

Use cases of Programmable Dataplane (P4)

ECE/CS598HPN

Radhika Mittal

Which paper(s) did you read?

NetCache

Slides borrowed from the authors' SOSPP'17 presentation

Goal: fast and cost-efficient rack-scale key-value storage

❑ Store, retrieve, manage key-value objects

- Critical building block for large-scale cloud services



- Need to **meet aggressive latency and throughput objectives efficiently**

❑ Target workloads

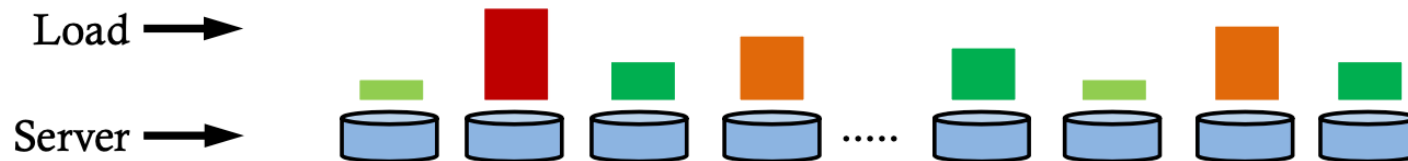
- Small objects
- Read intensive
- **Highly skewed and dynamic key popularity**

Key challenge: **highly-skewed** and rapidly-changing workloads

low throughput

&

high tail latency

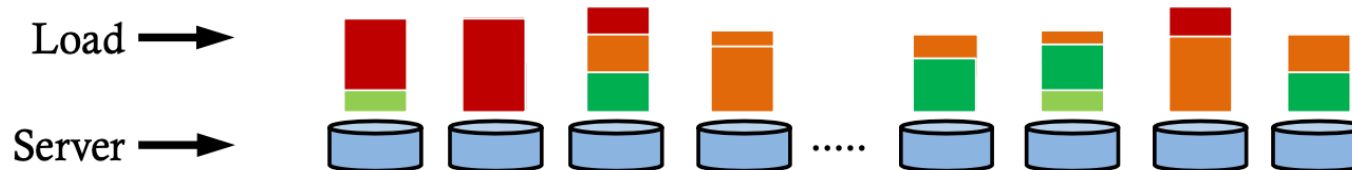


Key challenge: highly-skewed and rapidly-changing workloads

low throughput

&

high tail latency



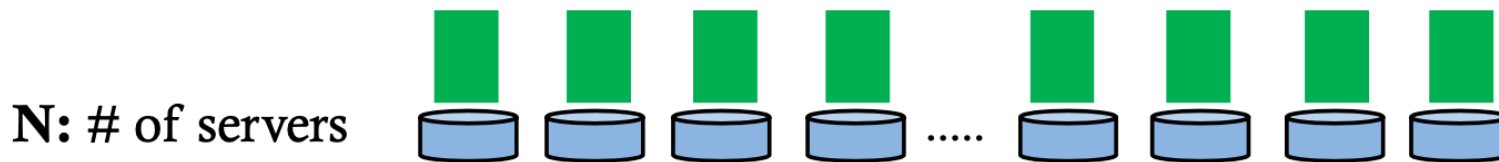
Q: How to provide effective dynamic load balancing?

Opportunity: fast, small cache can ensure load balancing

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

Cache $O(N \log N)$ hottest items

E.g., 10,000 hot objects



E.g., 100 backends with 100 billions items

Requirement: cache throughput $\geq$ backend aggregate throughput

NetCache: towards billions QPS key-value storage rack

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

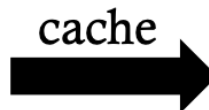


flash/disk

each: $O(100)$ KQPS

total: $O(10)$ MQPS

storage layer



in-memory

$O(10)$ MQPS

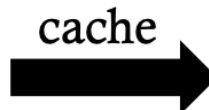
cache layer



in-memory

each: $O(10)$ MQPS

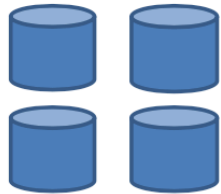
total: $O(1)$ BQPS



$O(1)$ BQPS

NetCache: towards billions QPS key-value storage rack

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



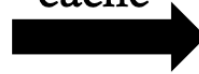
flash/disk

each: $O(100)$ KQPS

total: $O(10)$ MQPS

storage layer

cache



in-memory

$O(10)$ MQPS

cache layer

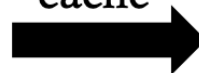


in-memory

each: $O(10)$ MQPS

total: $O(1)$ BQPS

cache



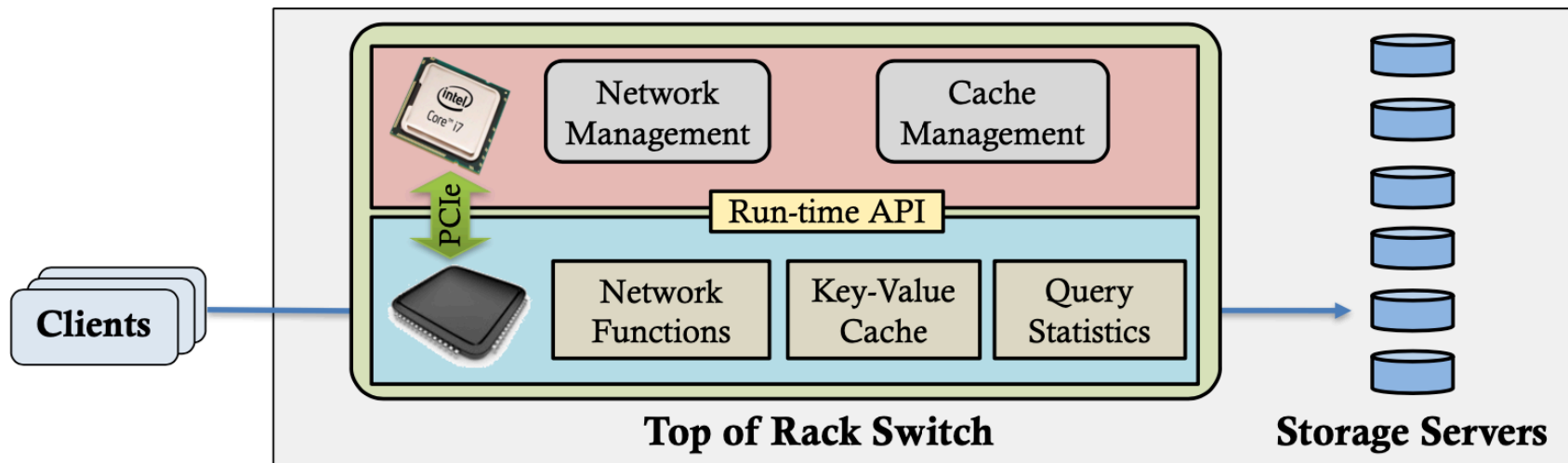
in-network

$O(1)$ BQPS

Small on-chip memory?

