Use cases of Programmable Dataplane (P4)

Part 2

ECE/CS598HPN

Radhika Mittal

Which paper(s) did you read?

NetCache

Slides borrowed from the authors' SOSP'17 presentation

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

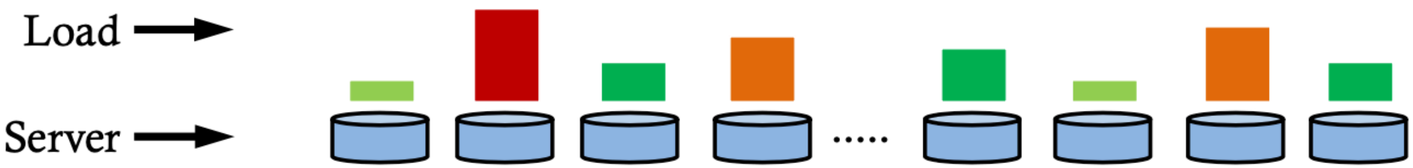
- **Store, retrieve, manage key-value objects**
  - Critical building block for large-scale cloud services

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

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

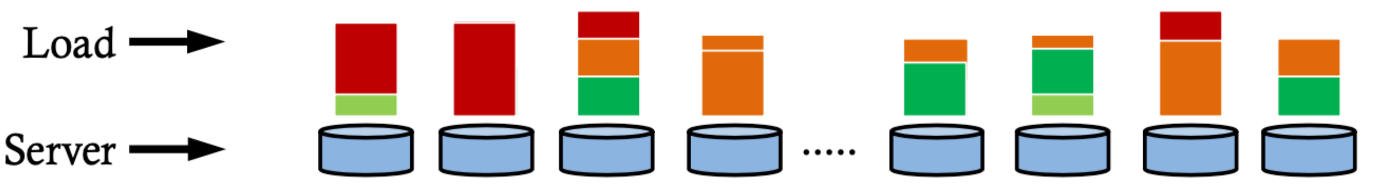
Key challenge: highly-skewed and rapidly-changing workloads

low throughput & **high tail latency**



Key challenge: highly-skewed and rapidly-changing workloads

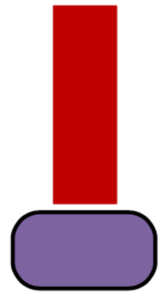
low throughput & **high tail latency**



Q: How to provide effective dynamic load balancing?

Opportunity: fast, <u>small</u> cache can ensure load balancing

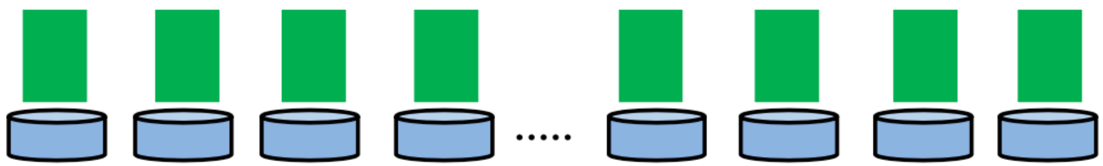
[B. Fan et al. **SoCC'11**, X. Li et al. **NSDI'16**]



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

E.g., 10,000 hot objects

N: # of servers

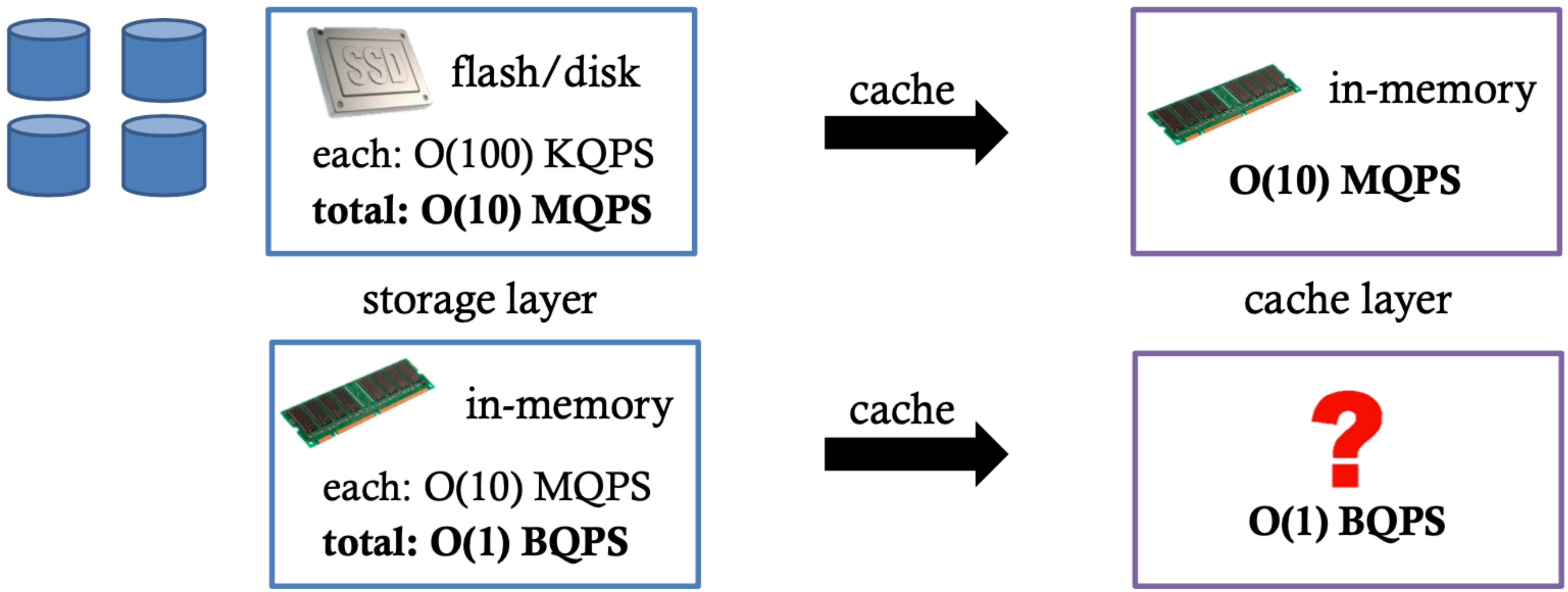


E.g., 100 backends with 100 billions items

Requirement: cache throughput ≥ backend aggregate throughput

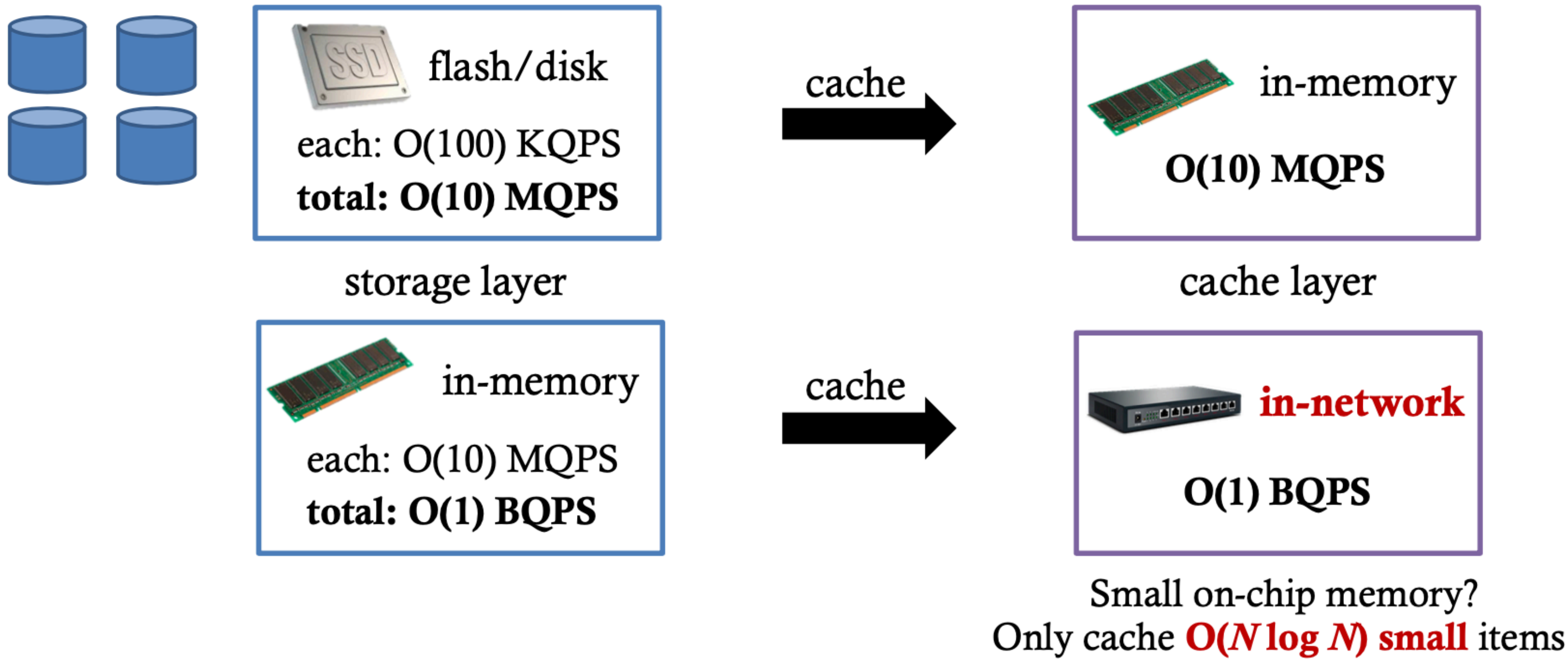
NetCache: towards billions QPS key-value storage rack

