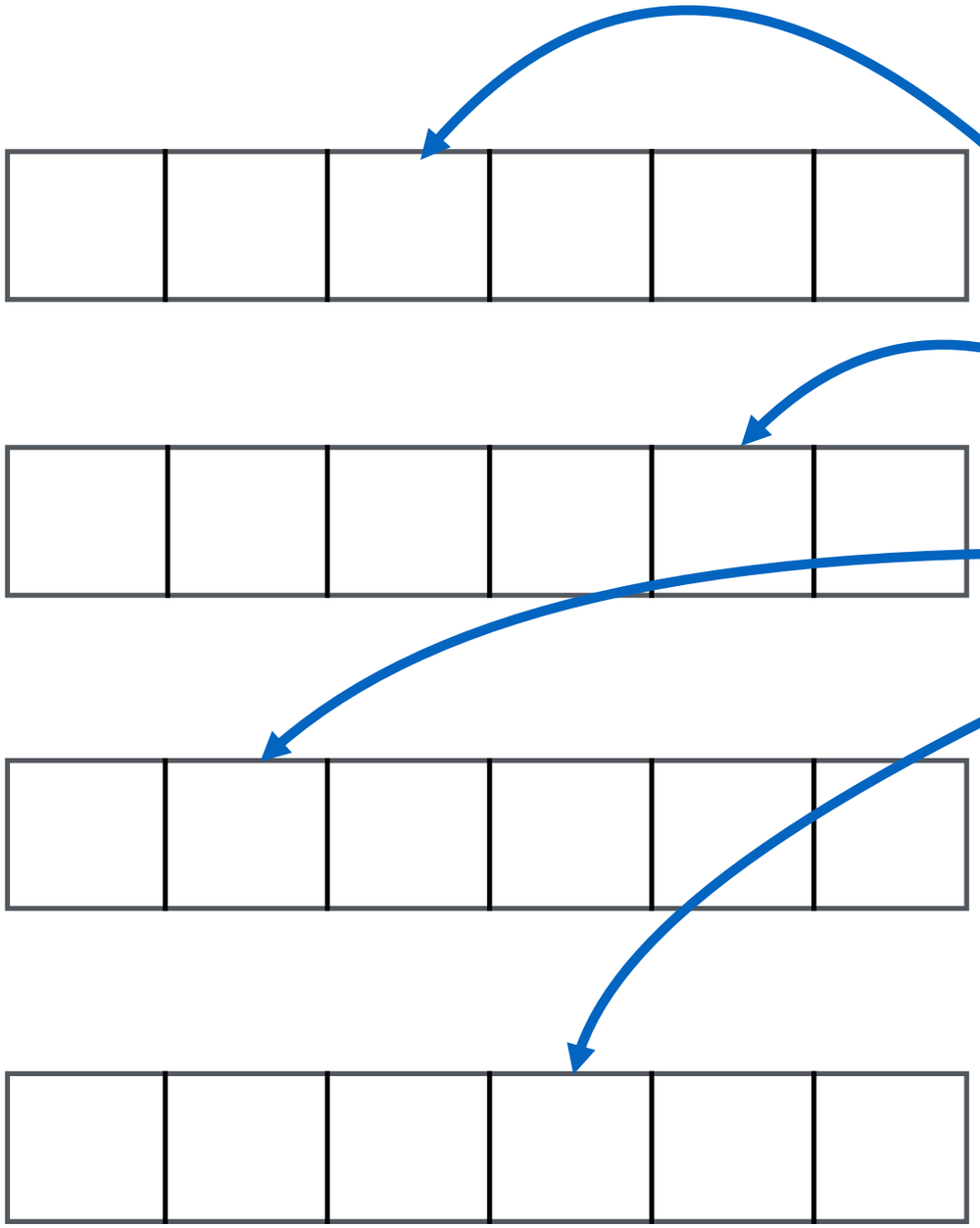
```
control MyIngress(…) {
    register register<bit<1>>(NB_CELLS) bloom_filter;
    apply {
       hash(meta.index1, HashAlgorithm.my_hash1, 0,
         {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);
       hash(meta.index2, HashAlgorithm.my_hash2, 0,
         {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);

       if (meta.to_insert == 1) {
         bloom_filter.write(meta.index1, 1);
         bloom_filter.write(meta.index2, 1);
       }
       if (meta.to_query == 1) {
         bloom_filter.read(meta.query1, meta.index1);
         bloom_filter.read(meta.query2, meta.index2);

         if (meta.query1 == 0 || meta.query2 == 0) {
           meta.is_stored = 0;
         }
         else {
           meta.is_stored = 1;
         }
       }
    }
}
```

Everything in bold red must be adapted for your program

Depending on the hardware limitations, splitting the bloom filter might be required



M cells are split into M/K disjoint groups

An element is hashed to K cells, one in each group

One hash function per group

About same performance, may be easier to implement or parallelize

Because deletions are not possible, the controller may need to regularly reset the bloom filters

Resetting a bloom filter takes some time during which it is not usable

Common trick: use two bloom filters and use one when the controller resets the other one

Bloom filters may be extended to allow deletions and to list the filter content.