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

NetCache rack-scale architecture



❑ Switch data plane

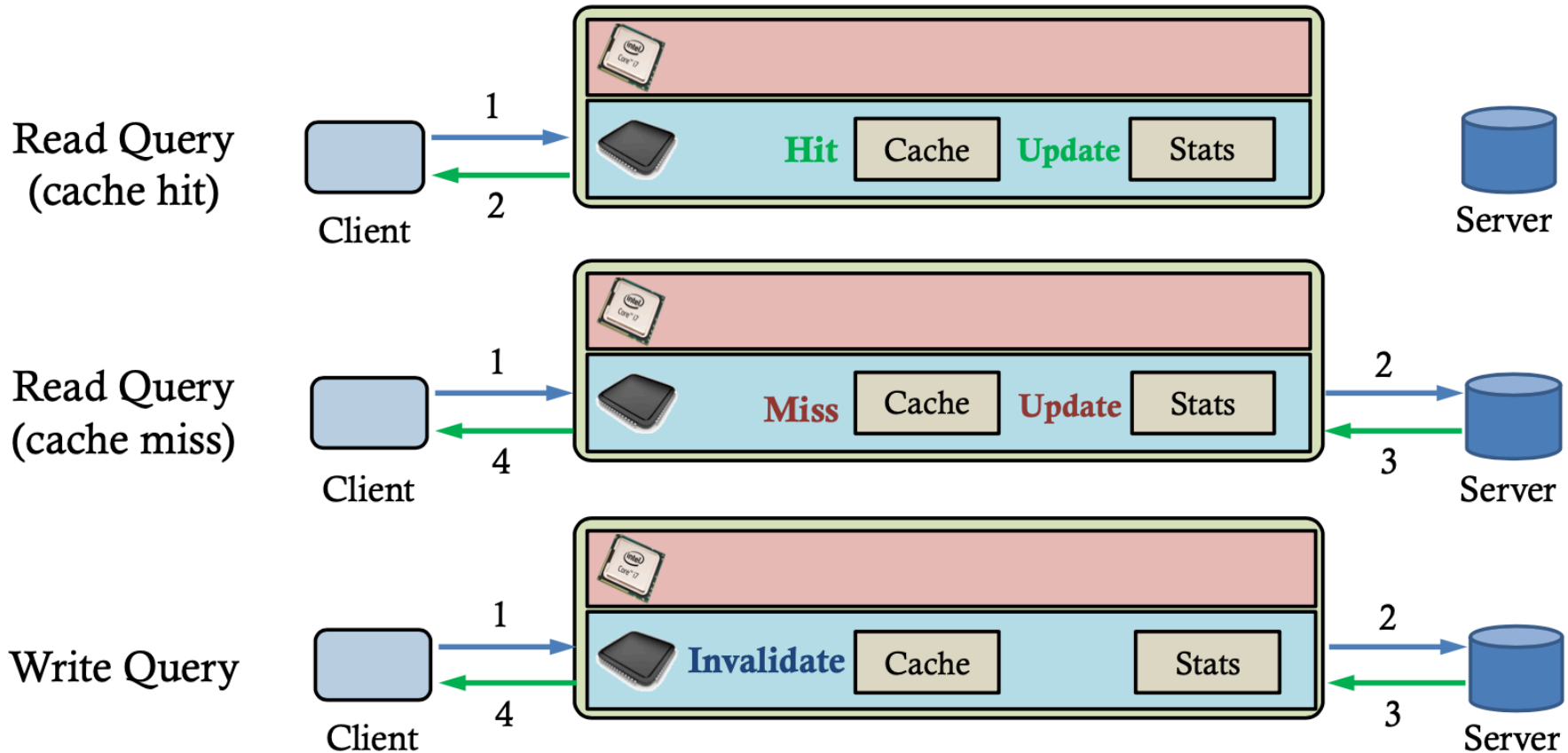
- Key-value store to serve queries for cached keys
- Query statistics to enable efficient cache updates

❑ Switch control plane

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

Assume the entire rack is dedicated to key-value storage.

Data plane query handling



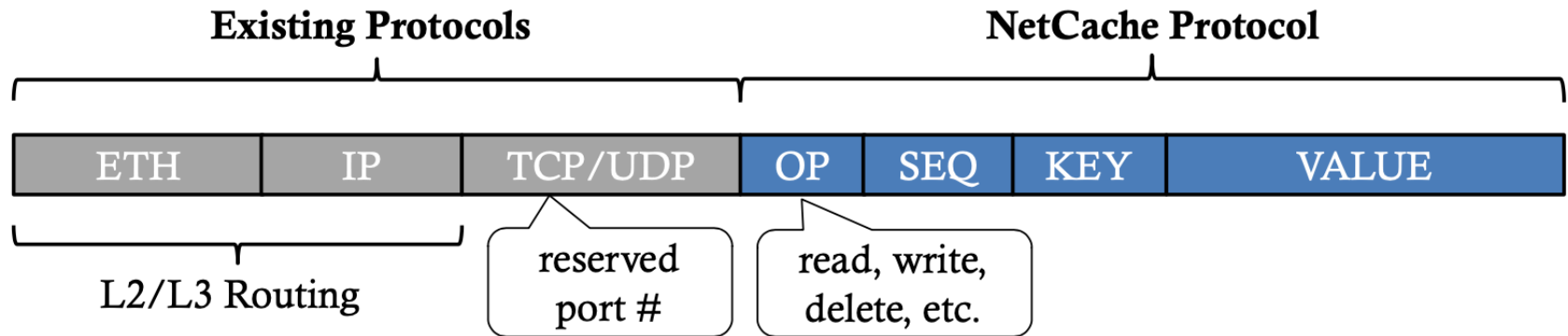
Key-value caching in network ASIC at line rate ?!

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

Key-value caching in network ASIC at line rate

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

NetCache Packet Format



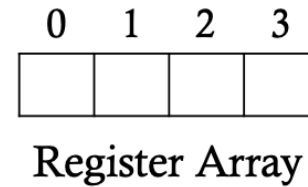
- ❑ Application-layer protocol: compatible with existing L2-L4 layers
- ❑ Only the top of rack switch needs to parse NetCache fields

Key-value caching in network ASIC at line rate

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

Key-value store using register array in network ASIC

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



Key-value store using register array in network ASIC

Match	<code>pkt.key == A</code>	<code>pkt.key == B</code>
Action	<code>process_array(0)</code>	<code>process_array(1)</code>

pkt.value:

A

B

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

0	1	2	3
A	B		

Register Array

Combine outputs from multiple arrays

Lookup Table

Match	pkt.key == A
Action	bitmap = <u>111</u> index = 0

pkt.value:

A0	A1	A2
----	----	----

Bitmap indicates arrays that store the key's value

Index indicates slots in the arrays to get the value

Minimal hardware resource overhead

Value Table 0

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

→

0	1	2	3
<u>A0</u>			

 Register Array 0

Value Table 1

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

→

<u>A1</u>			
-----------	--	--	--

 Register Array 1

Value Table 2

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

→

<u>A2</u>			
-----------	--	--	--

 Register Array 2

Combine outputs from multiple arrays

Lookup Table

Match	<code>pkt.key == A</code>	<code>pkt.key == B</code>
Action	<code>bitmap = 111</code> <code>index = 0</code>	<code>bitmap = 110</code> <code>index = 1</code>