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

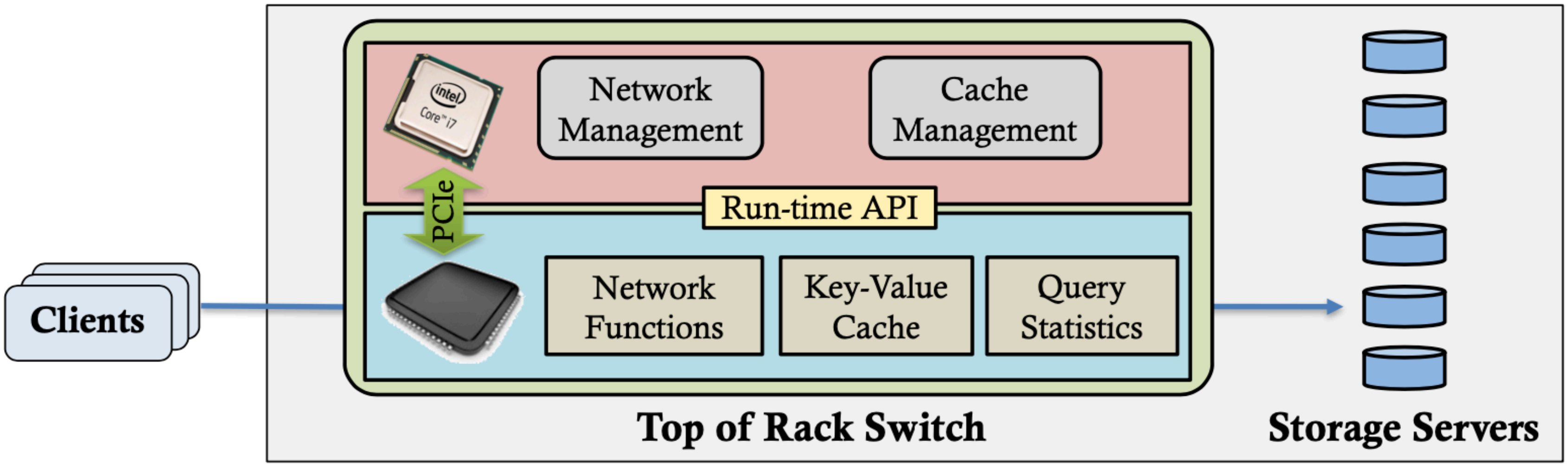


NetCache: towards billions QPS key-value storage rack

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



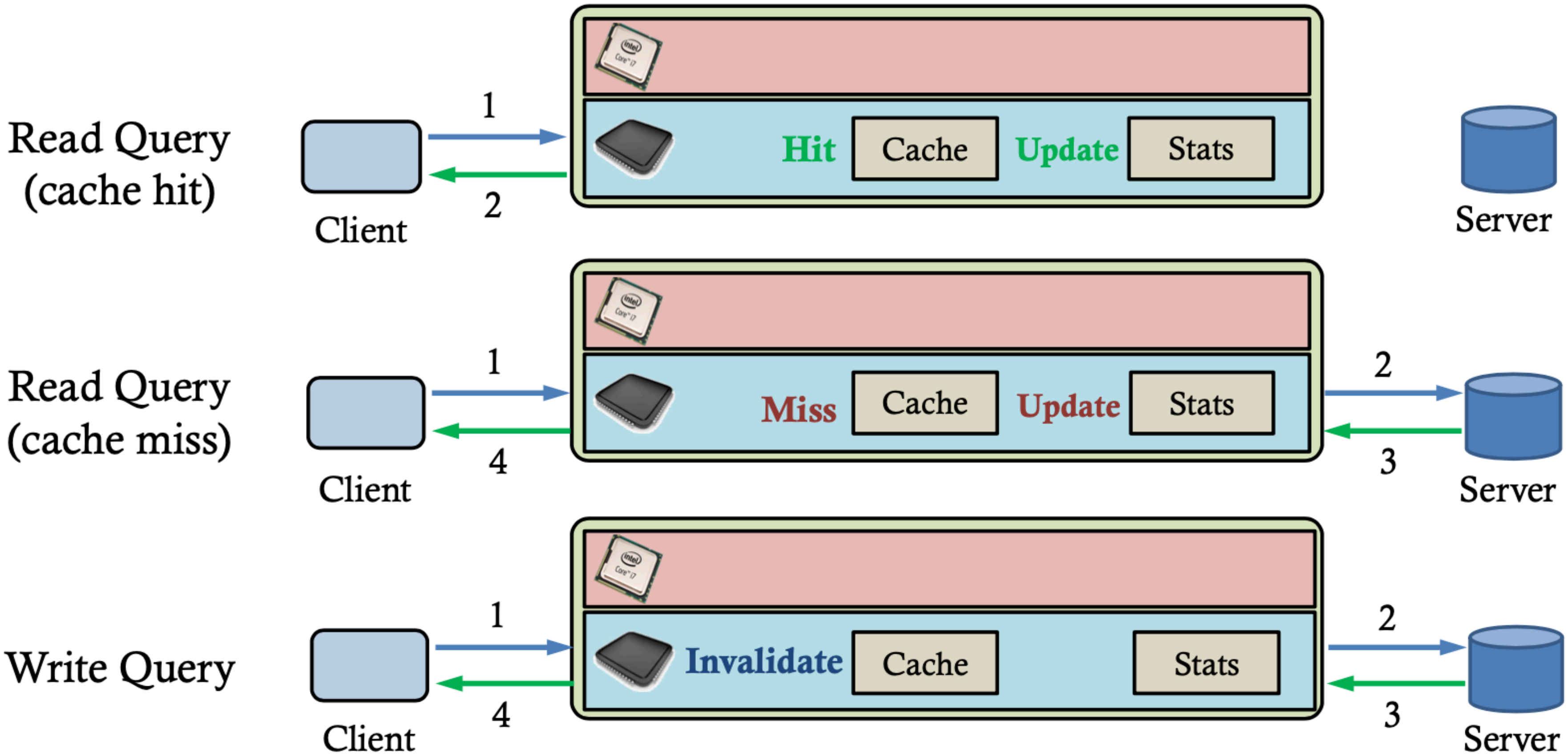
NetCache rack-scale architecture



- **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



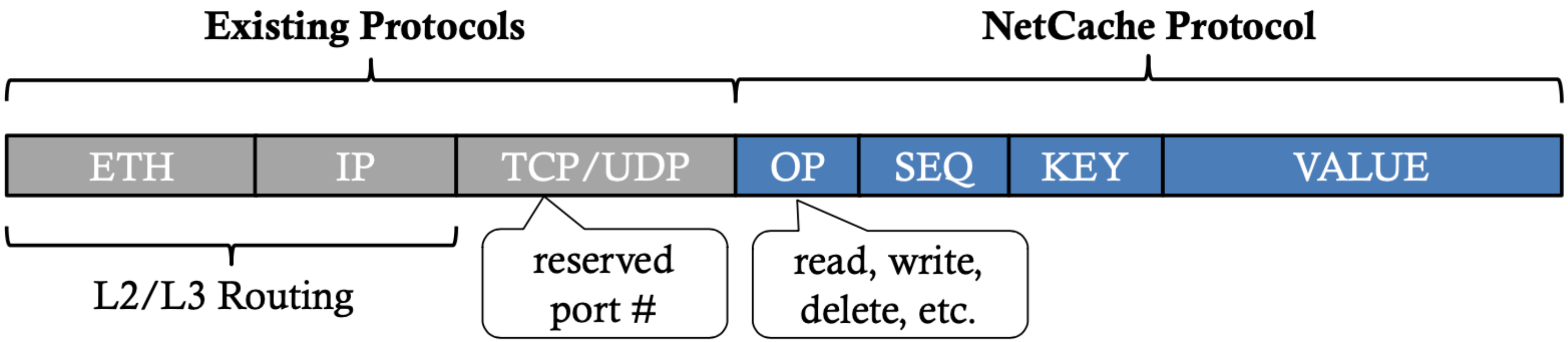
Key-value caching in network ASIC at line rate ?!