If you are curious, check out the extended slides for:

- counting Bloom filters (allow deletions)
- invertible Bloom filters (allow to list content)

Bloom filters are **probabilistic data structures** for **set membership queries**. For more info, see:

Space/Time Trade-offs in Hash Coding
with Allowable Errors. Burton H. Bloom. 1970.

Network Applications of Bloom Filters: A Survey.
Andrei Broder and Michael Mitzenmacher. 2004.

Invertible Bloom Lookup Tables.
Michael T. Goodrich and Michael Mitzenmacher. 2015.

FlowRadar: A Better NetFlow for Data Centers
Yuliang Li et al. NSDI 2016.

← You are looking at a stream of data (packets).

Today, I'll show you how set membership and frequency queries can be realized in P4.

PART 1

Is a certain element (e.g. ip address) in the stream?

→ Bloom filter

PART 2

How frequently does an element appear?

→ CountMin Sketch, Count Sketch, ...

part 2: **counting with sketches**

How frequently does an element appear?

(slides by yours truly)

We are going to look at **frequencies**,

i.e. **how often** an element occurs in a data stream.

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \end{bmatrix}$$

vector of frequencies (counts)

of all ***distinct elements*** $\boldsymbol{x}_i$

We are going to look at **frequencies**,

i.e. **how often** an element occurs in a data stream.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \end{bmatrix}$$

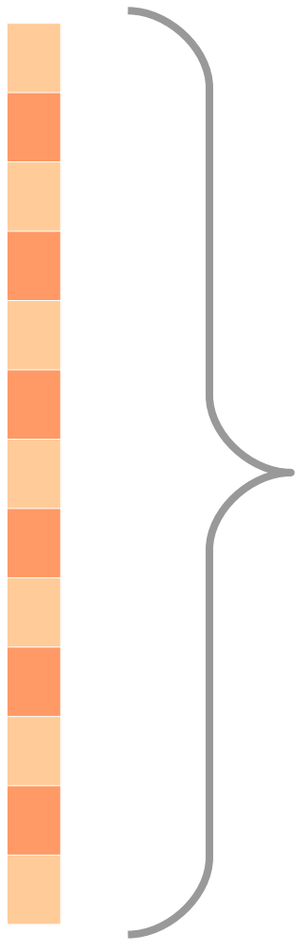
vector of frequencies (counts)

*of all **distinct elements** x_i*

e.g. flows, ip addresses, ...

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

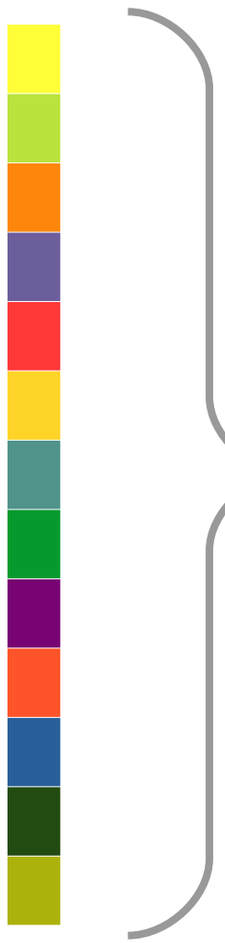
In the worst case, an algorithm providing **exact frequencies** requires **linear space**.



Data Stream

n elements in total

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



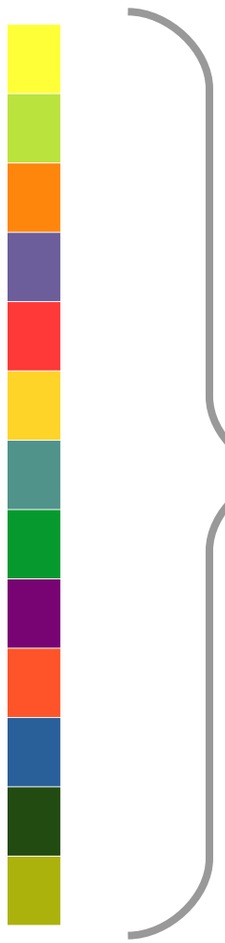
Data Stream

n elements in total

→ **n distinct elements**

(in the worst case)

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

More efficient by allowing false positives.

Probabilistic datastructures can help again!

Bloom Filters

quickly "filter" only those elements that might be in the set

More efficient by allowing false positives.

Sketches

provide a approximate frequencies of elements in a data stream.

More efficient by allowing mis-counting.

A **CountMin sketch** is **designed** to have **provable L1 error bounds** for frequency queries.

A **CountMin sketch** is **designed** to have **provable L1 error bounds** for frequency queries.