`pkt.value:` A0 A1 A2 B0 B1

Value Table 0

Match	<code>bitmap[0] == 1</code>
Action	<code>process_array_0 (index)</code>

→

0	1	2	3
<u>A0</u>	<u>B0</u>		

 Register Array 0

Value Table 1

Match	<code>bitmap[1] == 1</code>
Action	<code>process_array_1 (index)</code>

→

<u>A1</u>	<u>B1</u>		
-----------	-----------	--	--

 Register Array 1

Value Table 2

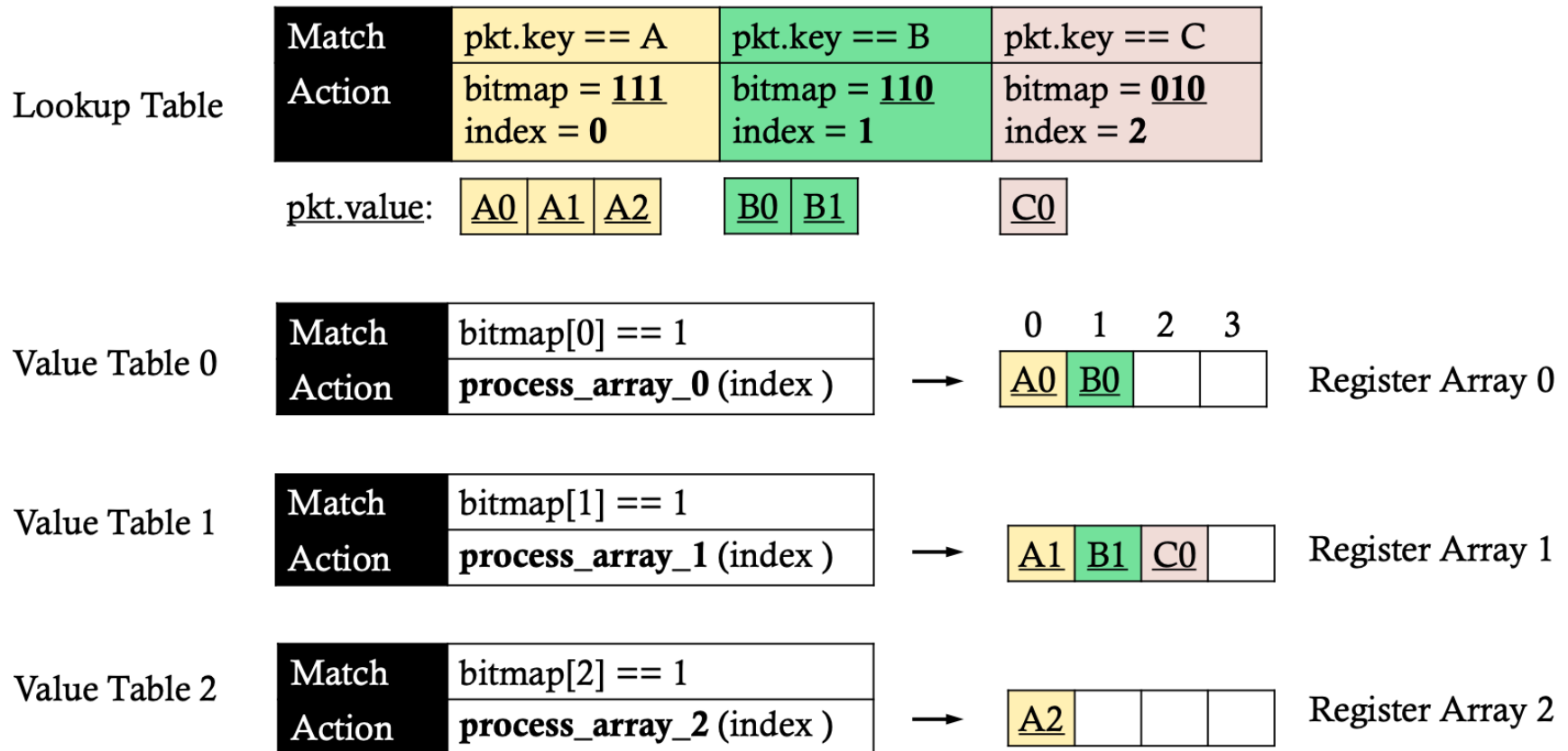
Match	<code>bitmap[2] == 1</code>
Action	<code>process_array_2 (index)</code>