- 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



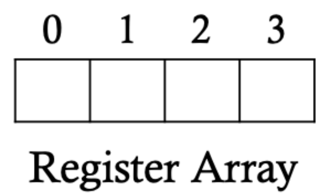
- Application-layer protocol: compatible with existing L2-L4 layers
- Only the top of rack switch needs to parse NetCache fields

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

pkt.value: A B

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

0	1	2	3
A	B		

Register Array

Combine outputs from multiple arrays

Bitmap indicates arrays that store the key's value

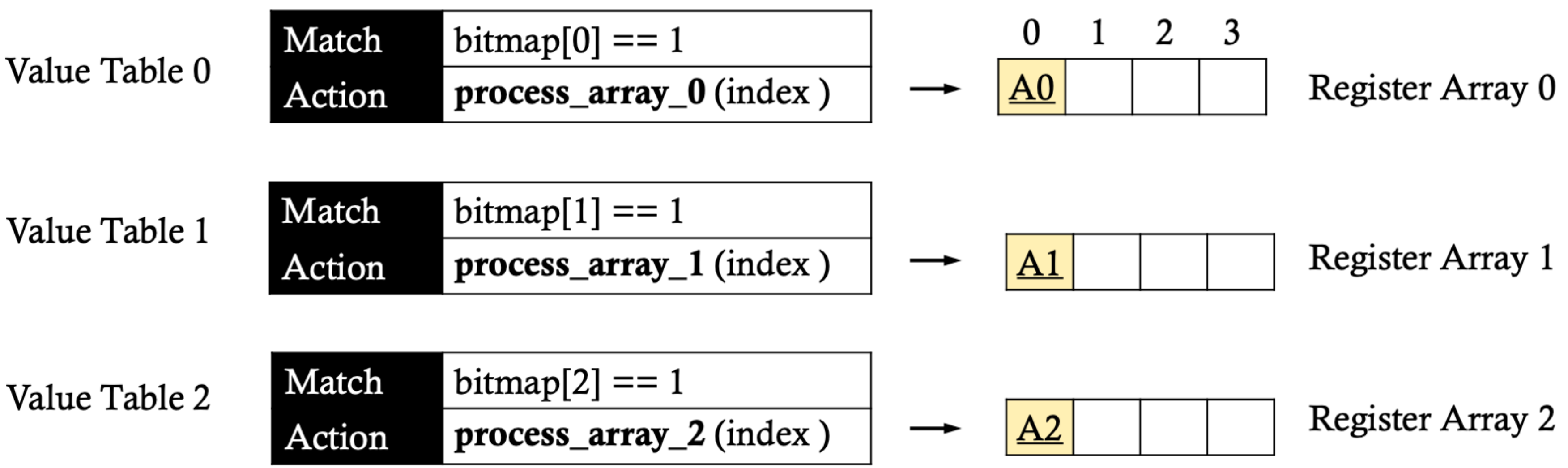
Index indicates slots in the arrays to get the value