Notation reminder:

vector of frequencies (counts) of all **distinct elements x_i**

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \end{bmatrix}$$

A **CountMin sketch** is **designed** to have **provable**

L1 error bounds for frequency queries.

$$Pr\left[\ \widehat{x}_i - x_i \geq \varepsilon \|\boldsymbol{x}\|_1\ \right] \leq \delta$$

estimated frequency — *true frequency* — sum *of frequencies*

The estimation error ***exceeds*** $\varepsilon \|\boldsymbol{x}\|_1$

with a ***probability smaller than*** δ

relative to L1 norm

$$Pr\left[\ \hat{x}_i\ -\ x_i\ \geq\ \varepsilon \left\|\boldsymbol{x}\right\|_1\ \right] \leq \delta$$

estimated frequency — *true frequency* — sum *of frequencies*

The estimation error ***exceeds*** $\varepsilon \left\|\boldsymbol{x}\right\|_1$

with a ***probability smaller than*** δ

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

Let $\|\mathbf{x}\|_1 = 10000\,, \quad \varepsilon = 0.01\,, \quad \delta = 0.05$

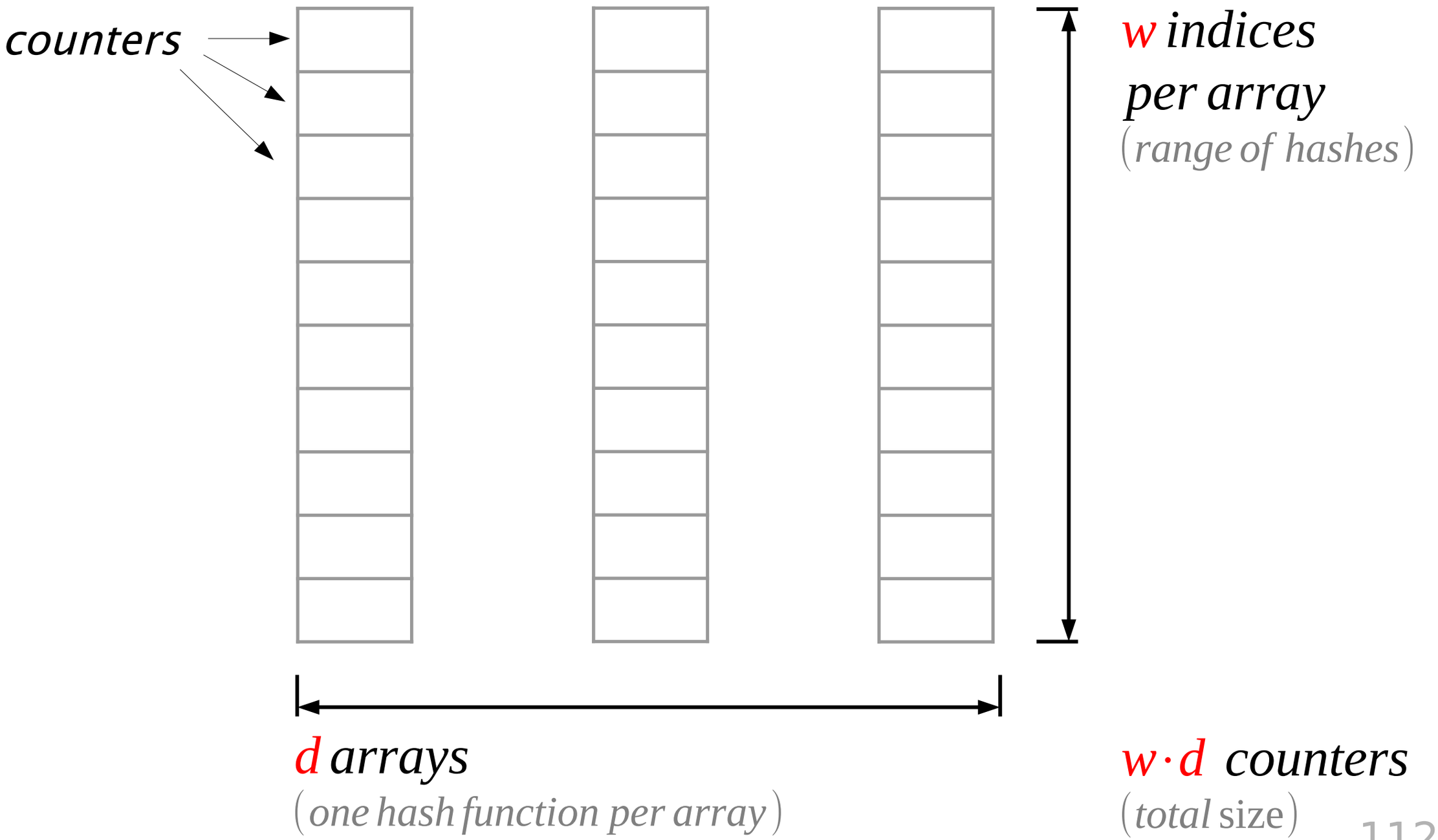
Then, after counting ***10000 elements in total****,*

