Pushing the Limits of In-Network Caching for Key-Value Stores

Gyuyeong Kim

USENIX NSDI 2025

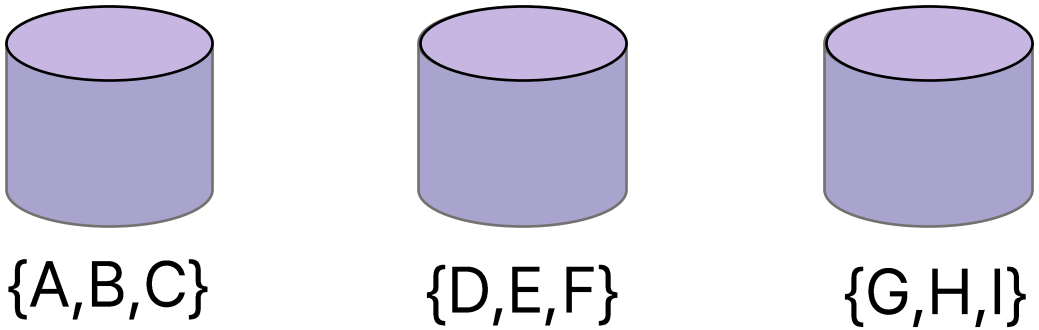
성신여자대학교
SUNGSHIN WOMEN'S UNIVERSITY

Distributed Key-Value Stores

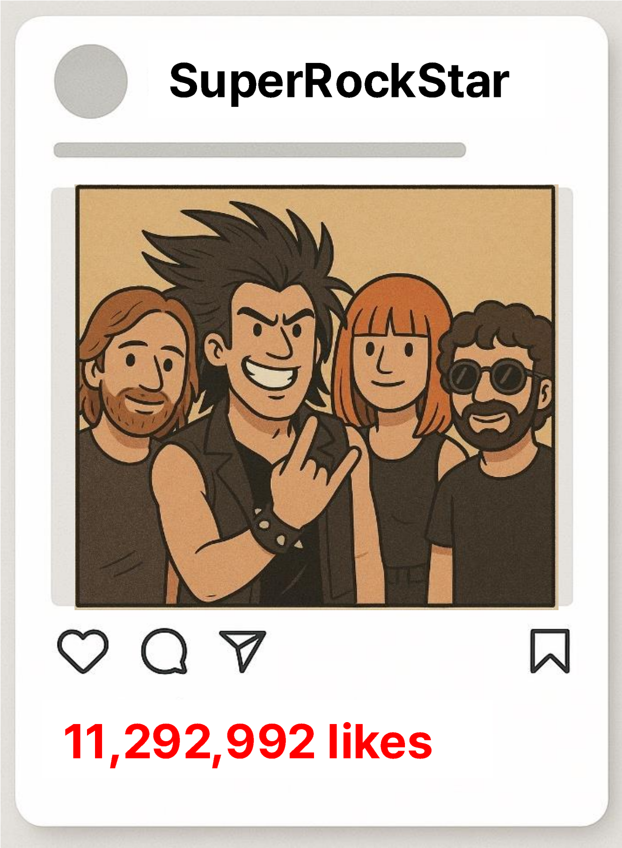
- Fundamental building blocks for modern online services
- Simple and fast data access
  - Requires low tail latency and high throughput

Key	Value
Key1	Value1
Key2	Value2
Key3	Value3

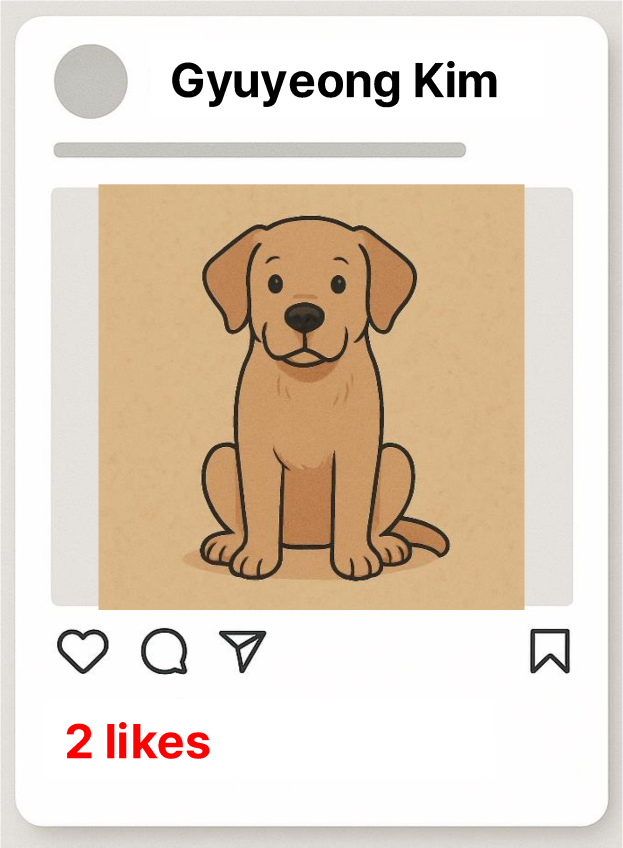
- Data is partitioned over multiple servers



Item Popurarity is Highly Skewed

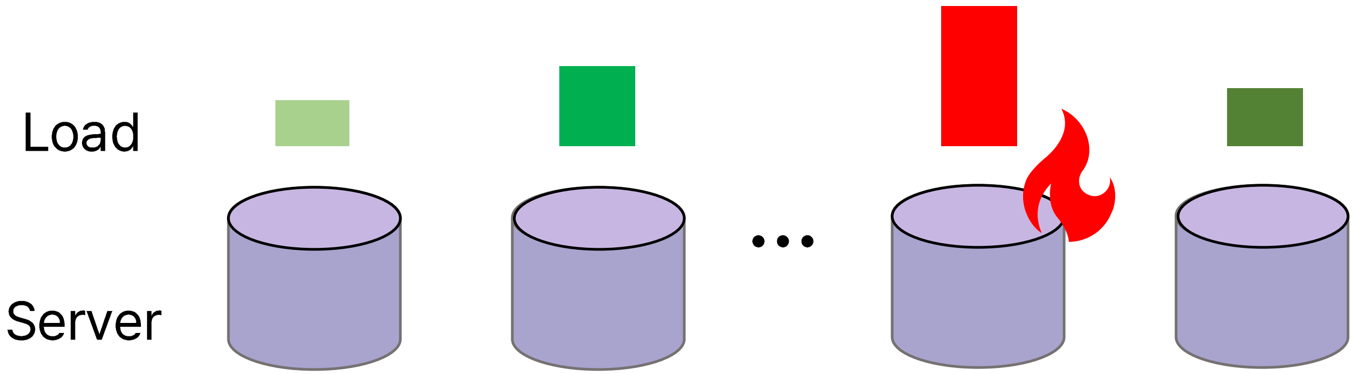


vs.