Minimal hardware resource overhead

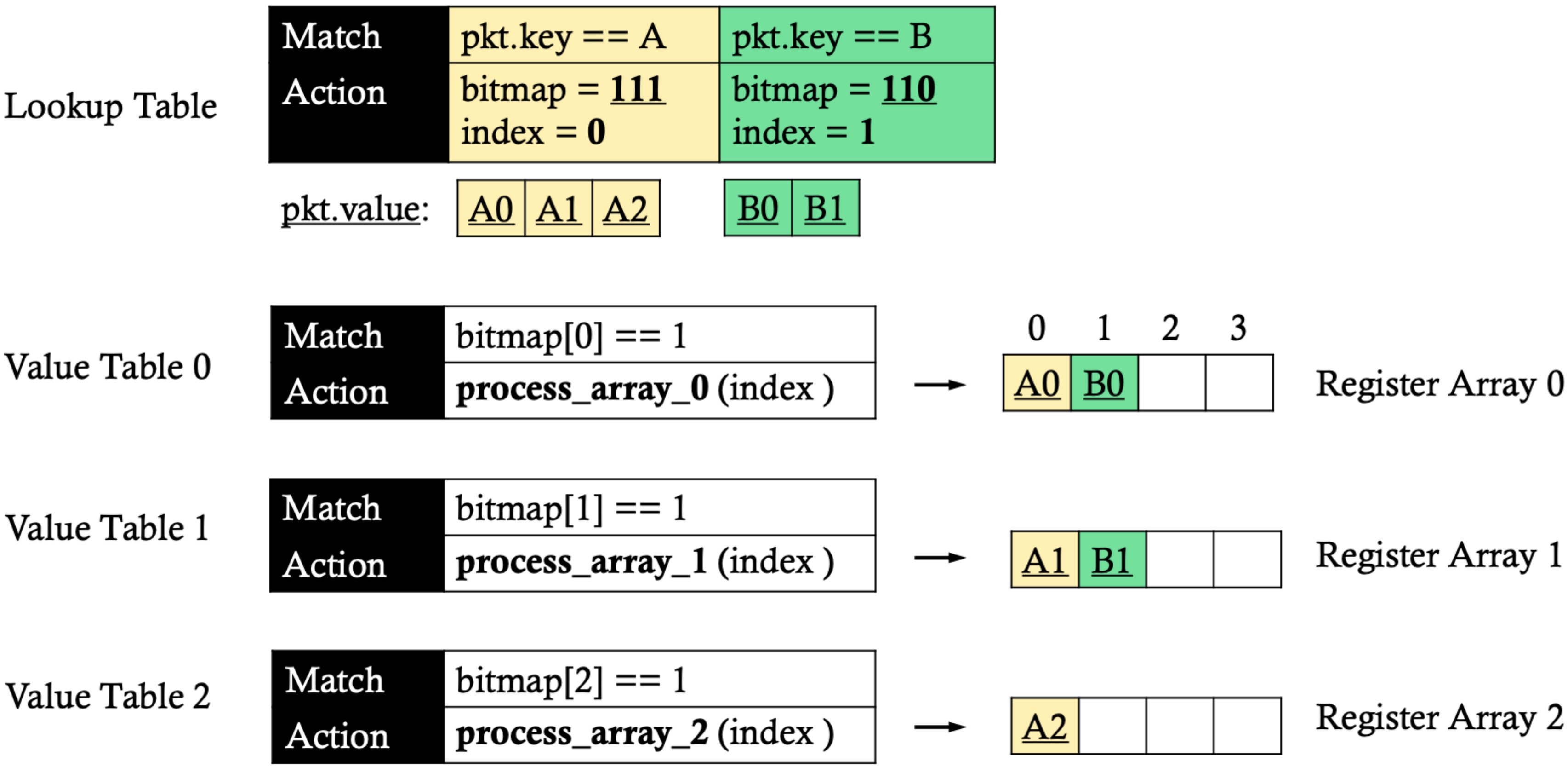
Lookup Table

Match	pkt.key == A
Action	bitmap = **111** index = **0**

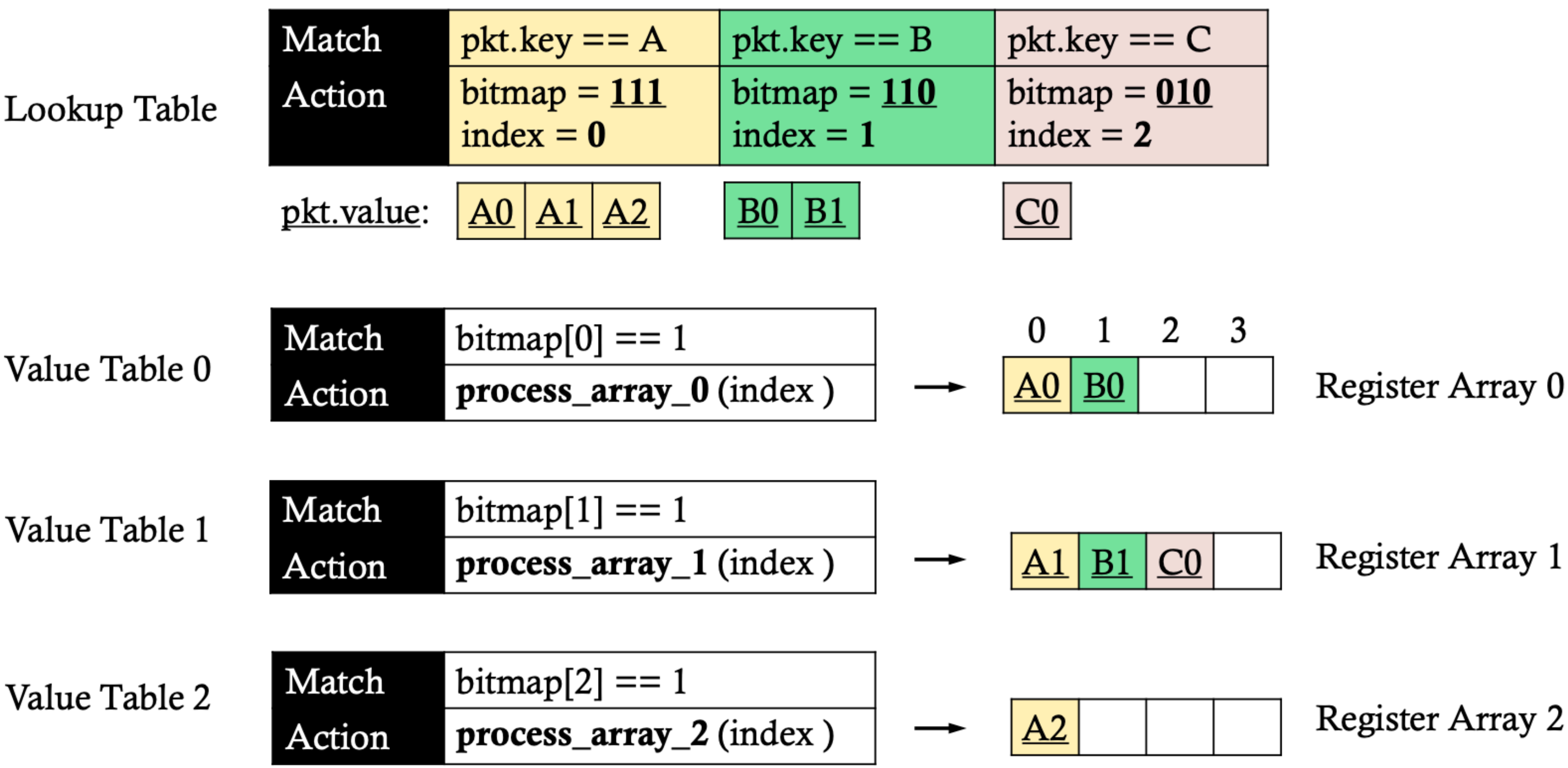
pkt.value: | A0 | A1 | A2 |



Combine outputs from multiple arrays



Combine outputs from multiple arrays



Combine outputs from multiple arrays

Lookup Table

Match	pkt.key == A	pkt.key == B	pkt.key == C	pkt.key == D
Action	bitmap = **111** index = **0**	bitmap = **110** index = **1**	bitmap = **010** index = **2**	bitmap = **101** index = **2**

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

Value Table 0

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

→ Register Array 0

0	1	2	3
A0	B0	D0	

Value Table 1

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

→ Register Array 1

A1	B1	C0	

Value Table 2

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

→ Register Array 2

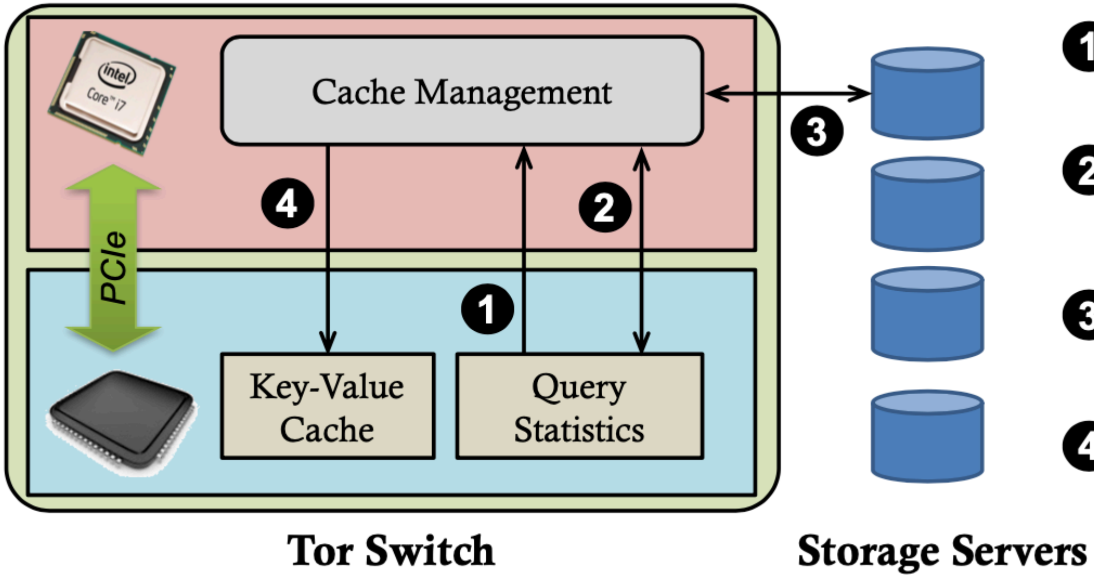
A2		D1	

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 ?