the probability for ***any estimate*** *to be*

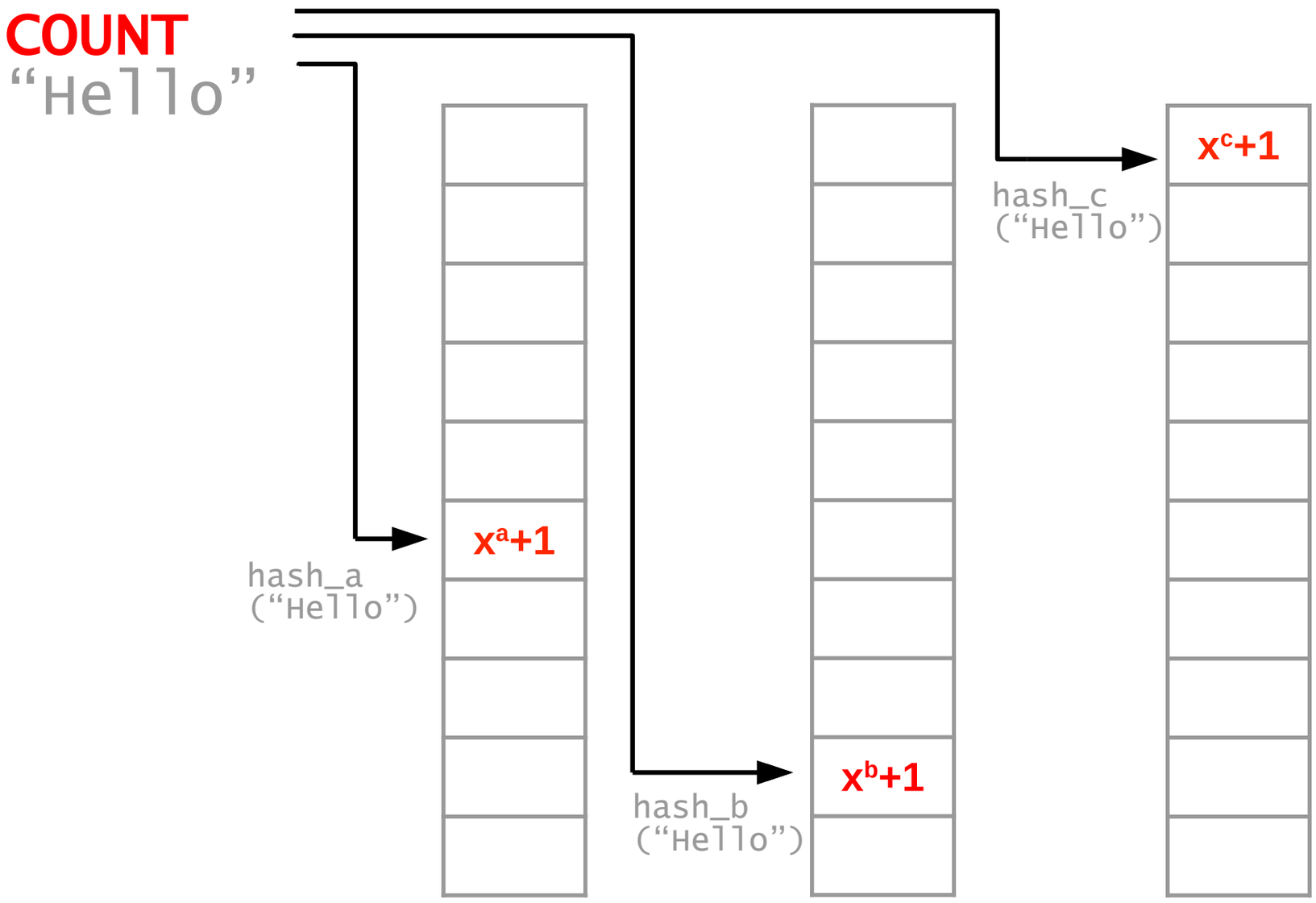
off by ***more than 100*** *is* ***less than 5%.***

A **CountMin sketch** is **designed** to have **provable L1 error bounds** for frequency queries.

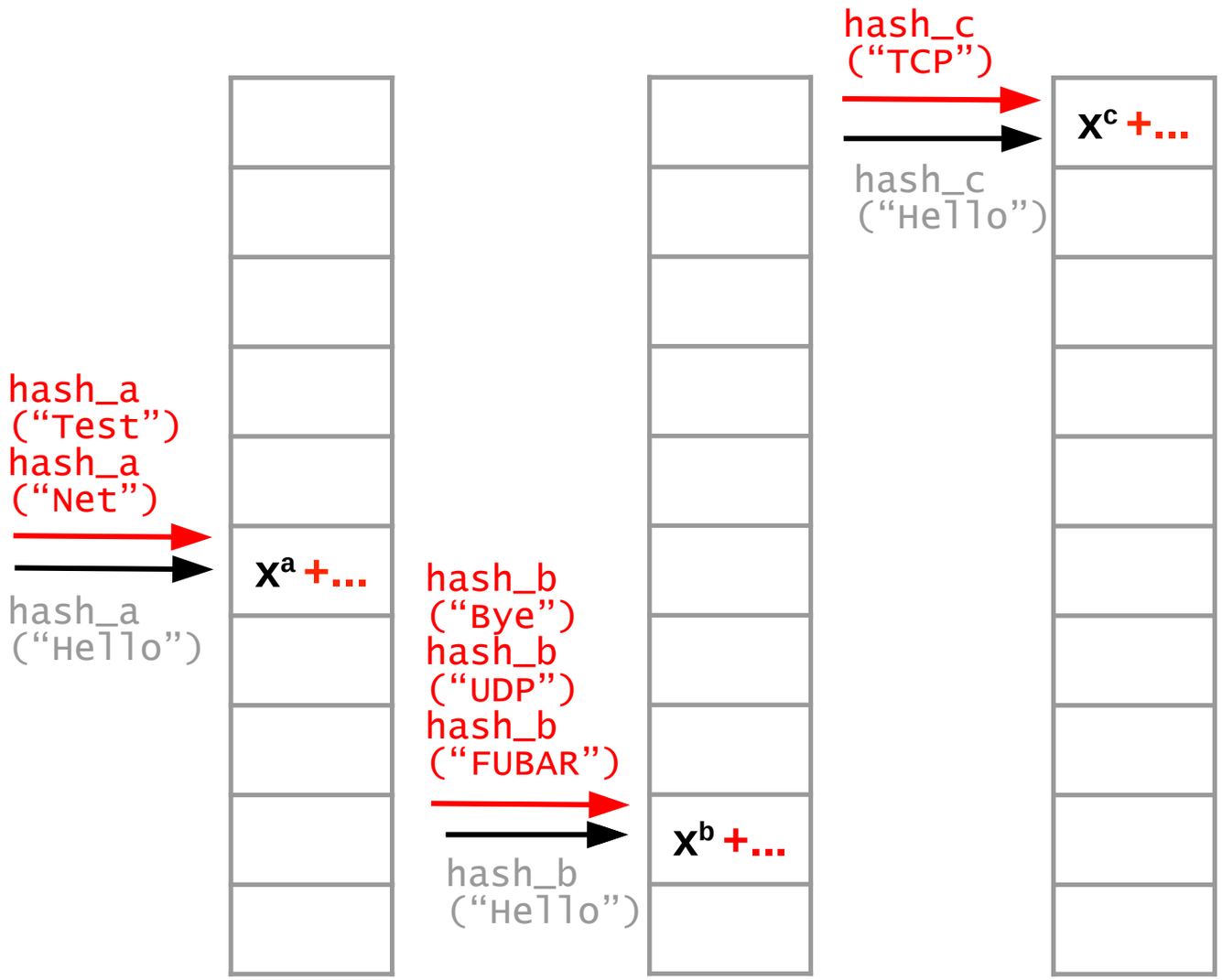
A **CountMin** Sketch uses multiple arrays and hashes.