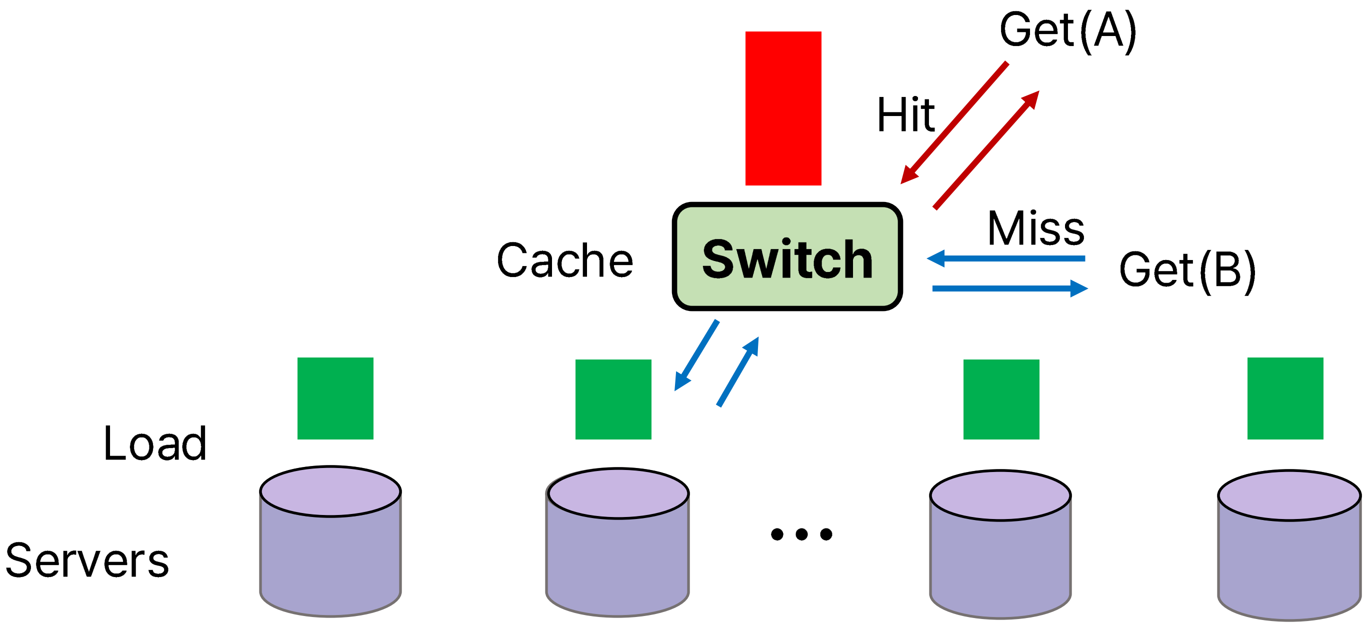
How to Handle Load Imbalance?

- **Skewed item popularity** causes **load imbalance** between servers
- Servers with hot items are overloaded



In-Network Caching

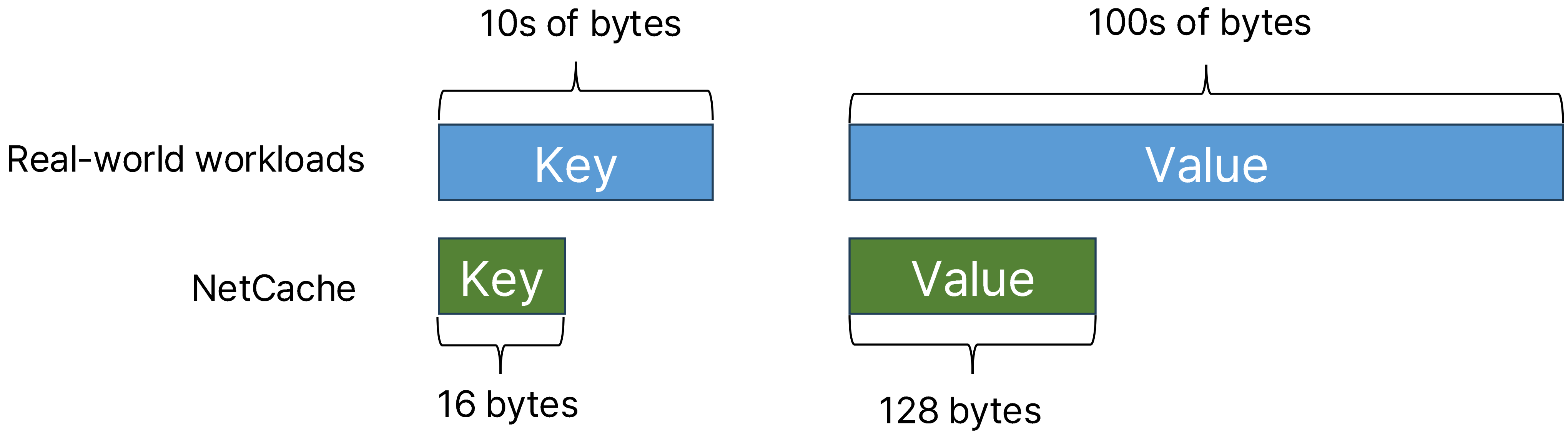
- Leverages **programmable switches** as a front load-balance cache
  - NetCache [SOSP’17], DistCache [FAST’19], FarReach [ATC’23]
- **Small cache, big effect**: caching $O(N \, log \, N)$ hottest items is enough
  - N: # of servers/partitions, not # of items nor requests [B. Fan et al., SoCC’11]^



^Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky, "Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services," in *Proc. of ACM SoCC*, 2011.

Limitation: Too Small Cacheable Item Size

- NetCache supports items up to 16-B keys and 128-B values
- Key-value items are small, but this is far from production workloads
- NetCache cannot cache even a single item for 42 of 54 Twitter workloads*