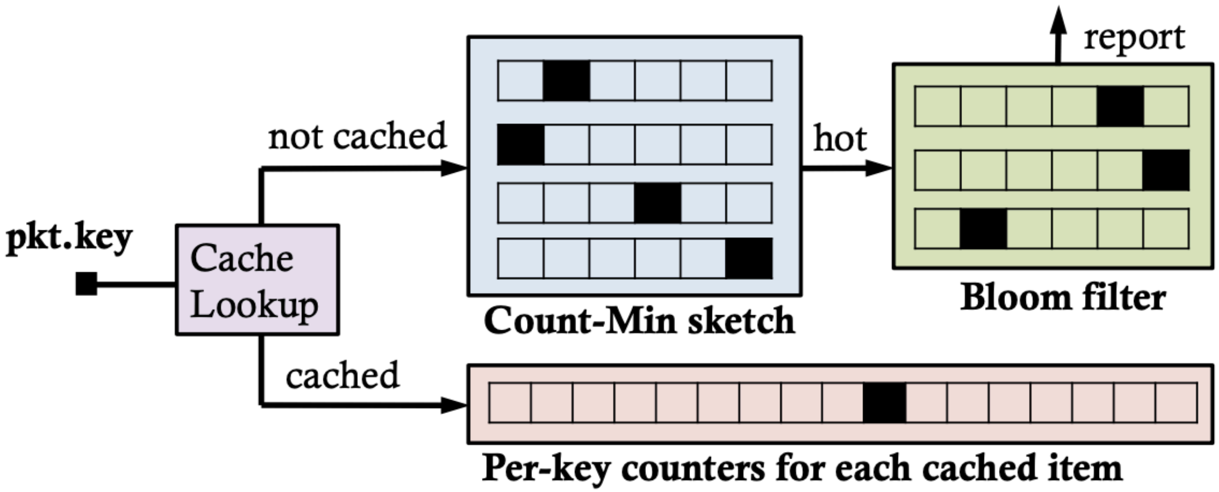
Cache insertion and eviction

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



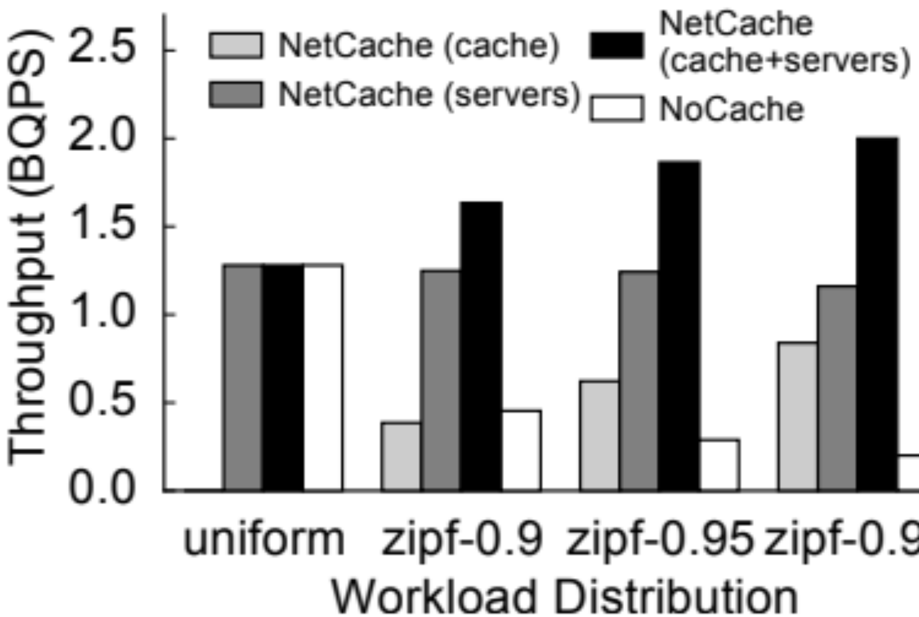
1. Data plane reports hot keys
2. Control plane compares loads of new hot and sampled cached keys
3. Control plane fetches values for keys to be inserted to the cache
4. Control plane inserts and evicts keys

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



NetCache (cache)
NetCache (servers)
NetCache (cache+servers)
NoCache
Throughput (BQPS)
2.5
2.0
1.5
1.0
0.5
0.0
uniform
zipf-0.9
zipf-0.95
zipf-0.99
Workload Distribution

Is this a good usecase of programmable dataplanes?

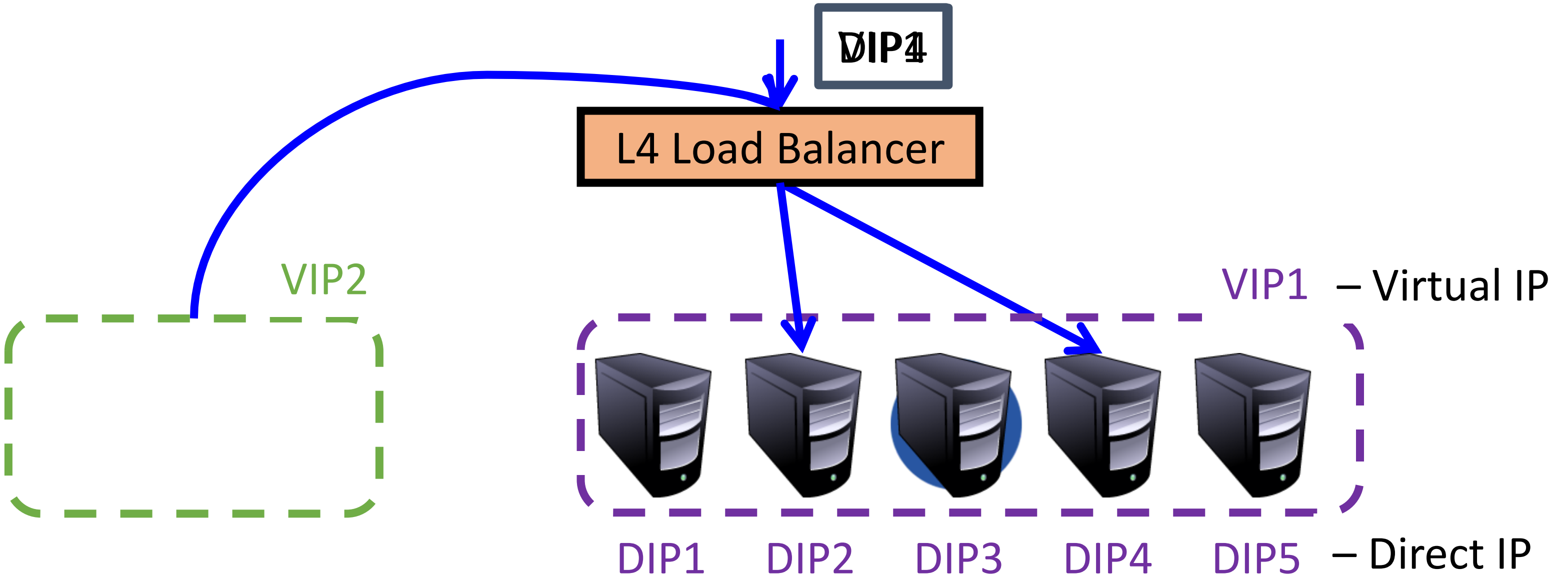
What are the limitations?

What could have been an alternate strategy?

SilkRoad

Slides borrowed from the authors' SIGCOMM'17 presentation

Layer-4 Load Balancing



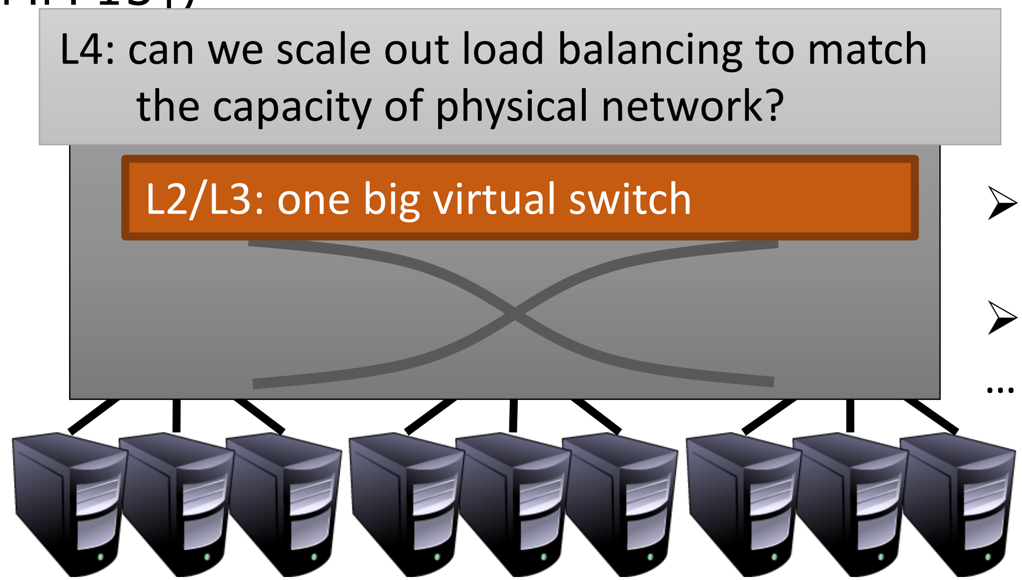
Layer-4 load balancing is a critical function

- handle both inbound and inter-service traffic
- >40%* of cloud traffic needs load balancing (Ananta [SIGCOMM'13])

Scale to traffic growth

Cloud traffic has a rapid growth

– doubling every year in Google, Facebook (Jupiter Rising [SIGCOMM’15])