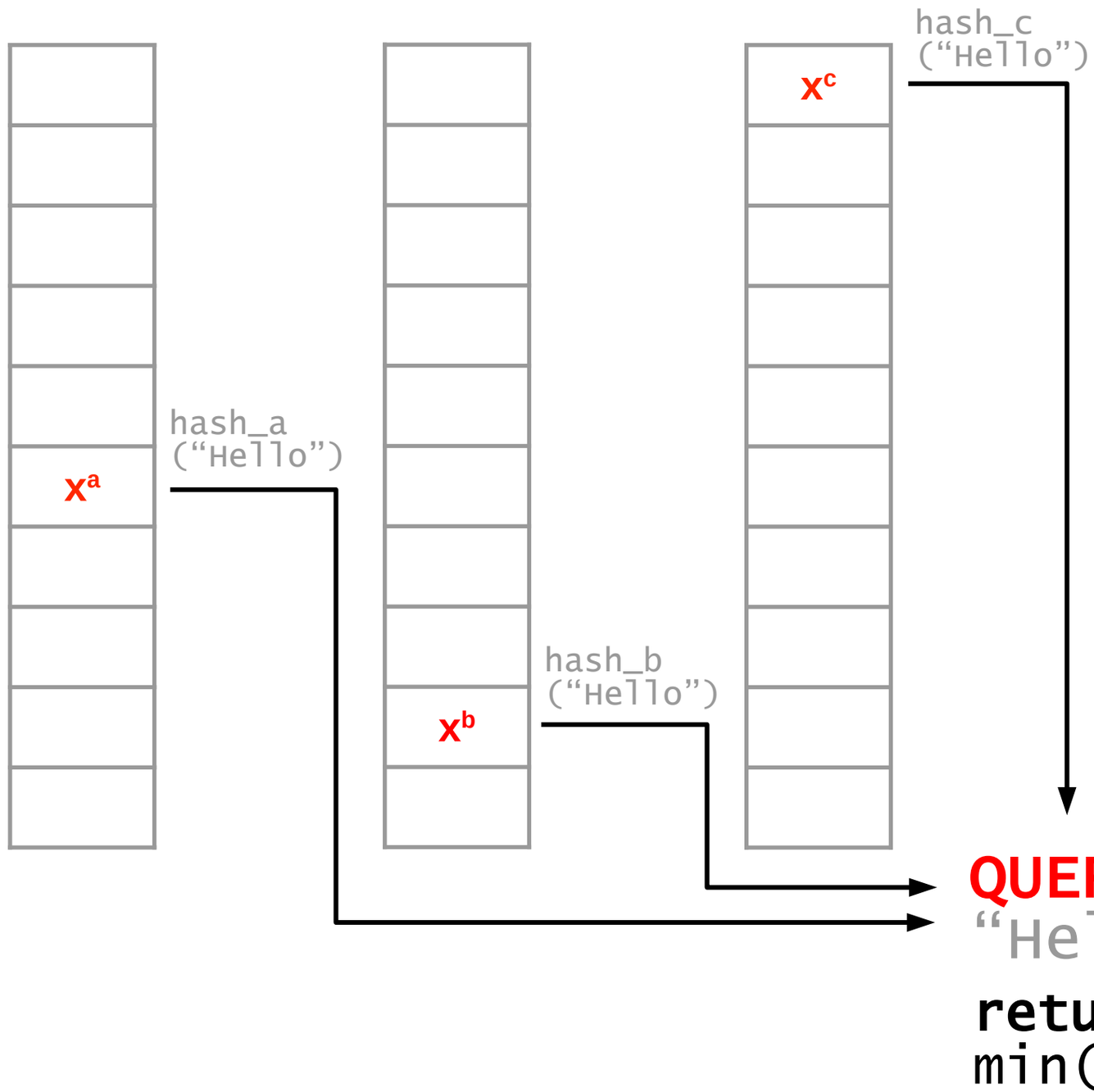
To count, increment **all** hash-indexed fields by 1.