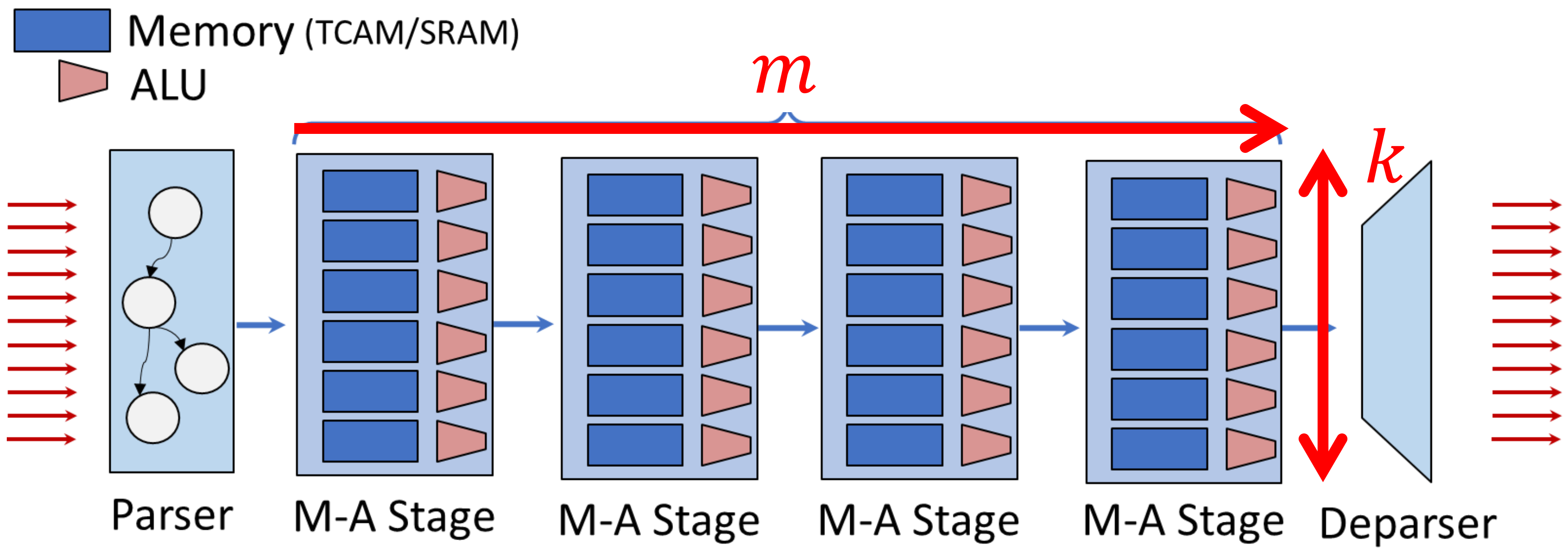


*Juncheng YAng, Yao Yue, and Rashmi Vinayak, "A large scale analysis of hundreds of in-memory cache clusters at Twitter," in *Proc. of USENIX OSDI*, 2020. (Dataset is publicly available in a Github repository)

How to Enable Variable-Length In-Network Caching?

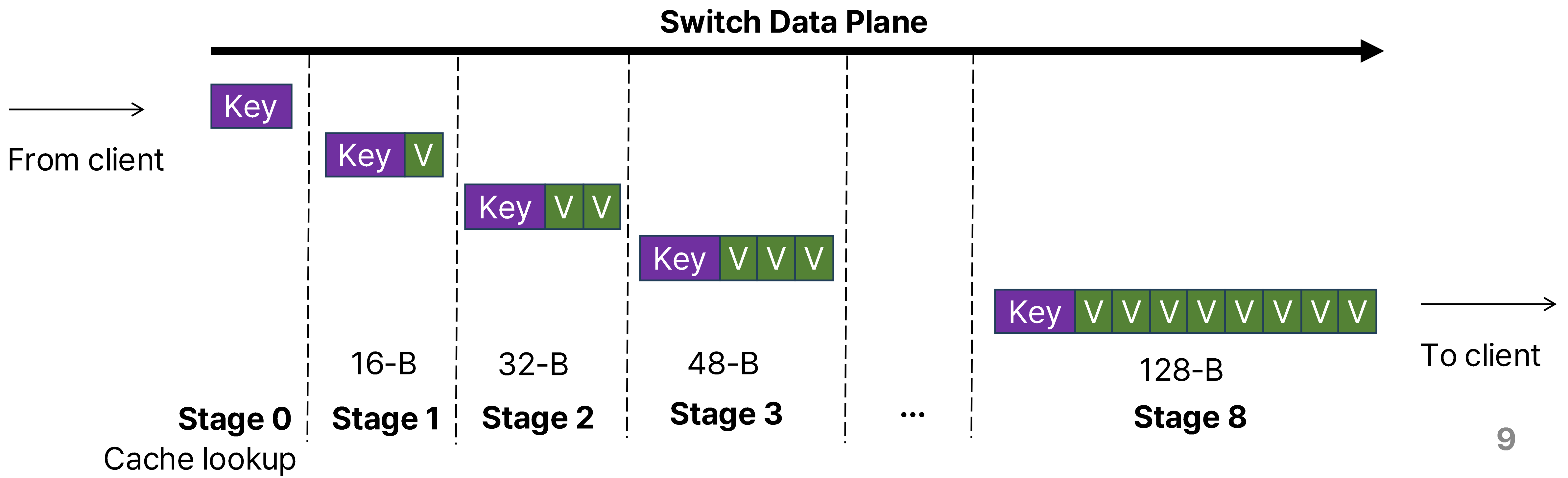
Memory Access in the Switch Data Plane

- The switch data plane consists of m **Match-Action (M-A) stages**
- Each M-A stage has a static memory and a few ALUs
- Packets go through a chain of M-A stages
- The switch can handle $\boldsymbol{k}$ **bytes per stage**



Why Is Value Size Limited?

- The value is fragmented over $n < m$ stages and each stage can handle k bytes
- The switch appends the value fragments to the packet within $\boldsymbol{n \times k}$ **constraint**
- E.g., if $n = 8$ and $k = 16$, the switch can cache values up to 128 bytes



Why Is Key Size Limited?

- The cache lookup table is implemented using a M-A table
- M-A table has the maximum width for the match key
  - The item key is the match key of the lookup table

```
table cache_lookup{
       key = {
           pkt.key: exact;
       }
       actions = {
           cache_hit;
           cache_miss;
       }
       size = 65536;
       default_action = cache_miss;
   }
```