- ➢ Multi-rooted topology
- ➢ Datacenter transport

…

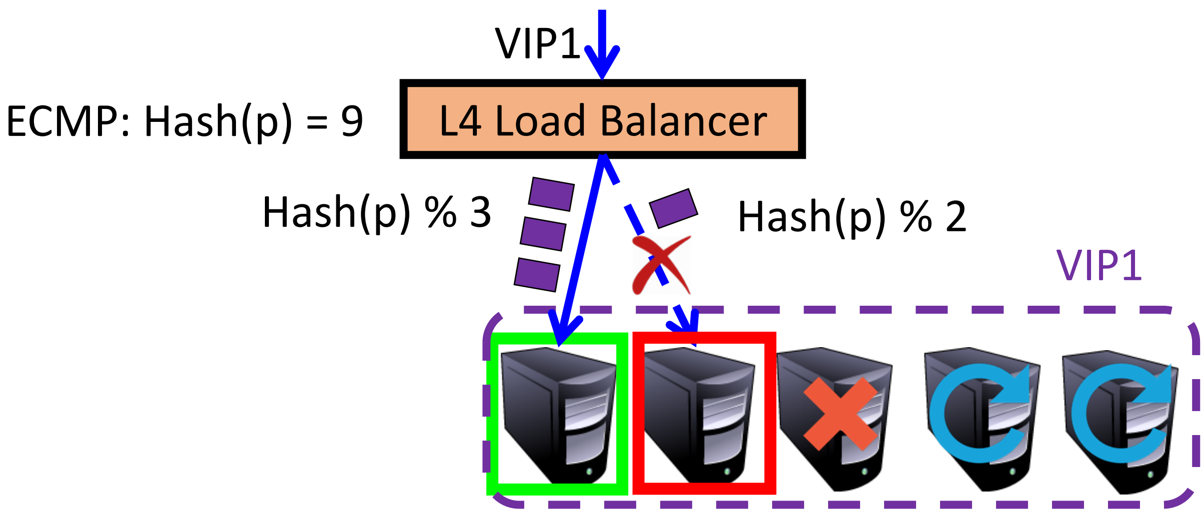
Frequent DIP pool updates

DIP pool updates

- failures, service expansion, service upgrade, etc.
- up to 100 updates per minute in a Facebook cluster

Hash function changes under DIP pool updates

- packets of a connection get to different DIPs
- connection is broken



Per-connection consistency (PCC)

Broken connections degrade the performance of cloud services

- tail latency, service level agreement, etc.

PCC: all the packets of a connection go to the same DIP

L4 load balancing needs connection states

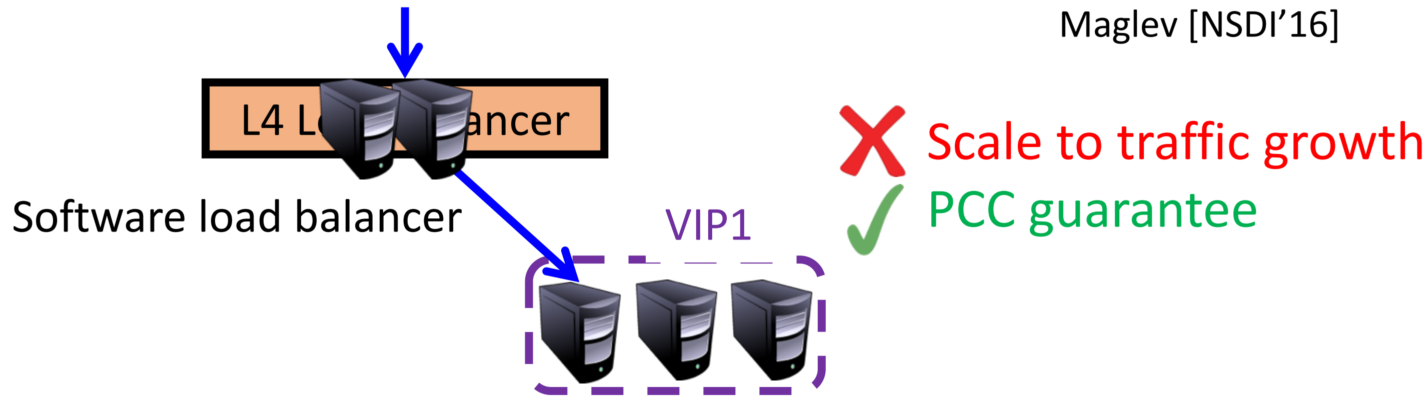
Design requirements

Scale to traffic growth

While ensuring PCC under frequent DIP pool updates

Existing solution 1: use software server

Ananta [SIGCOMM'13]
Maglev [NSDI'16]



High cost

- 1K servers (~4% of all servers) for a cloud with 10 Tbps

High latency and jitter

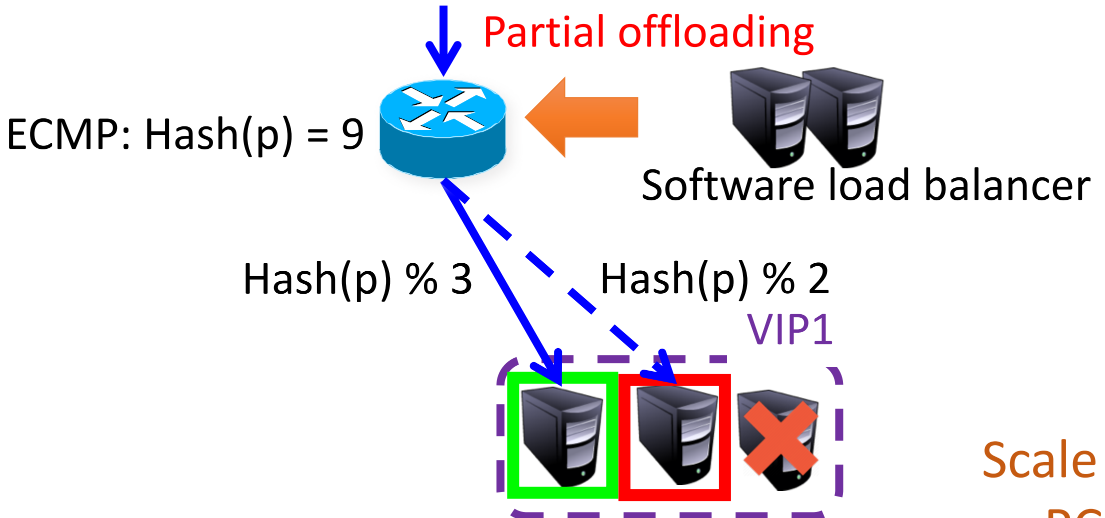
- add 50-300 μs delay for 10 Gbps in a server

Poor performance isolation

- one VIP under attack can affect other VIPs

Existing solution 2: partially offload to switches

Duet [SIGCOMM'14]
Rubik [ATC'15]



Scale to traffic growth or PCC guarantee

Hash function changes under DIP pool updates

- switch does not store connection states

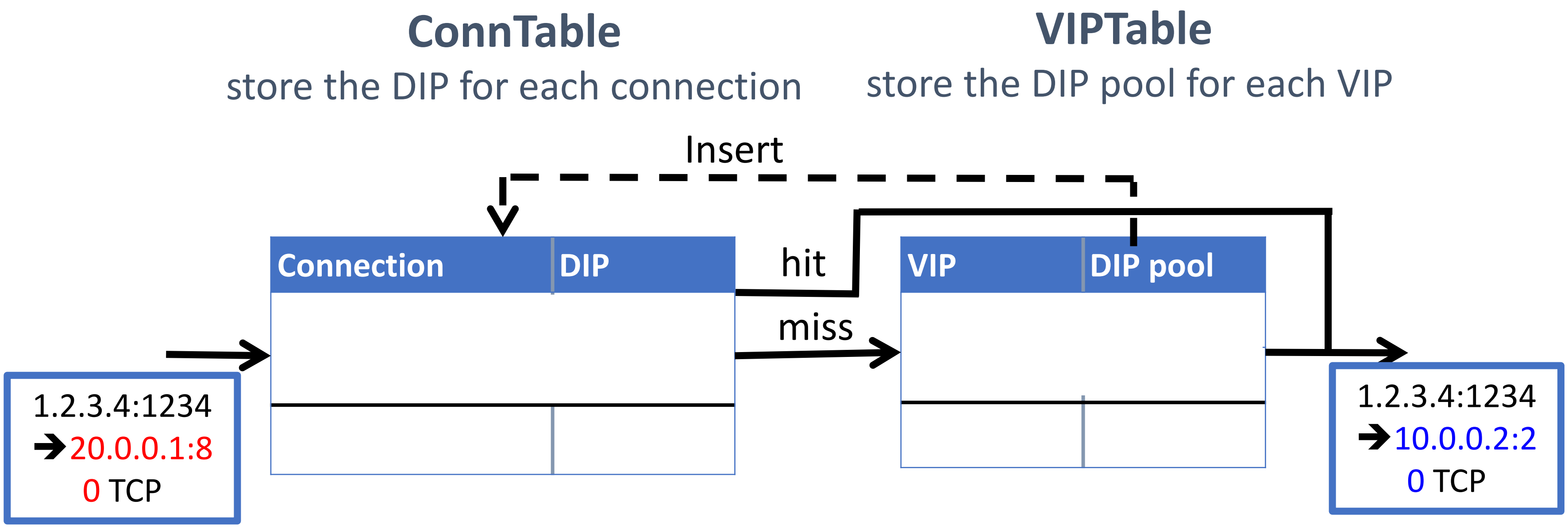
SilkRoad

Address such challenges using programmable hardware switch.