→

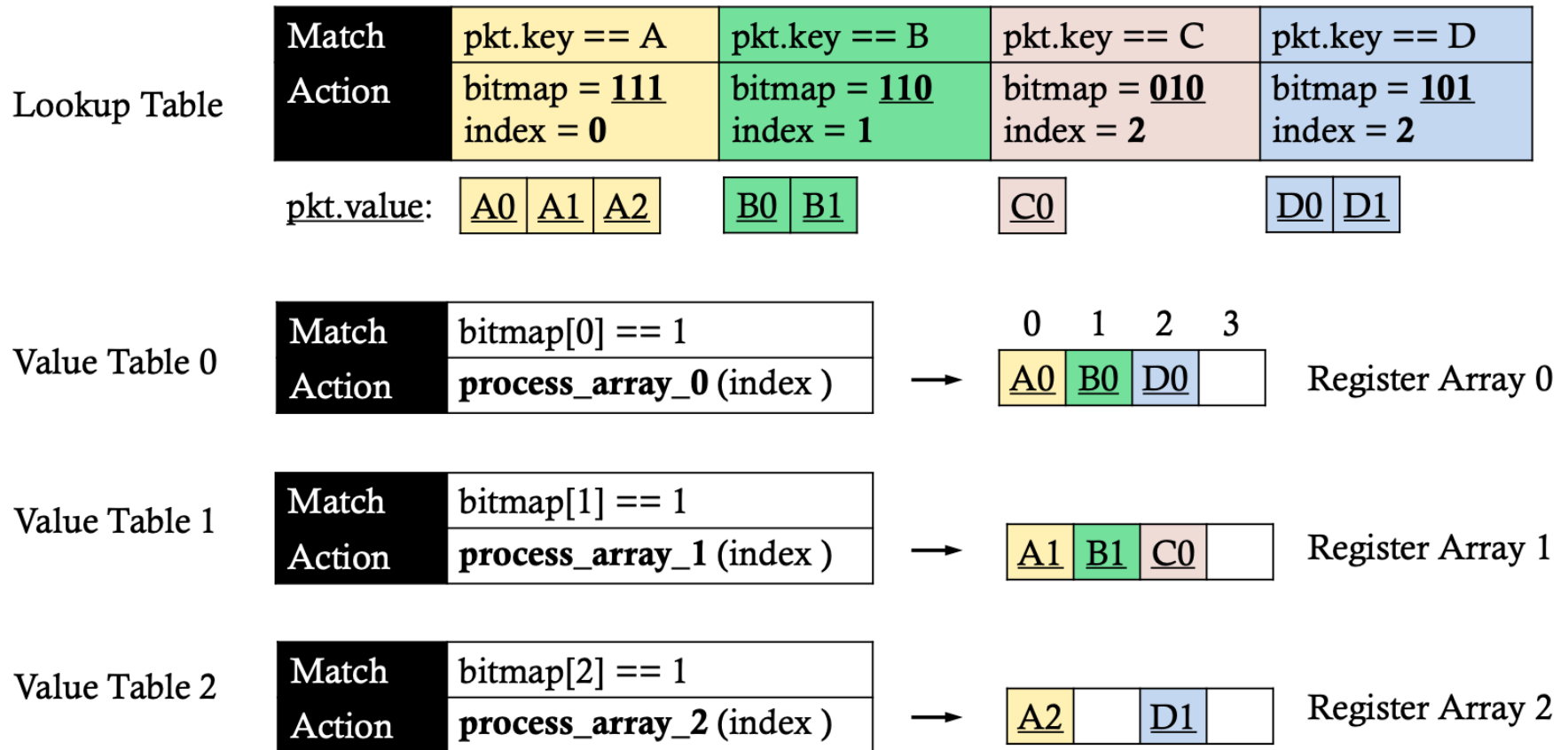
<u>A2</u>			
-----------	--	--	--

 Register Array 2

Combine outputs from multiple arrays



Combine outputs from multiple arrays

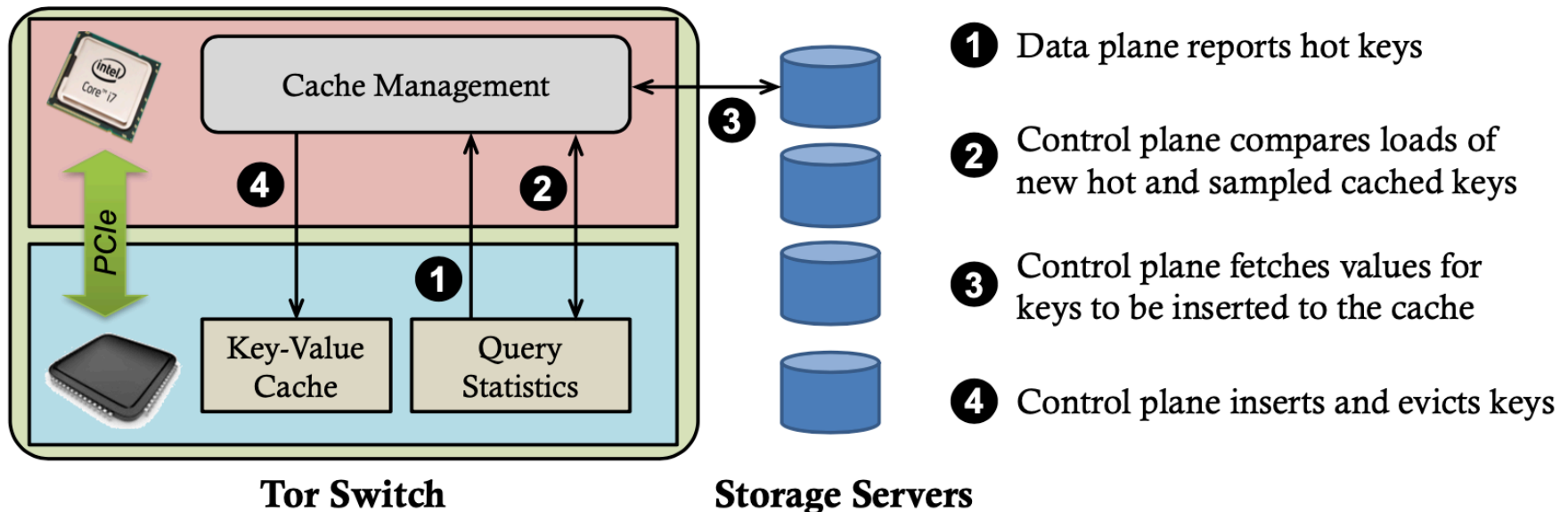


Key-value caching in network ASIC at line rate

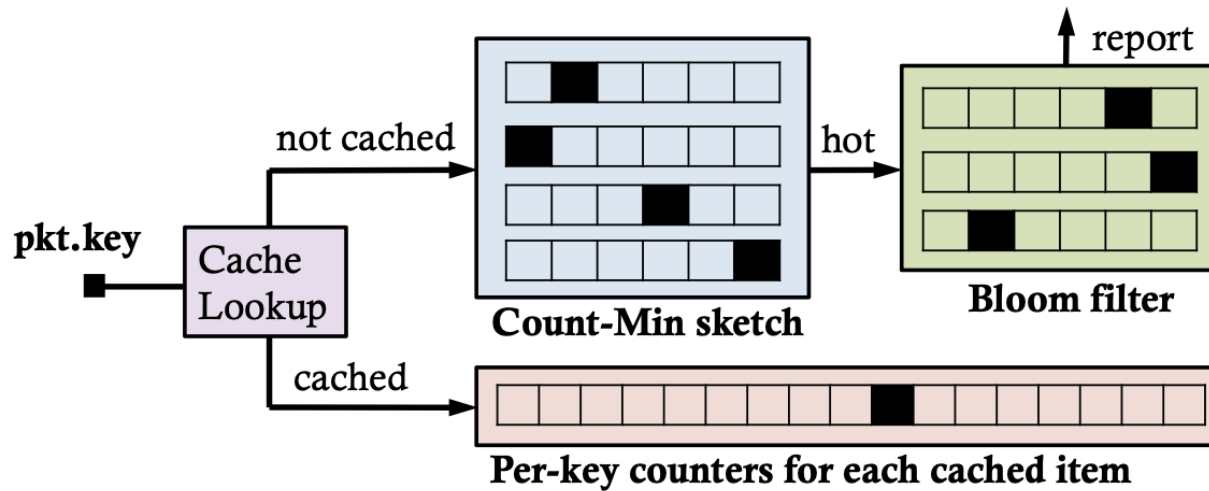
- ❑ How to identify application-level packet fields ?
- ❑ How to store and serve variable-length data ?
- ➔ ❑ How to efficiently keep the cache up-to-date ?

Cache insertion and eviction

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

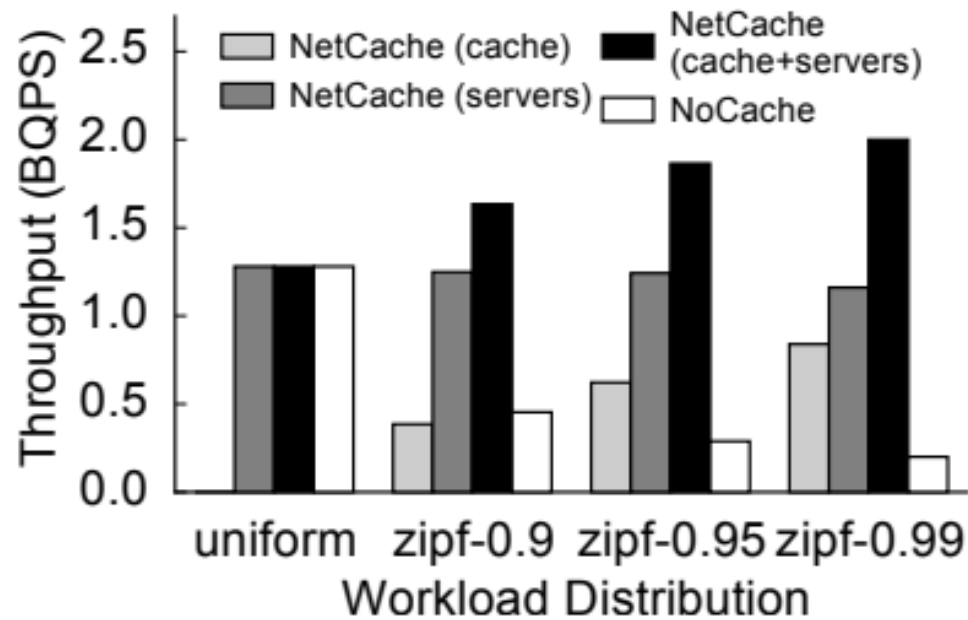


Query statistics in the data plane



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

Evaluation



Is this a good usecase of
programmable dataplanes?

What are the limitations?

What could have been an alternate strategy?

Alternative Design

- Replicate popular keys on other servers.
- Maintain key access statistics in the servers.
- Switches maintain rules on which key is replicated in which servers.
- [Pegasus, OSDI'20], [SwitchKV,, NSDI'16]

SailFish

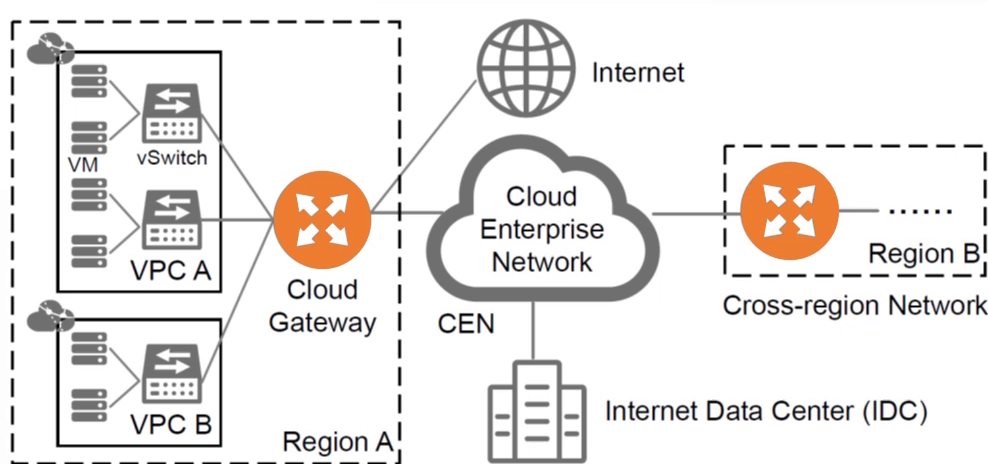
Slides borrowed from the authors' SIGCOMM'21 presentation