Limited

Match (pkt.key)	Action (cache_hit)
A	Idx=0, …
B	Idx=1, …
C	Idx=2, …
D	Idx=3, …

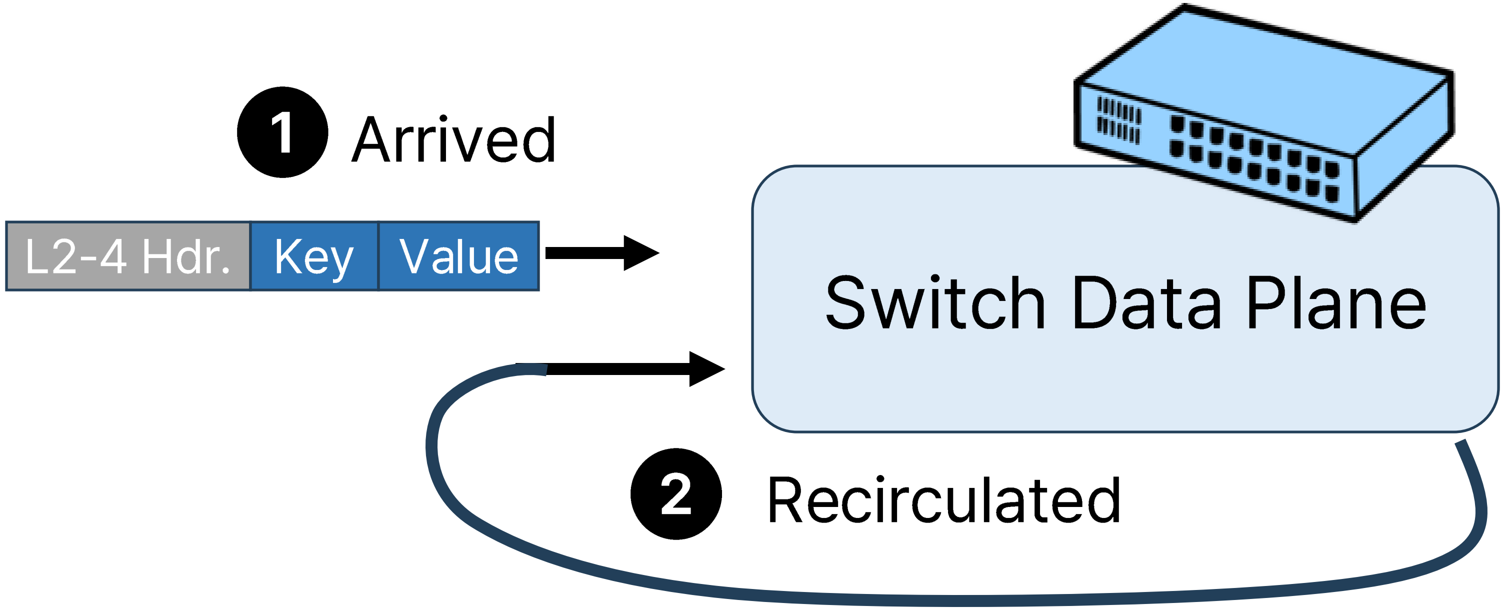
It is hard to realize variable-length in-network caching,
if we stick to the concept of caching data in the switch memory

Why? $n \times k$ is determined at the time of manufacturing

Where should we cache data instead of switch memory?

OrbitCache: Recirculation-based Caching

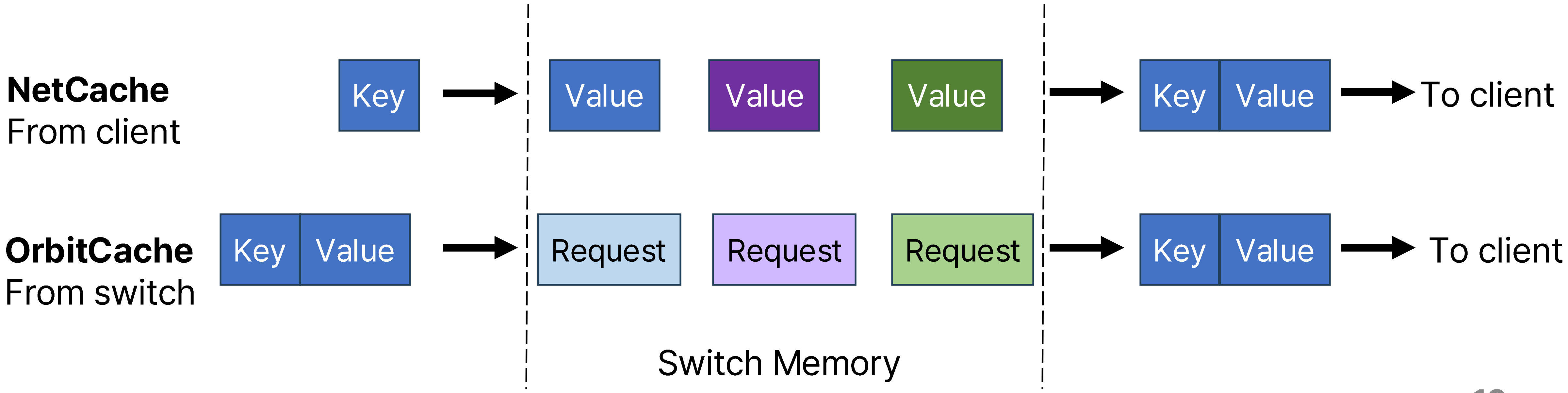
- **Idea:** Keeps cached items circulating using **packet recirculation**
- Recirculation makes the packet visit the switch data plane again
  - The switch has an internal loopback port for recirculation
- **No fragments, no size limits, but data is in the switch data plane**



Comparison with NetCache Architecture

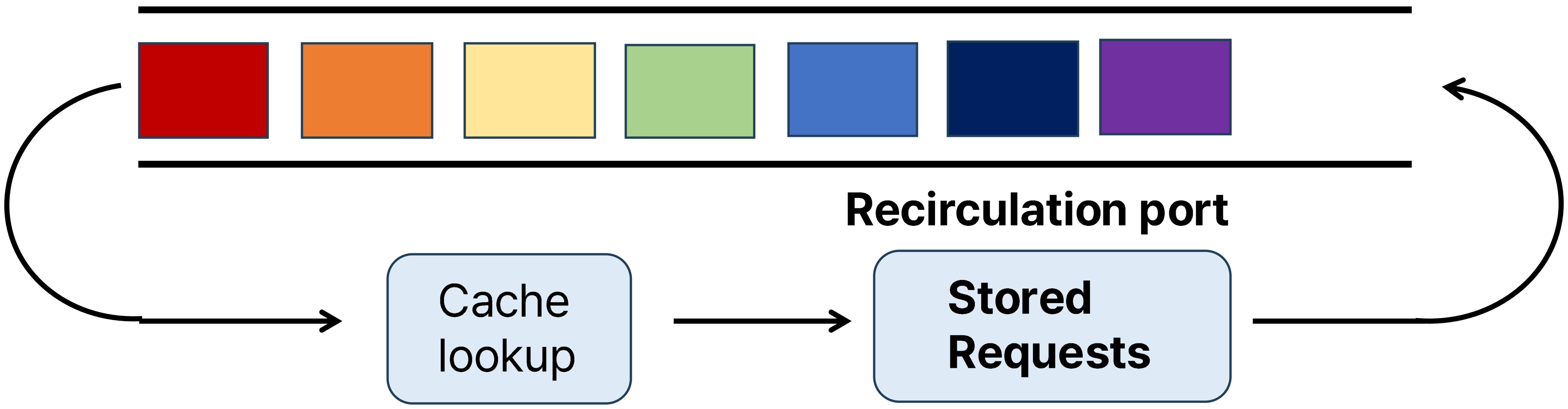
NetCache: Requests read cached data

OrbitCache: Cached data reads stored requests



Trade-Off in Cache Size

- The time to read a stored request is impacted by other in-flight cache packets
- Only a small number of items can be cached
  - Recall that we need only $O(N \log N)$ hottest items for load balancing