Scale to traffic growth: Build on switching ASICs with multi Tbps

PCC guarantee: key challenge

ConnTable in ASICs



Design challenges

Challenge 1: store millions of connections in ConnTable

Approach: novel hashing design to compress ConnTable

Challenge 2: do all the operations (e.g., PCC) in a few nanoseconds

Approach: use hardware primitives to handle connection state and its dynamics

Many active connections in ConnTable

- Up to 10 million active connections per rack in Facebook traffic
  - a naïve approach: 10M * (37-byte 5-tuple + 18-byte DIP) = 550 MB

Approach: novel hashing design to compress ConnTable

Compact connection match key by hash digests

Handling hash collisions

- the chance is small (<0.01%)
- detect collision and migrate entry to another stage with different hash function

ConnTable

5-tuple (37-byte)

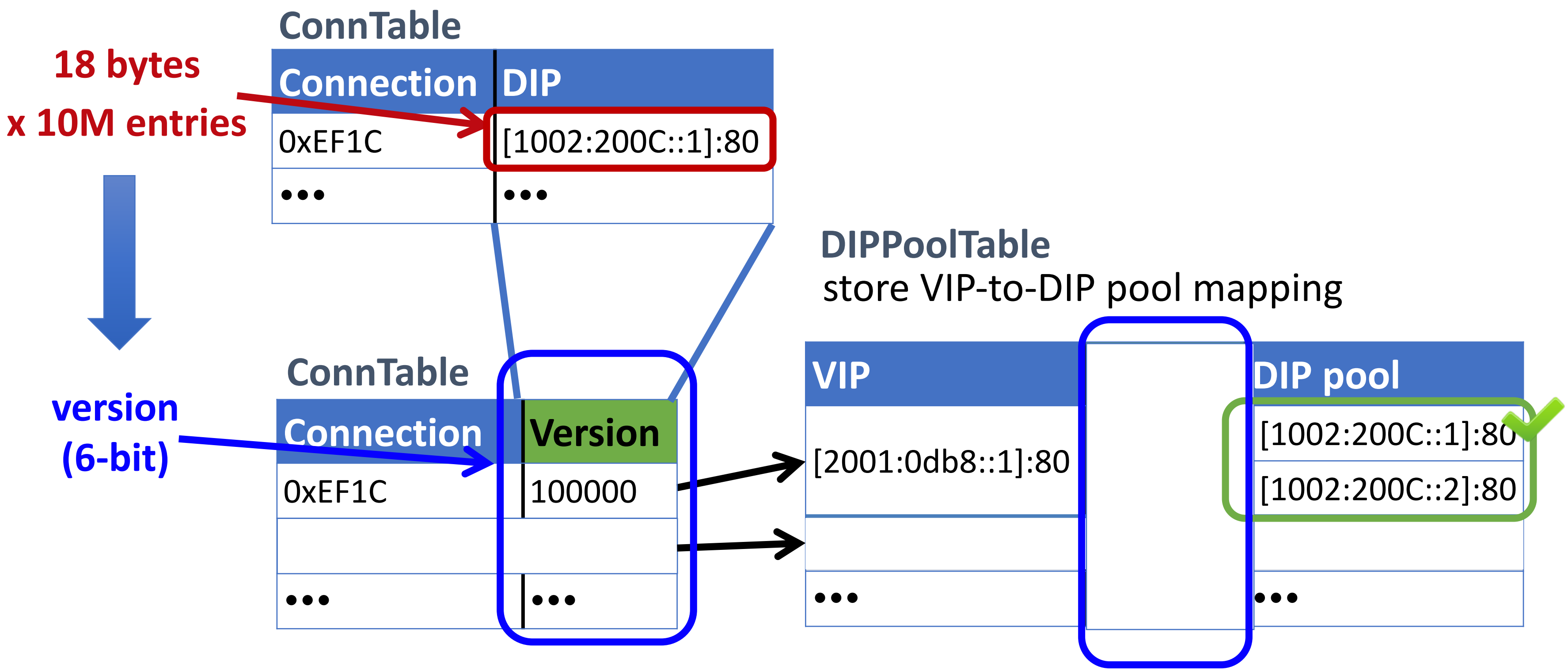
Connection	DIP
[2001:0db8::2]:1234→[2001:0db8::1]:80 TCP	[1002:200C::1]:80
•••	•••

hash digest (16-bit)

Connection	DIP
0xEF1C	[1002:200C::1]:80
•••	•••

Approach: compress ConnTable

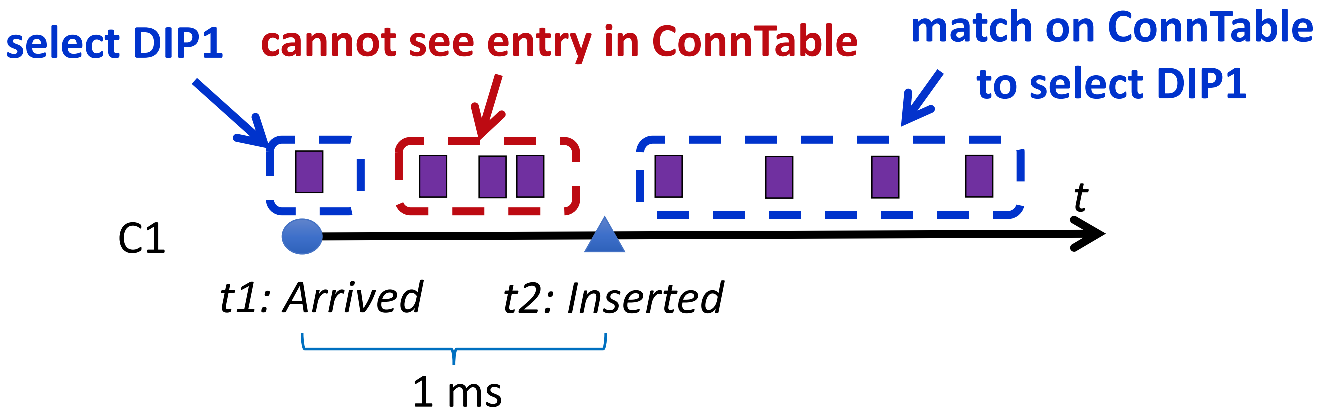
Compact action data with DIP pool versioning



Entry insertion is not atomic in ASICs

ASIC feature: ASICs use highly efficient hash tables

- fast lookup by connections (content-addressable)
- high memory efficiency
- but, require switch CPU for entry insertion, which is not atomic