Sailfish: Accelerating Cloud-Scale Multi-Tenant Multi-Service Gateways with Programmable Switches

Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, Shunmin Zhu



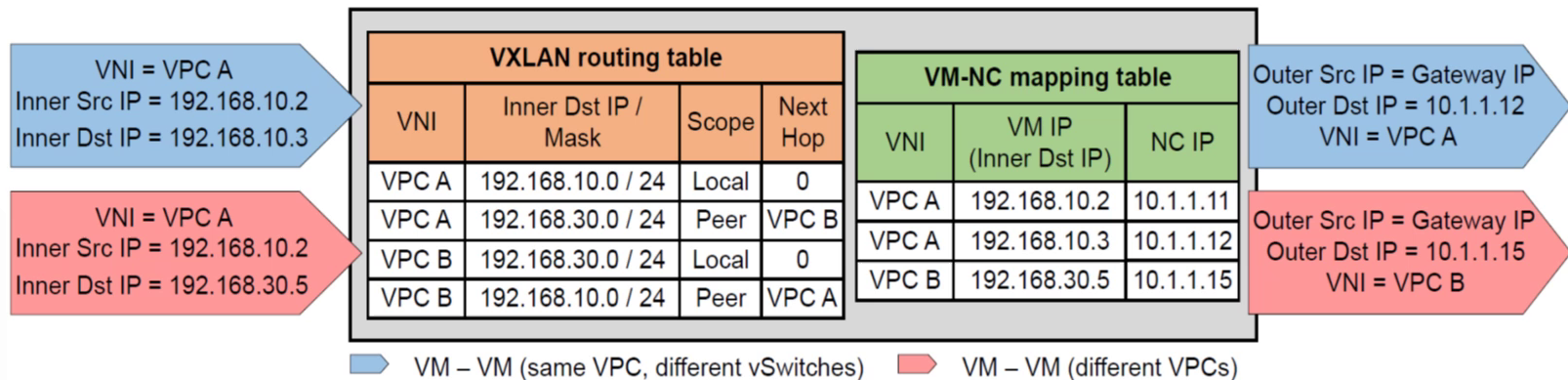
Where is the Cloud Gateway?



Cloud gateway is the **central hub** of east-west traffic (VM-VM), south-north traffic (VM-Internet) and IDC/cross-region traffic (VM-IDC/Cross-region) of the entire cloud

Traffic Routes	Cloud Service Examples
VM-VM (same VPC)	Message passing in distributed computing
VM-VM (different VPCs)	Communication between two VPCs in the same region
VM-IDC	Tenant pulls the computation results from cloud to his office
IDC-VM	Tenant logs into his VM from his office
VM-Internet	Tenant crawls web pages from the Internet
Internet-VM	Tenant logs into his VM from home
VM-Cross-region	Communication between one tenant from China and another tenant from USA

Packet Forwarding at the Cloud Gateway



Two basic forwarding steps:

1. Find the right region/IDC/VPC scope containing the destination VM according to the inner DIP of the VXLAN-encapsulated packet (**VXLAN routing table**)
2. Find the exact physical server address where the destination VM is hosted according to the inner DIP of the VXLAN-encapsulated packet and the region/IDC/VPC scope obtained from the first step (**VM-NC mapping table**)

Limitations of Software Gateways

Limitations of Software Gateways

1. High CapEx and OpEx



~150 XGW-x86s
for 15Tbps traffic



Safety margin



~300 XGW-x86s
with 50% water level



Disaster tolerance

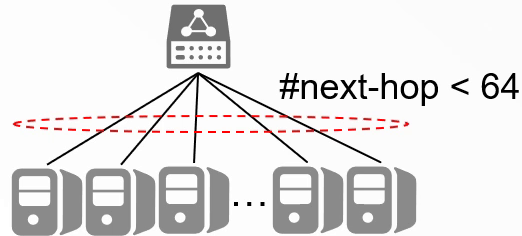


~600 XGW-x86s
for 1:1 backup

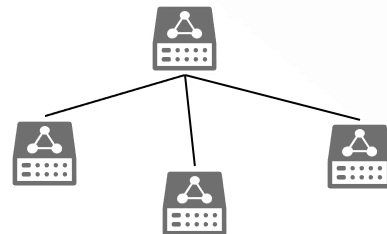


O(\$10M) acquisition cost
for a typical region

2. Load balancer's #next-hop limit

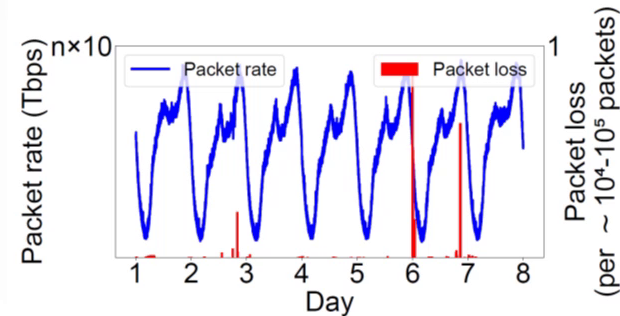
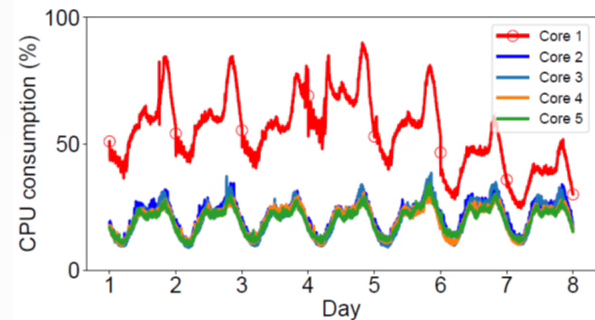


Split into multiple
gateway clusters



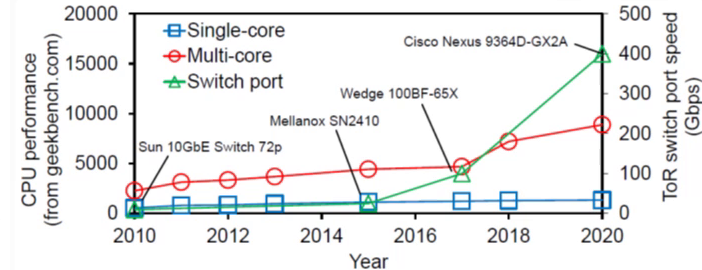
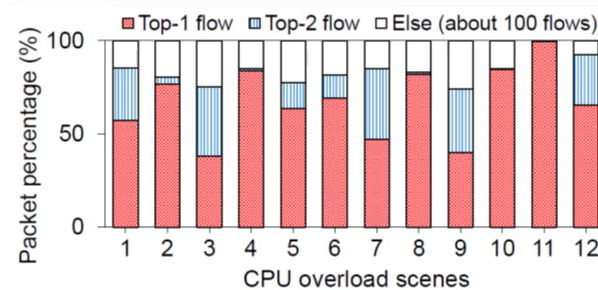
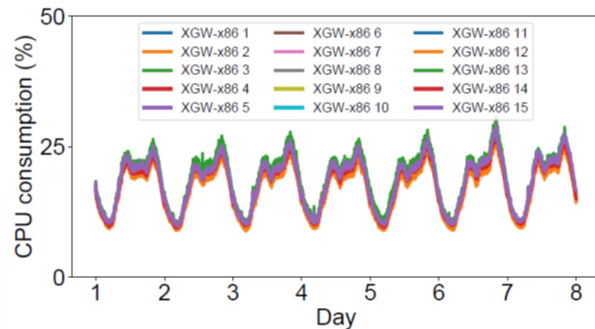
Additional maintenance
overhead

3. CPU overload due to traffic bursts



10^{-5} - 10^{-4} packet loss in Day 6,
user satisfactory affected

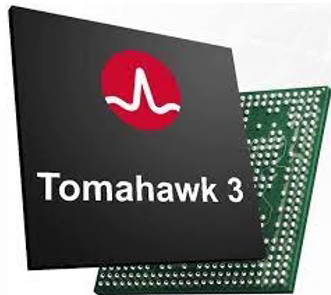
Why Horizontal Scaling Does not Work?