Technical Challenges

1. How to maintain multiple requests in the switch memory?

2. How to make a cache packet serve multiple requests once fetched?

3. How to ensure cache coherence?

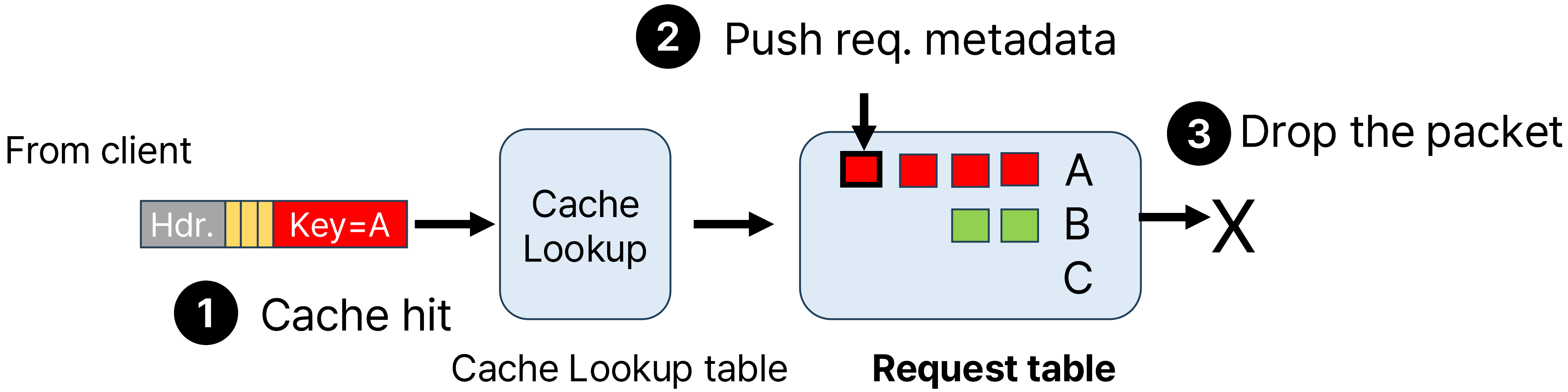
4. How to update cache entries?

Technical Challenges

1. How to maintain multiple requests in the switch memory?
2. How to make a cache packet serve multiple requests once fetched?
3. How to ensure cache coherence?
4. How to update cache entries?

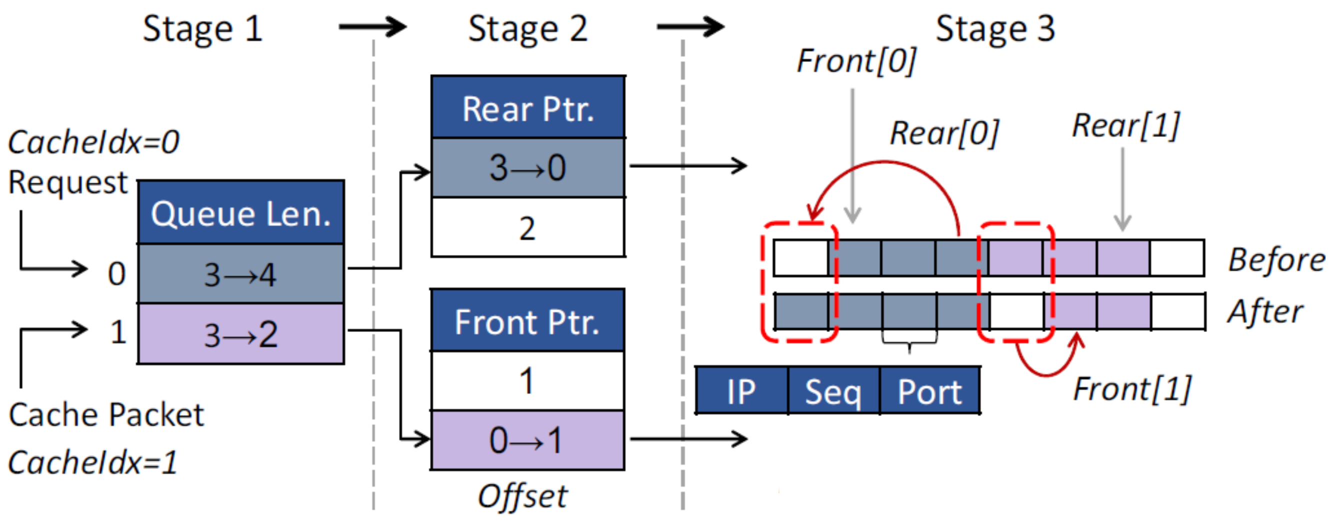
Handling Requests With Cache Hit

- The switch drops the request after inserting it into the queue
- Requests will be handled by circulating cache packets soon

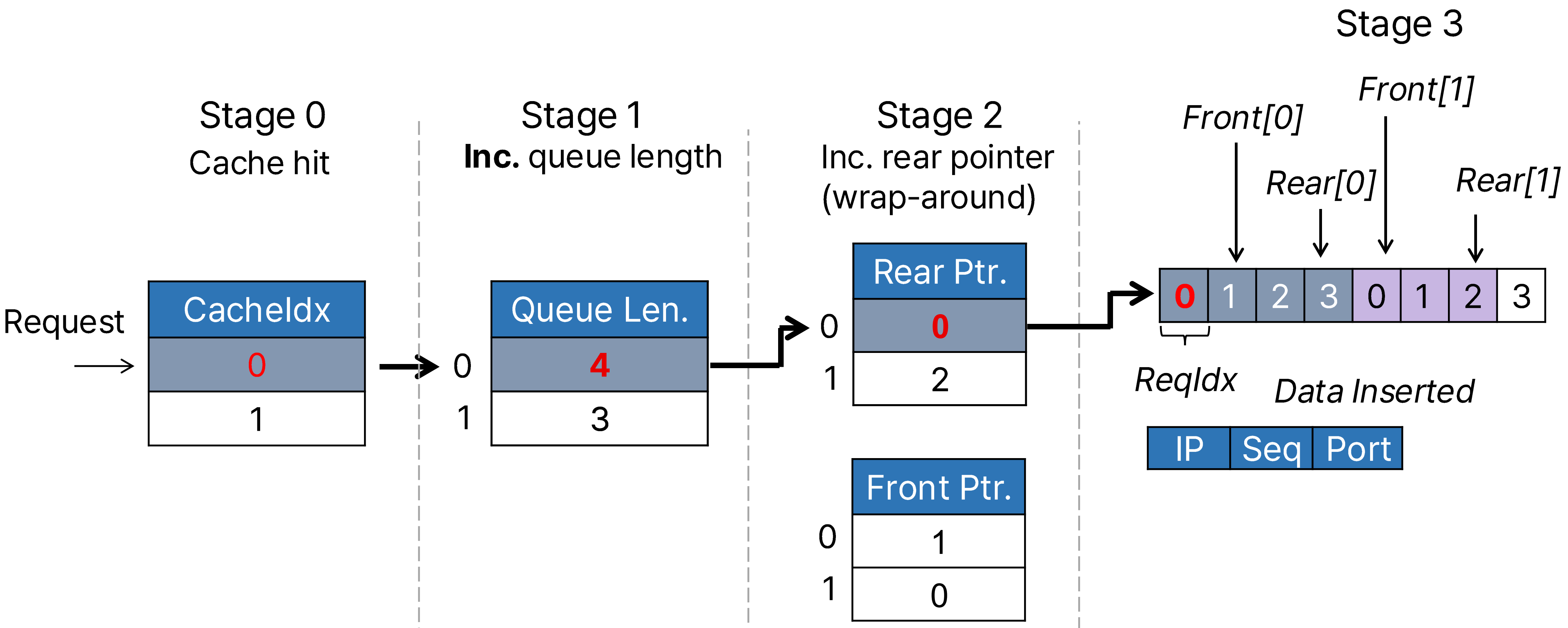


Request Table: In-Switch Circular Queue

- Supports per-key request queue with small memory footprints
- The table consists of a few register arrays
  - Request metadata, queue length, and the front/tail pointers