Hash collisions cause over-counting.



Returning the **minimum value** minimizes the error.



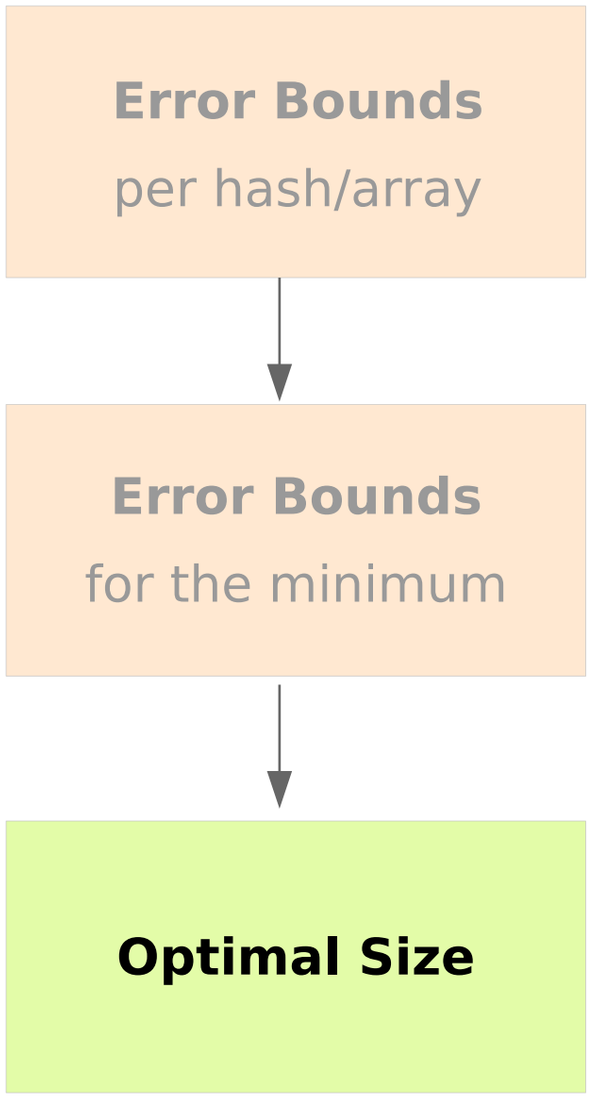
A **CountMin sketch** is designed to have **provable**

L1 error bounds for frequency queries.

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

A **CountMin** sketch recipe

(see extended slides for derivation)



Given ε, δ **, choosing**

$$w = \left\lceil \frac{e}{\varepsilon} \right\rceil$$ *(hash range)*

$$d = \left\lceil \ln \frac{1}{\delta} \right\rceil$$ *(#hashes)*

requires the **minimum number of counters** s.t. the CountMin Sketch can guarantee that

$$\hat{x}_i - x_i \geq \varepsilon \left\| \boldsymbol{x} \right\|_1$$

with a probability less than δ

A **CountMin sketch** is **designed** to have **provable L1 error bounds** for frequency queries.

***CountMin** sketch recipe*

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

Then $\hat{x}_i - x_i \geq \varepsilon \left\| \boldsymbol{x} \right\|_1$ *with a probability less than* δ

Sketches **are the new black**

OpenSketch

NSDI ‘13

[source]

Software Defined Traffic Measurement with OpenSketch

Minlan Yu† *Lavanya Jose** *Rui Miao*†
† *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 NetFPGA, 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 control without measurement is blind, 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).

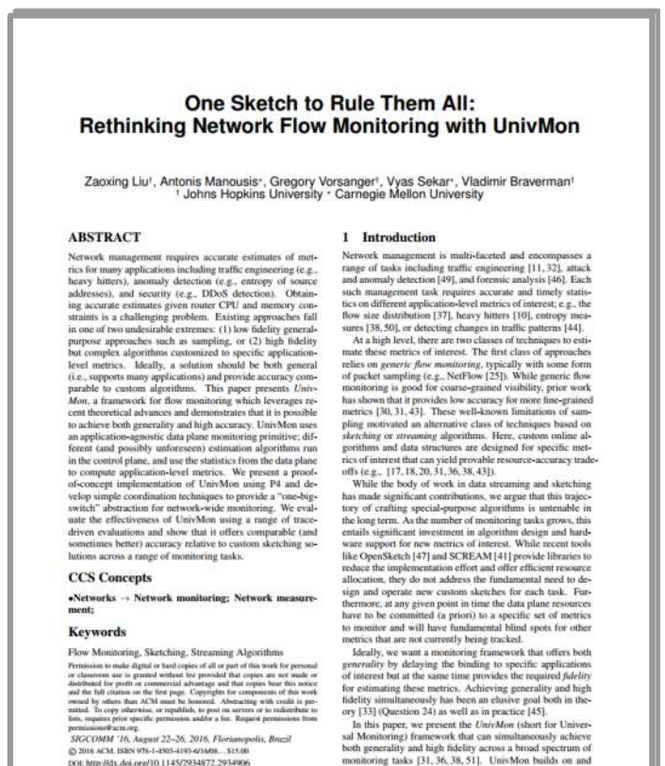
Flow-based measurements such as NetFlow [2] and sFlow [42] provide generic support for different measurement tasks, but consume too resources (e.g., CPU, memory, bandwidth) [28, 18, 19]. For example, to identify the big flows whose byte volumes are above a threshold (i.e., heavy hitter detection which 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†, 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

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.