1. Load is perfectly balanced among gateways
2. Load is unequally distributed among CPU cores
1. Top-1 and top-2 flows dominate the traffic during CPU core overload
2. Flow-based hashing in the run-to-completion model cannot achieve perfect load balancing
3. Packet-based hashing can achieve perfect load balancing but needs dedicated hardware for fast packet reordering
1. ToR switch port speed: 40x
2. Multi-core performance: 4x
3. Single-core performance: 2.5x
4. Traffic growth >> Moore's law > single-core improvement
5. XGW-x86 will be challenged by more and more heavy hitters

Turn to Hardware Gateways

1. Fixed-function ASICs



1. Performant, affordable, energy-efficient
2. Lack of programmability
3. Time-to-market cycle is too long for service-driven cloud networks

2. FPGAs



1. Programmable but with limited performance
2. More suitable for building smartNICs at the end hosts rather than cloud gateways at the central hub

3. Programmable ASICs



1. Performant & programmable
2. Similar unit price as XGW-x86
3. More options today (e.g., Tofino, Silicon One, Trident)

Technical Challenges Using Programmable ASICs

Technical Challenges Using Programmable ASICs

1. Larger table entry numbers with longer table entries

$O(1M)$

VXLAN routing table				VM-NC mapping table		
VNI	Inner Dst IP / Mask	Scope	Next Hop	VNI	VM IP (Inner Dst IP)	NC IP
VPC A	192.168.10.0 / 24	Local	0	VPC A	192.168.10.2	10.1.1.11
VPC A	192.168.30.0 / 24	Peer	VPC B	VPC A	192.168.10.3	10.1.1.12
VPC B	192.168.30.0 / 24	Local	0	VPC B	192.168.30.5	10.1.1.15
VPC B	192.168.10.0 / 24	Peer	VPC A			

$O(1M)$

2. Limited total on-chip memory with fixed SRAM/TCAM ratio

$O(10MB)$ SRAM and TCAM; TCAM is much less

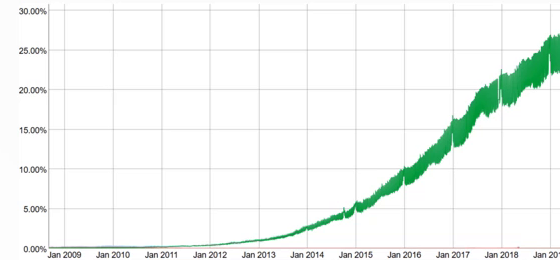
Table name	Key	Match type	IP type	Key length (bit)	Match SRAM	TCAM
VXLAN routing table	VNI, Inner Dst IP	LPM	IPv4	24 + 32		311%
			IPv6	24 + 128		622%
VM-NC mapping table	VNI, Inner Dst IP	EXACT	IPv4	24 + 32	58%	
			IPv6	24 + 128	233%	
Sum (75% IPv4, 25% IPv6)					102%	388.75%

Overflow

3. Handling diverse cloud services

Routing tables, ACL rules, meters, SNAT tables, DNAT tables, etc

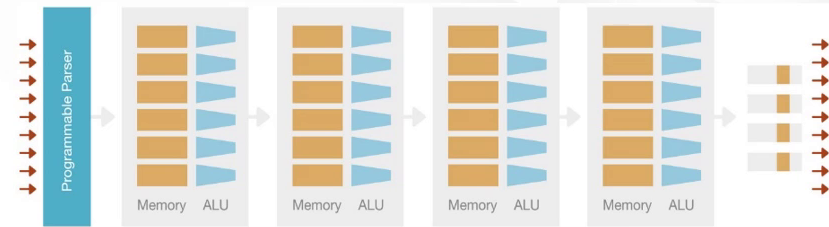
4. Processing yearly growing IPv6 traffic



Longer IPv6 entries

Dynamic allocation of memory for growing IPv4 and IPv6 entries

5. Table placement on noncontinuous memory

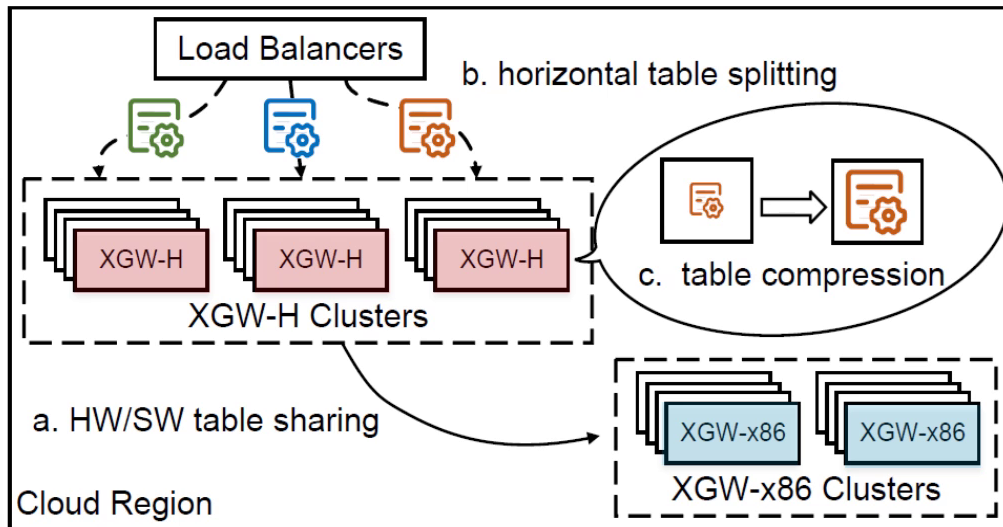


Local memory of each pipeline stage cannot be accessed from other stages

Metadata transferring between ingress Pipe and egress Pipe is not allowed

Sailfish Design Overview

Sailfish Design Overview

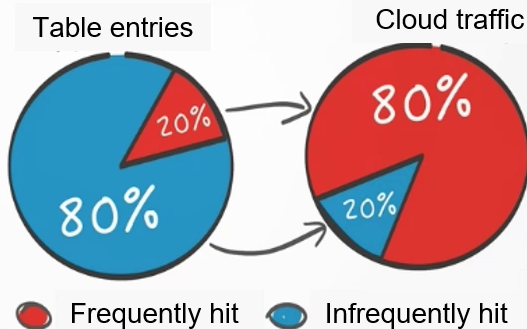


- a. Hardware and software co-design for table sharing
- b. Horizontal table splitting among XGW-H clusters
- c. Single-node table compression on XGW-H