Enqueue for Request Packets



Metadata
CacheIdx=0
ReqIdx=0
Offset=0

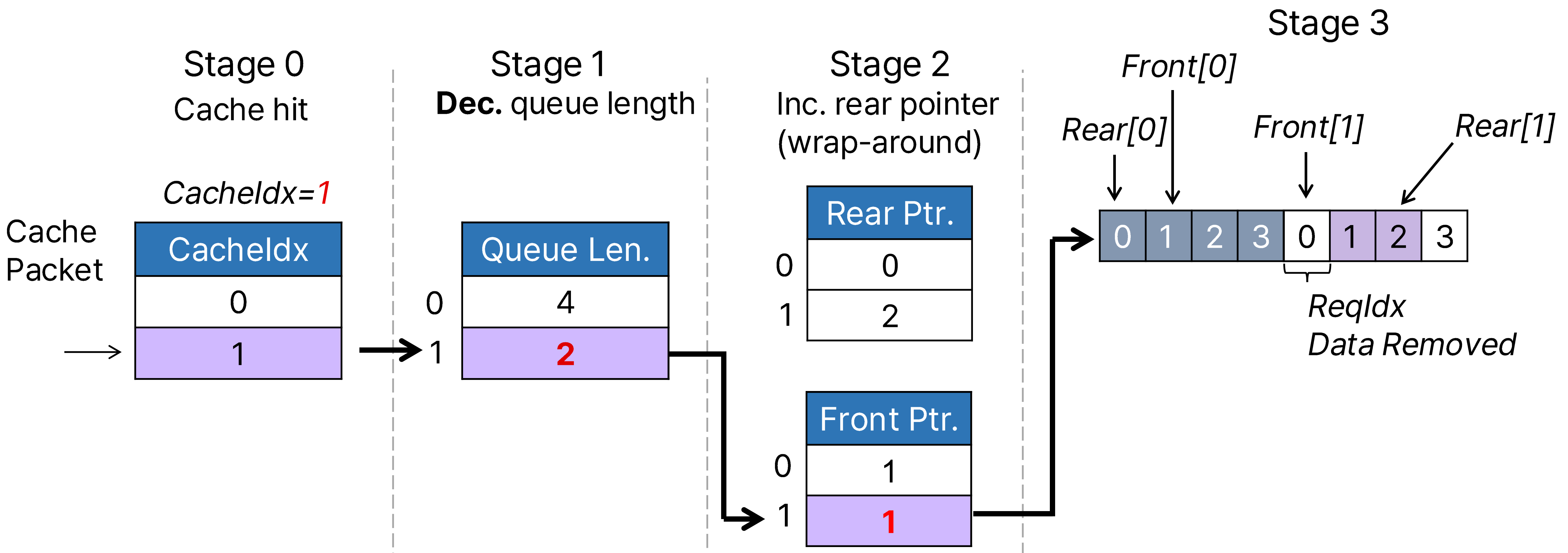
Metadata
CacheIdx=0
ReqIdx=0
Offset=0

Metadata
CacheIdx=0
ReqIdx=0
Offset=0

*ReqIdx=CacheIdx*MaxQLen+Offset*

MaxQlen = 4

Dequeue for Cache Packets



Metadata
CacheIdx=1
ReqIdx=0
Offset=0

Metadata
CacheIdx=1
ReqIdx=0
Offset=0

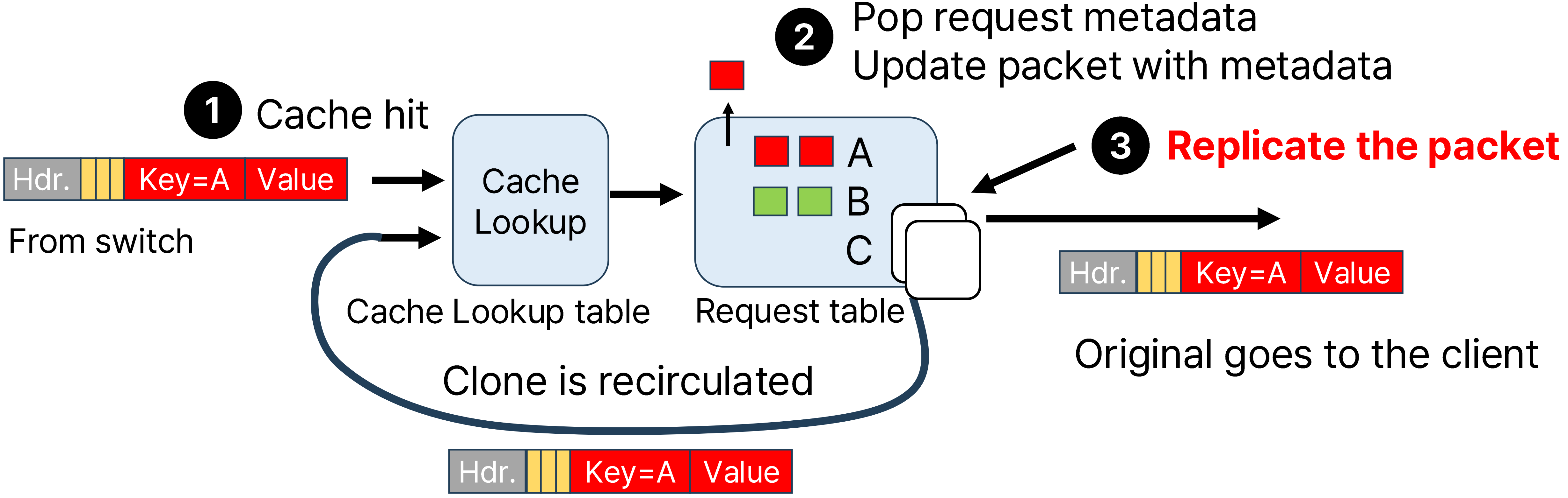
Metadata
CacheIdx=1
ReqIdx=5
Offset=1

*ReqIdx=CacheIdx*MaxQLen+Offset*

MaxQlen = 4

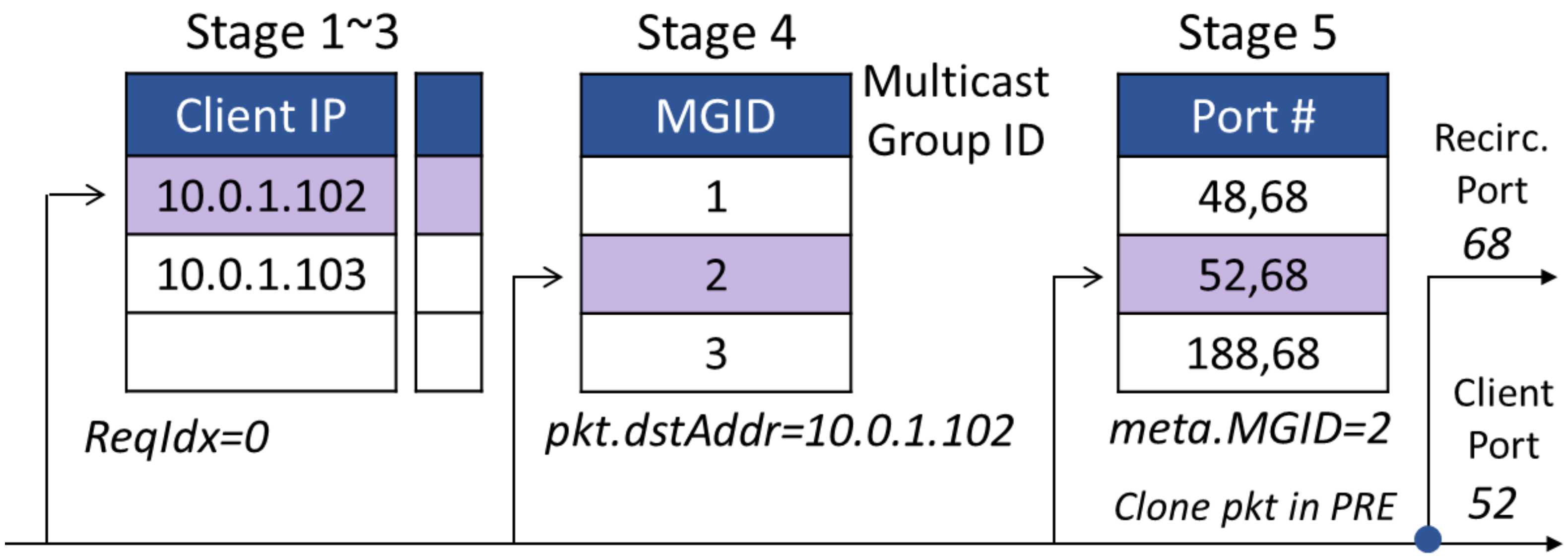
Handling Cache Packets when Requests Exists

- Packet replication makes the cache packet serve more requests



Replicating Cache Packets for Further Serving

- Implemented with multicast functionality
  - Each multicast group ID specifis a pair of ports