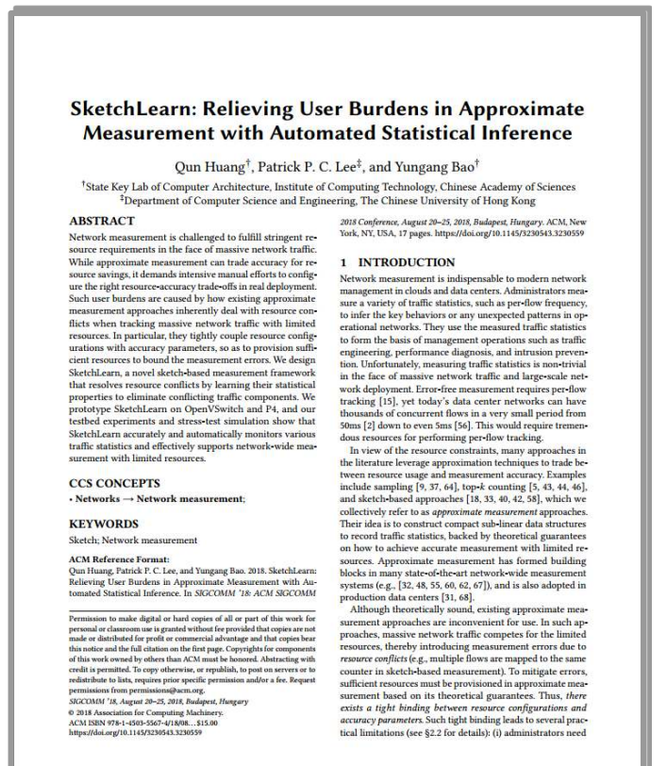
Ideally, we want a monitoring framework that offers both *generality* by delaying the binding to specific applications of interest but at the same time provides the required *fidelity* for estimating these metrics. Achieving generality and high fidelity simultaneously has been an elusive goal both in 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†, Patrick P. C. Lee‡, and Yungang Bao†
†State Key Lab of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
‡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

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

ACM ISBN 978-1-4503-5567-4/18/08…$15.00
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 5ms [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, 60, 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

Sketches **are the new black**

LightGuardian

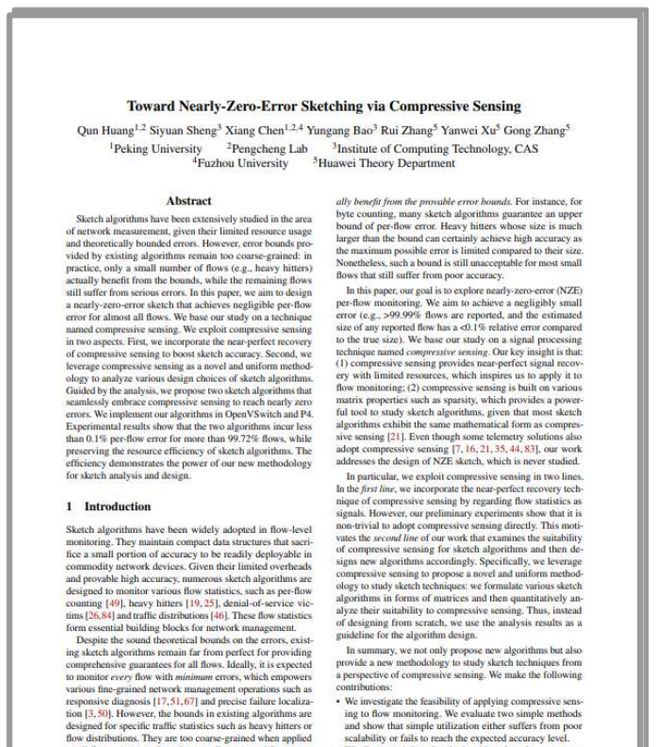
NSDI ‘21

[source]