A multi-pronged approach to solve the problem of insufficient on-chip memories

Hardware and Software Co-Design

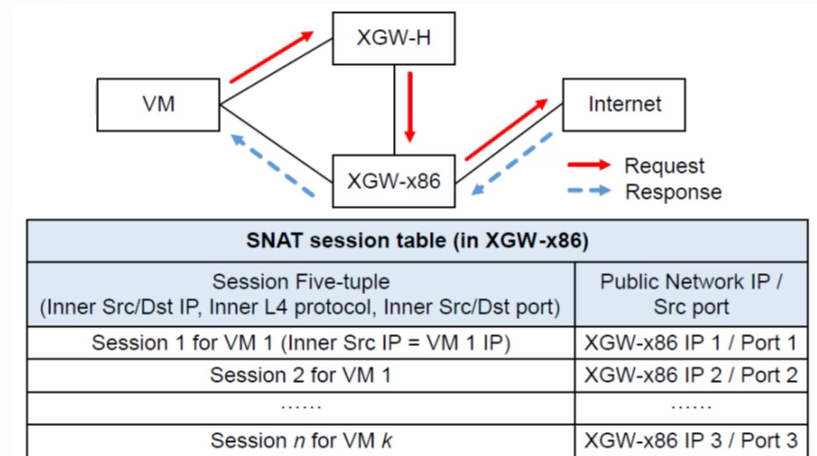
80-20 rule



Design principles:

1. XGW-H stores a few key tables for basic cloud services frequently hit by the majority of traffic
2. XGW-x86 maintains volatile tables or very large stateful tables that cannot be fit in XGW-H
3. Unstable new services with less traffic are carried by XGW-x86 while mature services with heavy and stable traffic are turned over to XGW-H

One example: SNAT

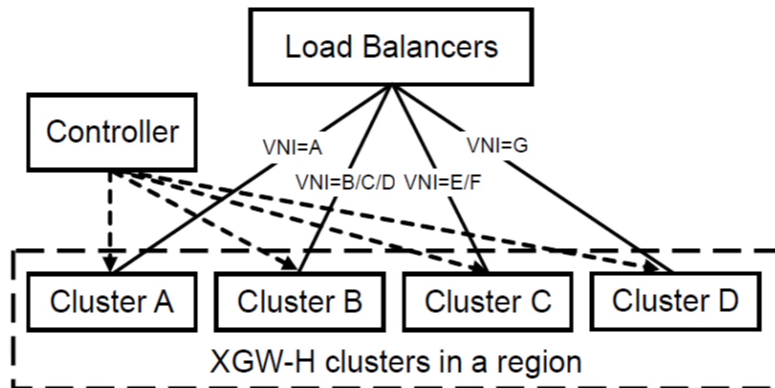


SNAT table stores per-flow states of $O(100M)$

SNAT table is kept in XGW-x86

**Hundreds of XGW-x86s
=> 10 XGW-Hs + 4 XGW-x86s
(saving more than 90% CapEx)**

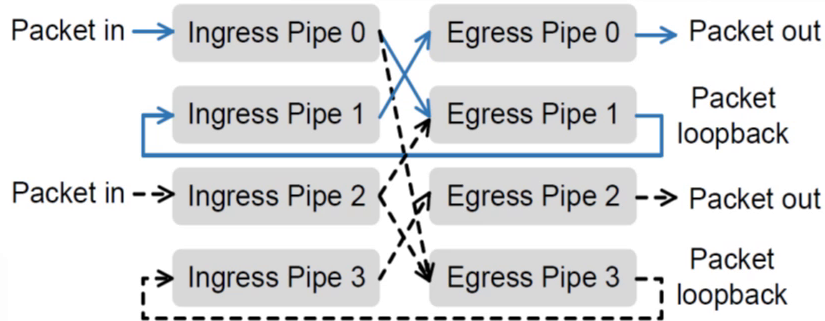
Horizontal Table Splitting Among Clusters



Load balancers distribute traffic into different XGW-H clusters according to the **VNI** tag (i.e., tenant ID)

- What is horizontal table splitting?
 - Each gateway cluster in a region maintains all the tables but a portion of entries from each table based on the tenant ID
- Horizontal splitting vs vertical splitting
 - Horizontal splitting: adding a new tenant into the cloud will only cause gateway updates within one cluster (failure isolation)
 - Vertical splitting: splitting different tables of all entries into different clusters; adding a new tenant will affect all the clusters
- Horizontal splitting vs flow-based hashing
 - Horizontal splitting: tractable load balancing
 - Flow-based hashing: blind load balancing

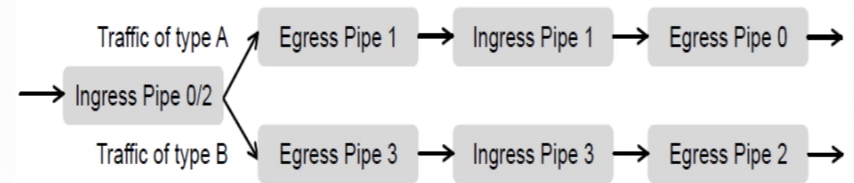
Pipeline-Aware Memory Optimizations



- Pipeline folding

- Original: pipeline x 4
=> throughput x 4
=> memory in each pipeline 25%
- After folding: pipeline x 2
=> throughput x 2
=> memory in each pipeline 50%

Trade-off between memory and throughput



- Table splitting between pipelines

- After pipeline folding, Egress Pipe 1 and Egress Pipe 3 contain the same table entries
- To reduce the table redundancy, we can use hashing to further divide the traffic into different pipes
- Less table redundancy => more available memory

IPv4/IPv6 Table Pooling

IPv4 table



IPv6 table

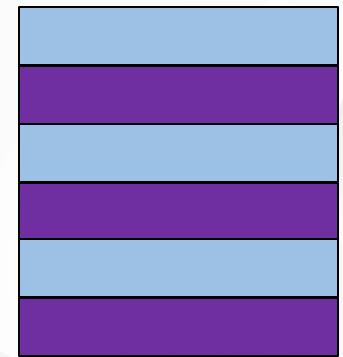


Exact match
(VM-NC mapping table)



+ Conflicting
table

Longest prefix match
(VXLAN routing table)



- Separate IPv4/IPv6 tables

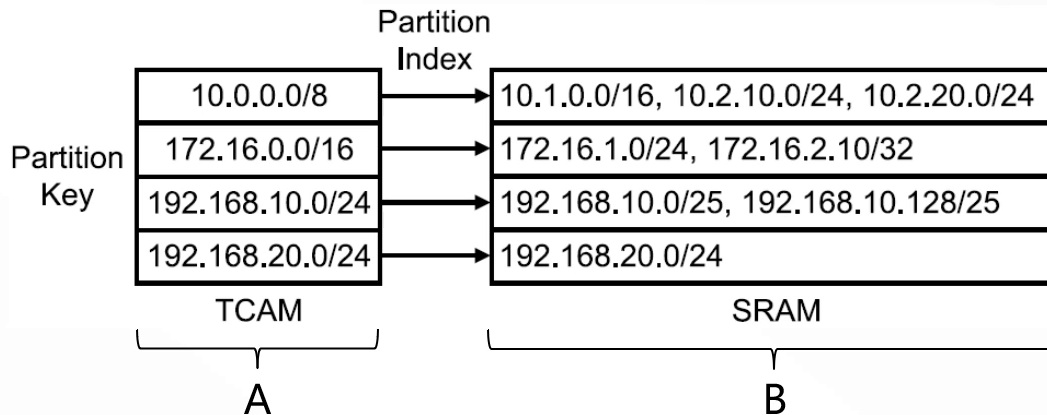
- Pros
 - ▶ Update separately
 - ▶ Expand separately
- Cons
 - ▶ Inefficient memory usage due to memory fragments