    - The recirculation port and the client-directed port

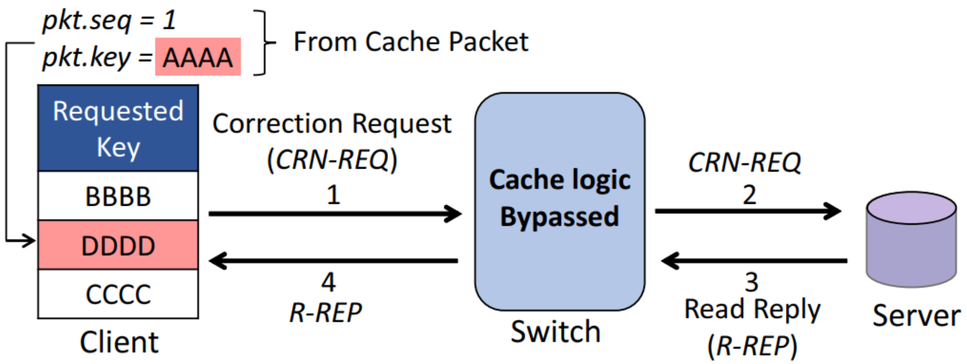


Supporting Variable-Length Keys

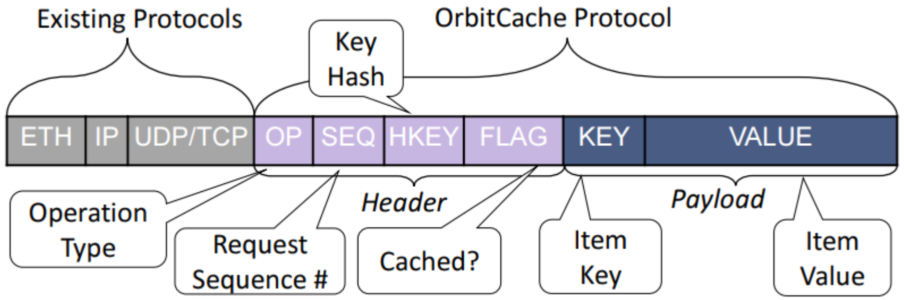
- 128-bit keyhash for cache lookup table

Keyhash	CacheIdx
h(A)	0
h(B)	1
h(C)	2

- How to resolve hash collisions?
  - Detecting hash collisions at the client by comparing the maintained key and the retrieved key
  - The client gets the correct value from the storage server



Hash Collision Resolution Mechanism



OrbitCache Packet Format

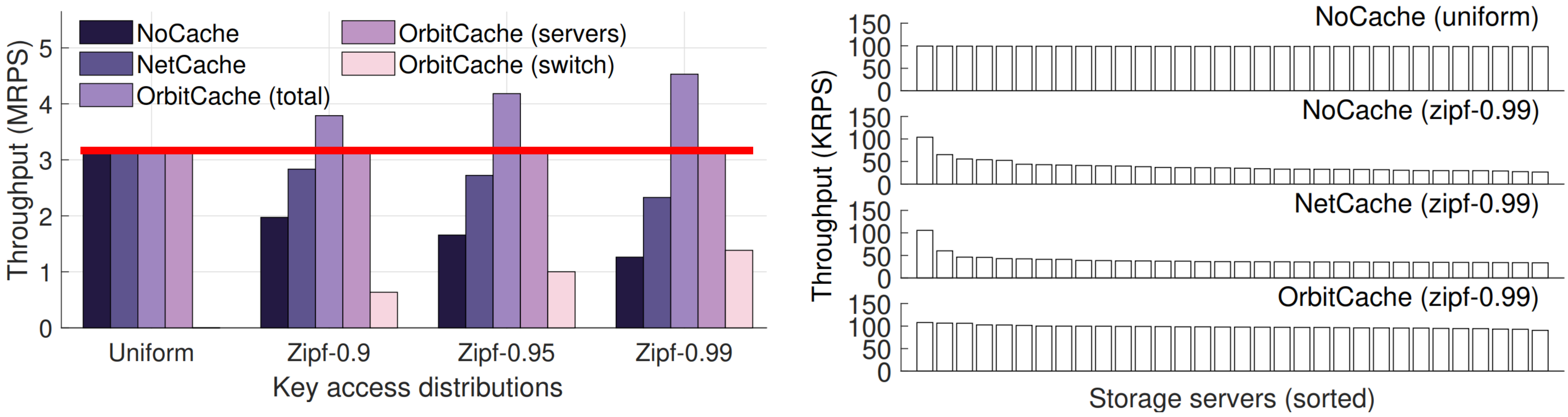
Implementation

- Switch data plane
  - Intel Tofino switch ASIC
  - Written in $\mathrm{P4}_{16}$
- Clients and servers
  - Open-loop multi-threaded applications in C
  - NVIDIA VMA for kernel-bypass packet processing
  - TommyDS for in-memory key-value stores