Nearly-Zero-Error

NSDI ‘21

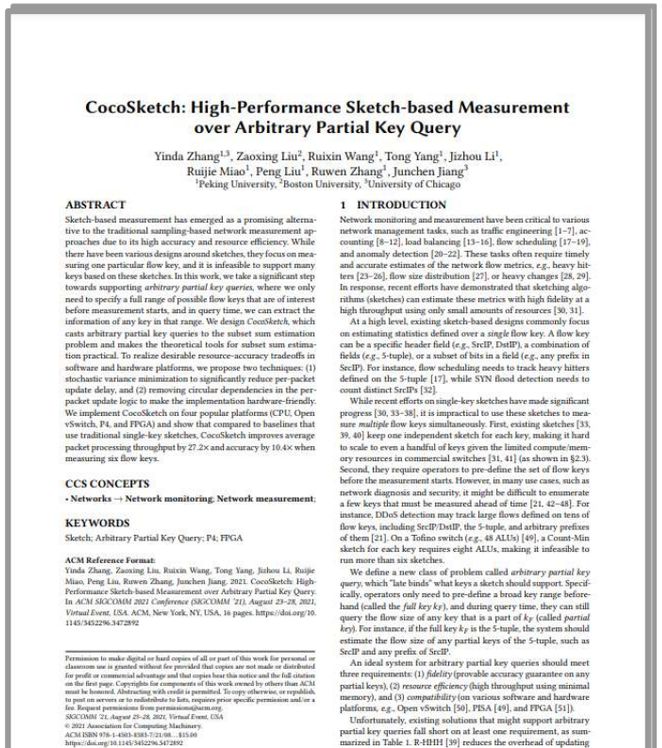
[source]



CocoSketch

SIGCOMM ‘21

[source]



Sketches have limitations. They **compute statistical summaries** and favor elements with **high frequency**.

$$Pr\left[\hat{x}_i - x_i \geq \varepsilon \left\|\boldsymbol{x}\right\|_1\right] \leq \delta$$

estimation error

relative to sum of all elements

Sketches have limitations. They **compute statistical summaries** and favor elements with **high frequency**.

$$\textit{Let} \quad \varepsilon = 0.01, \quad \|\boldsymbol{x}\|_1 = 10000 \quad \left(\Rightarrow \varepsilon \cdot \|\boldsymbol{x}\|_1 = 100\right)$$

$$\textit{Assume two flows} \quad x_a, \quad x_b,$$

$$\textit{with} \quad \|x_a\|_1 = 1000, \quad \|x_b\|_1 = 50$$

high frequency

low frequency

Sketches have limitations. They **compute statistical summaries** and favor elements with **high frequency**.

$$Let \quad \varepsilon = 0.01, \quad \|\boldsymbol{x}\|_1 = 10000 \quad \left(\Rightarrow \varepsilon \cdot \|\boldsymbol{x}\|_1 = 100\right)$$

$$Assume\ two\ flows \quad x_a, \quad x_b,$$

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

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

Sketches have limitations. They **compute statistical summaries** and favor elements with **high frequency**.

$$Let \quad \varepsilon = 0.01\,, \quad \|\boldsymbol{x}\|_1 = 10000 \quad \left(\Rightarrow \varepsilon \cdot \|\boldsymbol{x}\|_1 = 100\right)$$

$$Assume\ two\ flows \quad x_a\,, \quad x_b\,,$$

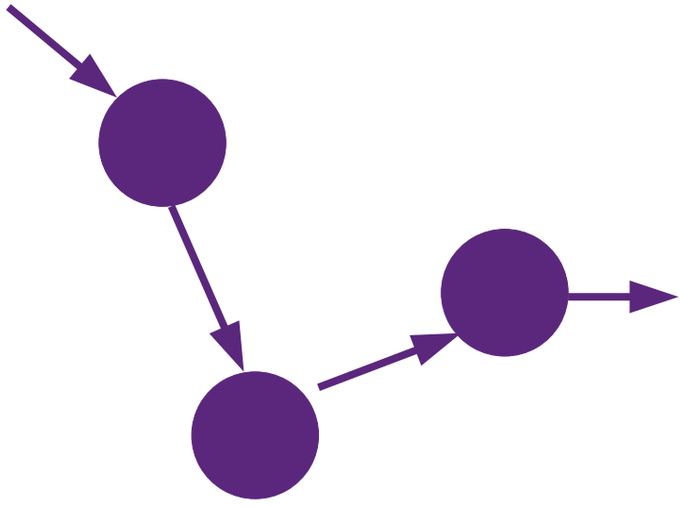
$$with \quad \|x_a\|_1 = 1000\,, \quad \|x_b\|_1 = 50$$

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

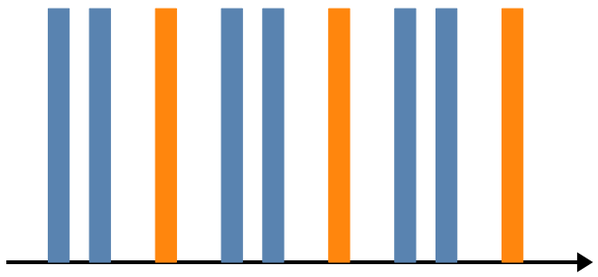
flow size: x_a: 10%, x_b: **200%**

Other problems a sketch **can't handle**:

causality



patterns



rare things

← You are looking at a stream of data (packets). Today, I'll show you how set membership and frequency queries can be realized in P4.

PART 1

Is a certain element (e.g. ip address) in the stream?

→ Bloom filter

PART 2

How frequently does an element appear?

→ CountMin Sketch, Count Sketch, ...

TAKEAWAY

Probabilistic data structures provide **trade-offs between resources and error**, and **provable guarantees** to rely on.

Advanced Topics in Communication Networks

Laurent Vanbever

nsg.ee.ethz.ch

ETH Zürich

Tue 11 Oct 2022