- IPv4/IPv6 table pooling strategies

- VM-NC mapping table:
 - ▶ 128-bit IPv6 hashed to 32-bit
 - ▶ Conflicting table for hash collision resolution
- VXLAN routing table:
 - ▶ 32-bit IPv4 expanded to 128-bit

TCAM Conservation for Large FIBs

When programmable ASIC chips are taped out, the SRAM/TCAM ratio is fixed (TCAM < SRAM), but the table lookup requirements of LPM or exact match change over time



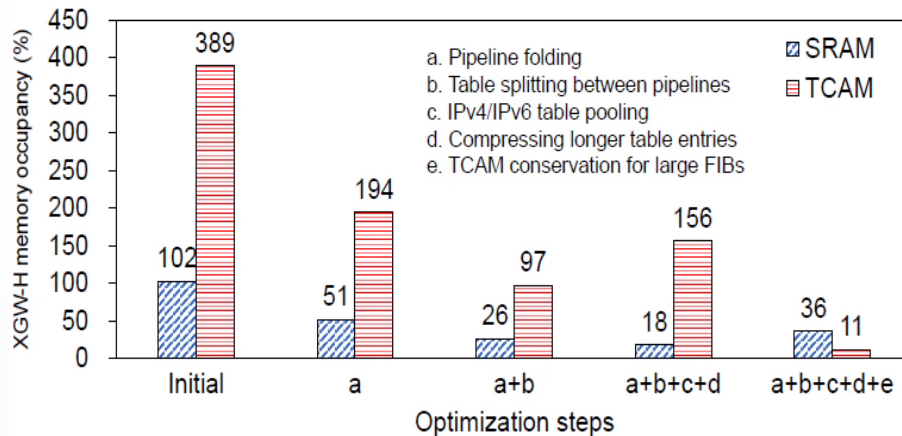
ALPM

Henry Wang. 2019. Algorithmic longest prefix matching in programmable switch. (Dec. 17 2019). US Patent 10,511,532.

- Trade-off between lookup efficiency and TCAM occupation
 - Larger A => higher lookup efficiency, larger TCAM usage
 - Larger B => lower lookup efficiency, more SRAM usage, much reduced TCAM usage

Memory Usage After Step-by-Step Optimizations

Memory usage after step-by-step compression



Memory occupancy of VXLAN routing table & VM-NC mapping table

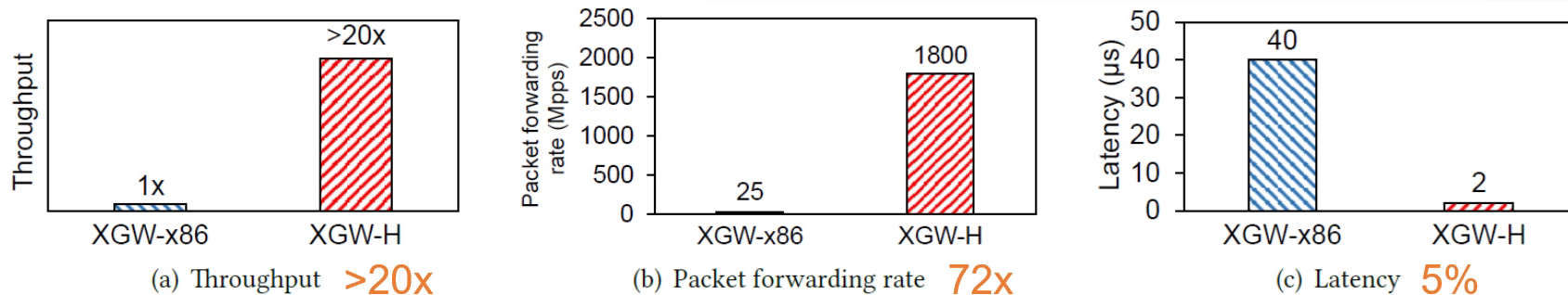
Table name	Match SRAM	TCAM
VXLAN routing table	18%	11%
VM-NC mapping table	18%	
Sum	36%	11%

Overall memory consumption

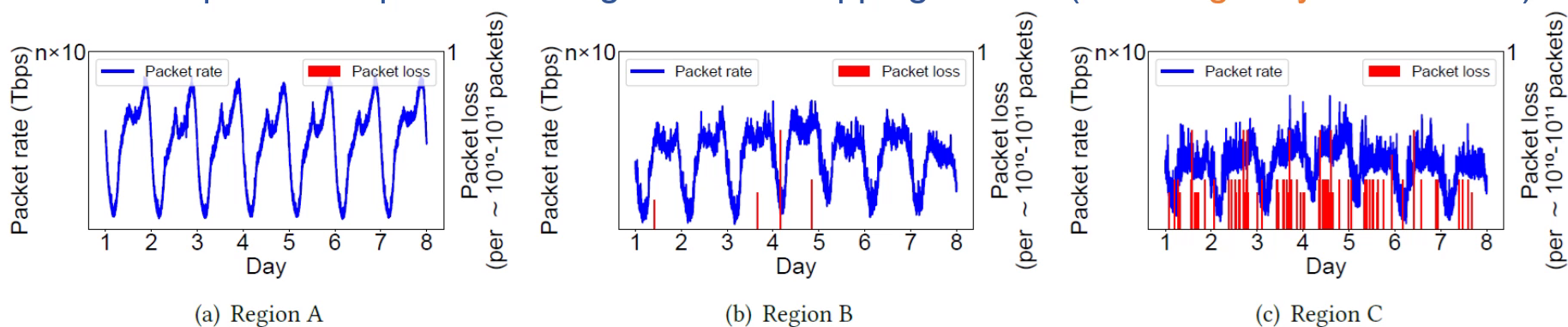
Pipeline	Match SRAM	TCAM
Pipeline 0/2	70%	41%
Pipeline 1/3	68%	22%
Sum	69%	32%

Sailfish Performance

XGW-H vs XGW-x86



Sailfish's packet drop rate in a large on-line shopping festival (from originally 10^{-4} to 10^{-10})



Conclusion

- We investigate the CPU overload problem in cloud gateways
- We address the problem by accelerating the cloud gateways with programmable ASICs with a much larger safety margin
- To resolve the memory shortage issue, we propose techniques such as hardware/software co-design, horizontal table splitting among gateway clusters and single-node table compression
- Sailfish is the first P4-based cloud gateway in the industry

Is this a good usecase of
programmable dataplanes?

What are the limitations?

Alternate strategy?

P4 usecases (in-network mechanisms)

- Network-level load balancing:
 - HULA: Scalable Load Balancing Using Programmable Data Planes, SOSR'16
- Congestion control:
 - Evaluating the Power of Flexible Packet Processing for Network Resource Allocation, NSDI'17
 - *Support RCP and XCP on programmable switches*
 - HPCC: High Precision Congestion Control, SIGCOMM'19
 - *Obtain precise link information for congestion control*
- A new protocols for more efficient L2 switching
 - The Deforestation of L2, SIGCOMM'16
- Source-routed multicast
 - Elmo: SIGCOMM'19
- Network monitoring
 - Beaucoup: SIGCOMM'20
 - Marple: SIGCOMM'17

App-level P4 usecases

- NetChain: in-network key-value store (NSDI'18).
- NetLock: Switching support to manage locks (SIGCOMM'20).
- NetPaxos: implement Paxos on programmable switches (SOSR'15)
- DAEIT: In-network data aggregation (SOCC'17)
- NoPaxos (OSDI'16), Eris (SOSP'17): in-network primitives for distributed protocols.
- SilkRoad: L4 load balancer (SIGCOMM'17)
- Robot arm control (NSDI'22)
-