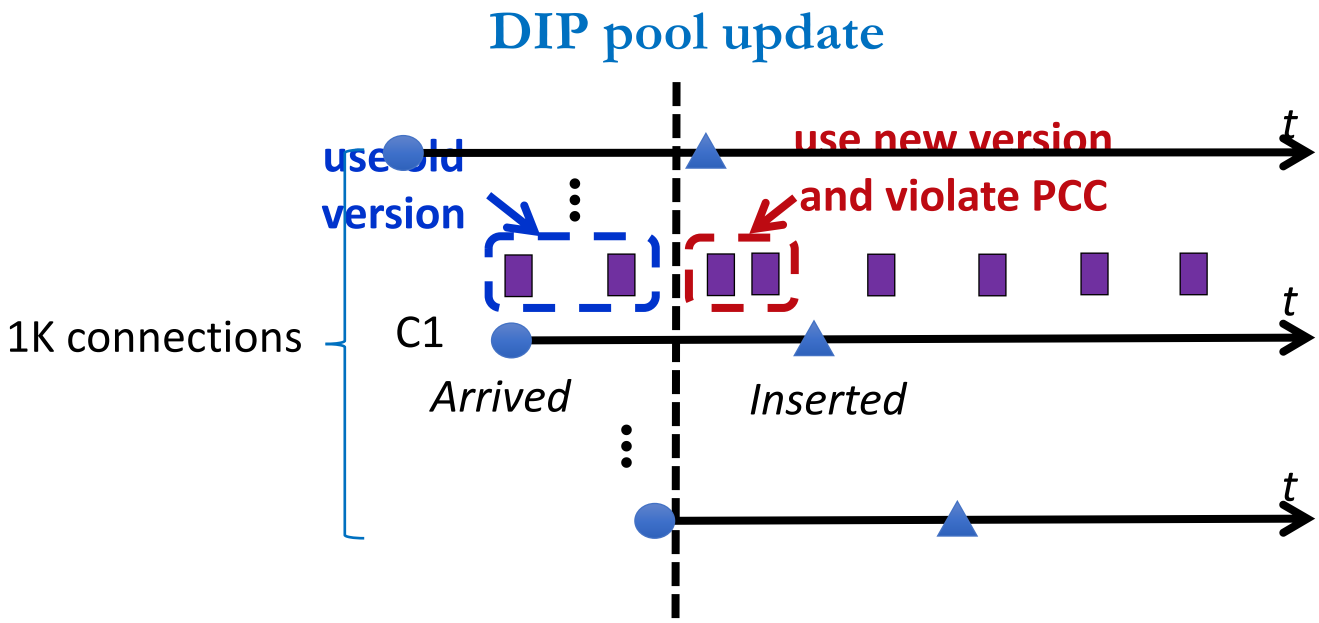
C1 is a pending connection between t1 and t2

Many broken connections under DIP pool updates

DIP pool update breaks PCC for pending connections

Frequent DIP pool updates

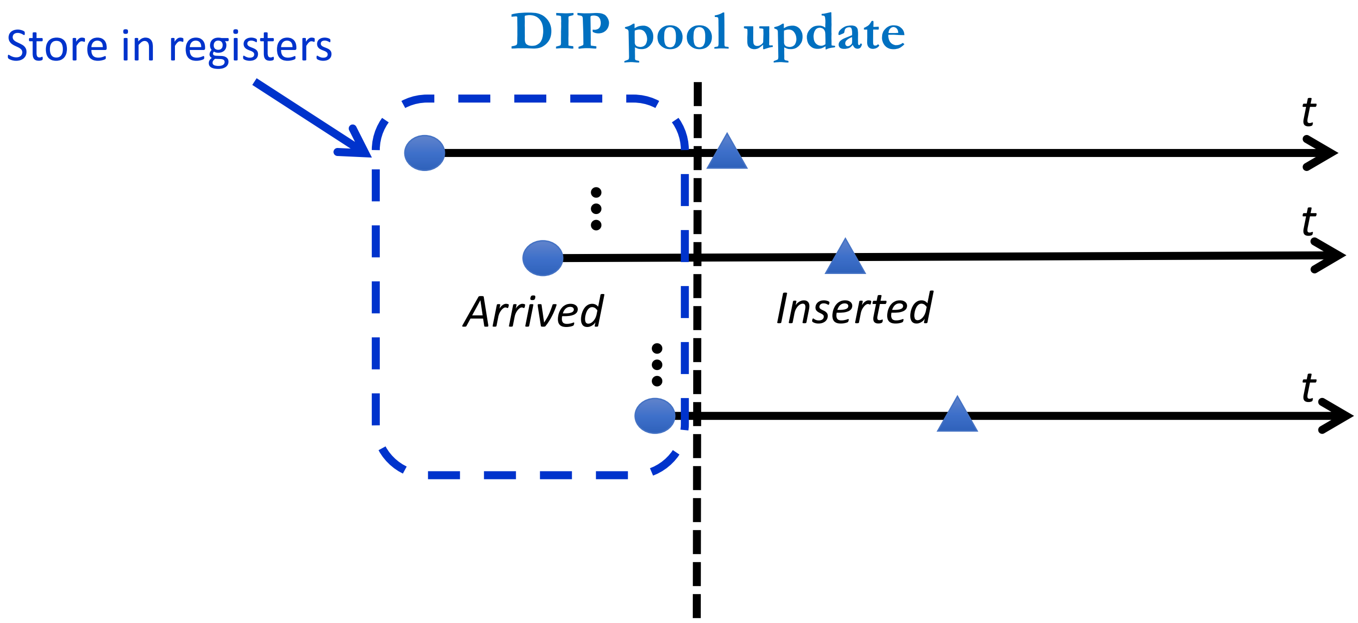
- a cluster has up to 100 updates per minute



Approach: registers to store pending connections

ASIC feature: registers

- support atomic update directly in ASICs
- store pending connections in registers

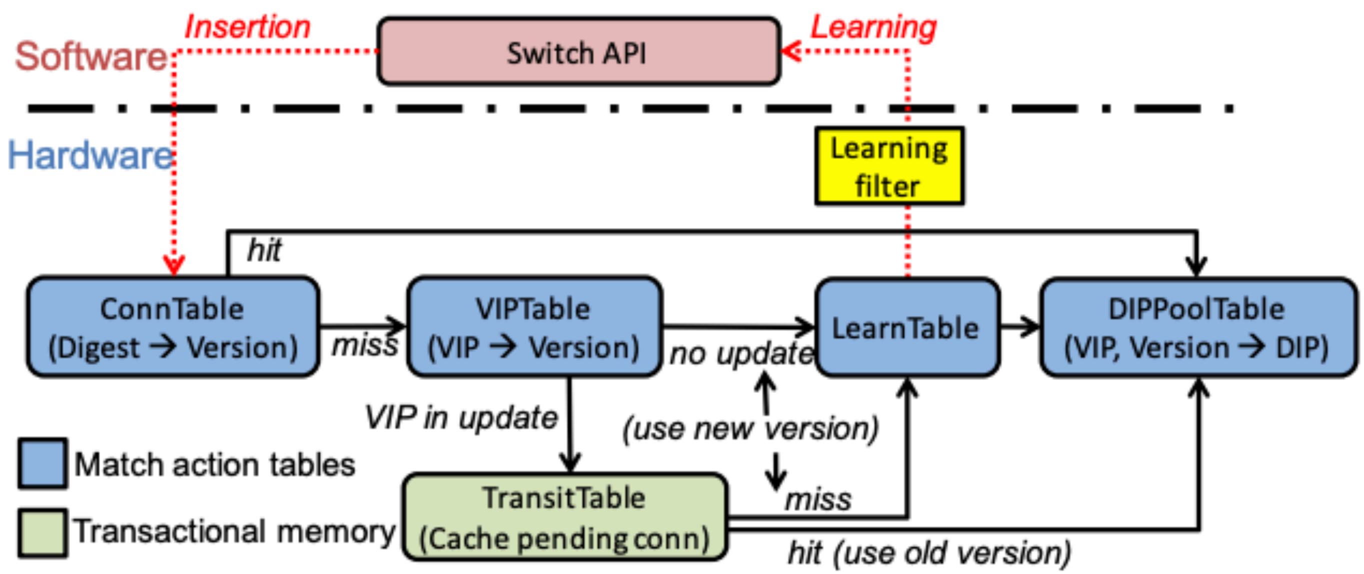


Approach: registers to store pending connections

Key idea: use Bloom filters to separate old and new DIP pool versions

- store pending connections with old DIP pool version
- other connections choose new DIP pool version
- this is a membership checking, and only need index addressable

System Architecture



Prototype performance

Throughput

- a full line rate of 6.5 Tbps
- one SilkRoad can replace up to 100s of software load balancers
- save power by 500x and capital cost by 250x

Latency

- sub-microsecond ingress-to-egress processing latency

Robustness against attacks and performance isolation

- high capacity to handle attacks
- use hardware rate-limiters for performance isolation

PCC guarantee

Is this a good usecase of programmable dataplanes?

What are the limitations?

What could have been an alternate strategy?

Which paper did you like the most?

- BeauCoup
- Elmo
- NetCache
- Silkroad

Which paper did you dislike the most?

- BeauCoup
- Elmo
- NetCache
- Silkroad

Other app-level 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.
- SailFish: cloud gateway deployed by Alibaba (SIGCOMM'21)
- Robot arm control (NSDI'22)
- ….

Logistics

- Feedback on your reviews.
- Warm-up assignment 2 due today.
- First project report due next Friday (10/13).