Evaluation

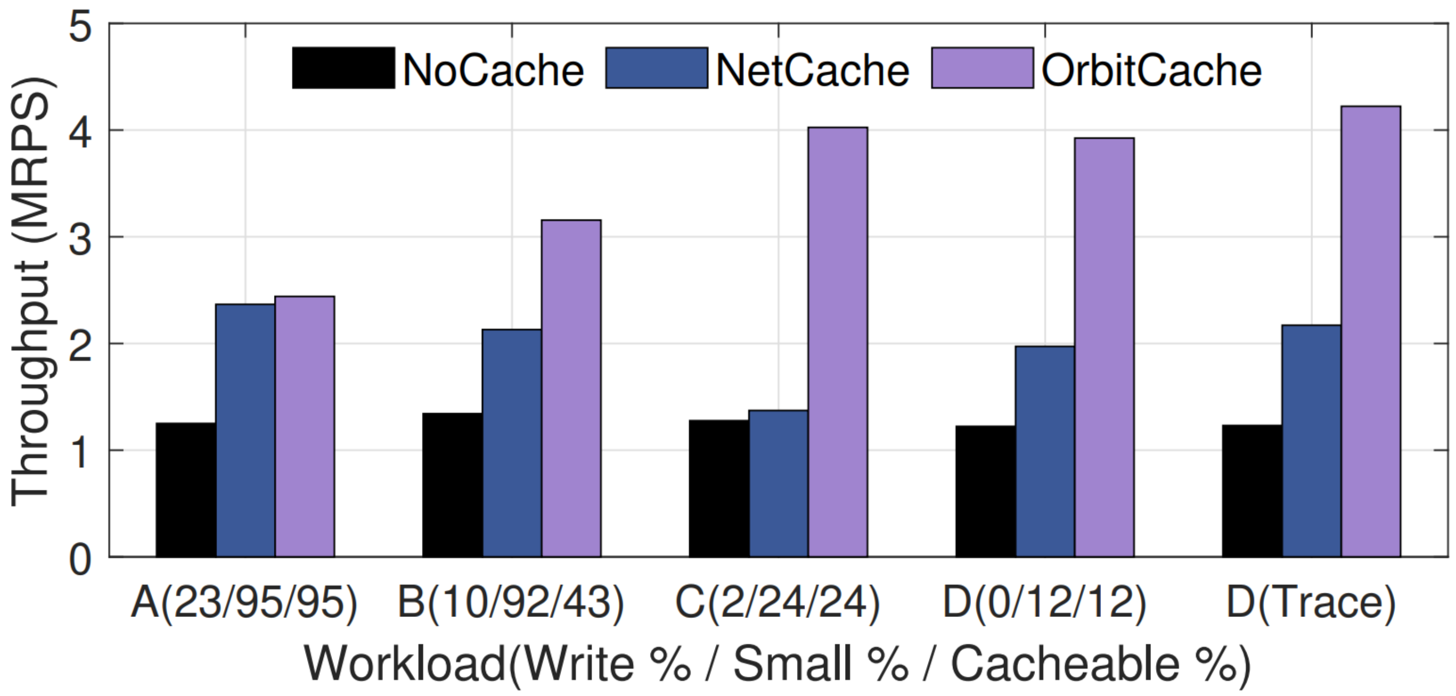
- Testbed
  - 6.5Tbps Intel Tofino switch
  - 8 nodes with Nvidia ConnectX-5 100G NIC
    - 4 nodes are clients
    - 4 nodes emulate multiple storage servers with per-core partitioning
- Default workload
  - 32 servers with 10M items
  - 128 cached items for OrbitCache, 10K cached items for NetCache
  - The `Cluster018` workload of Twitter
    - 82% items are cacheable by NetCache
- Compared Schemes
  - NoCache (the baseline without caching)
  - NetCache (X. Jin et al., SOSP'17)

Throughput with Different Skewness



OrbitCache can balance highly skewed workloads

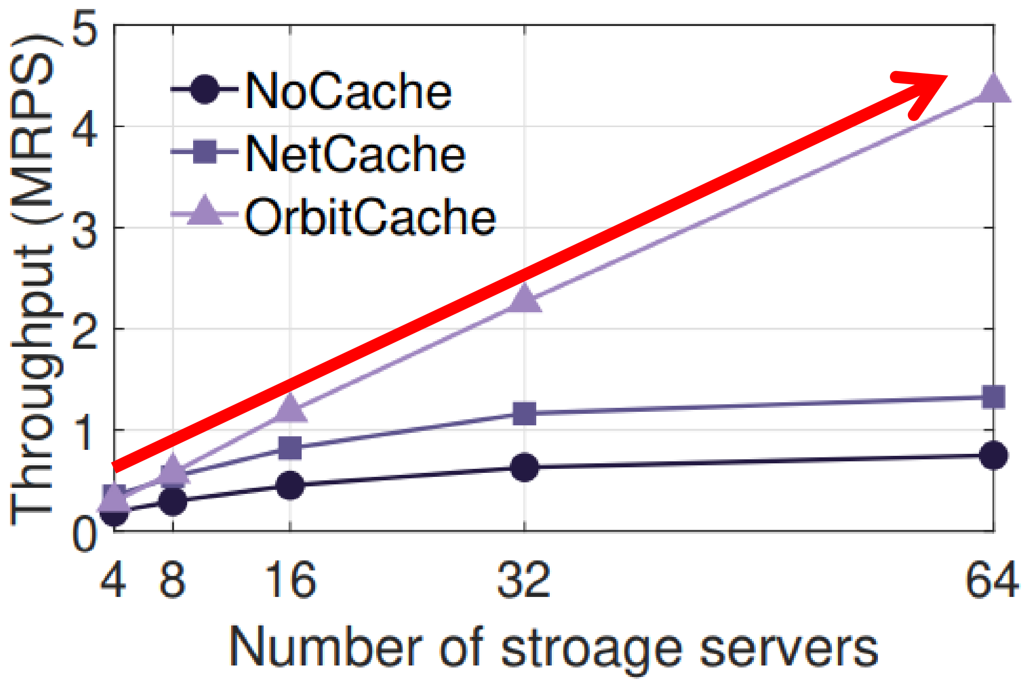
Performance with Diverse Workloads

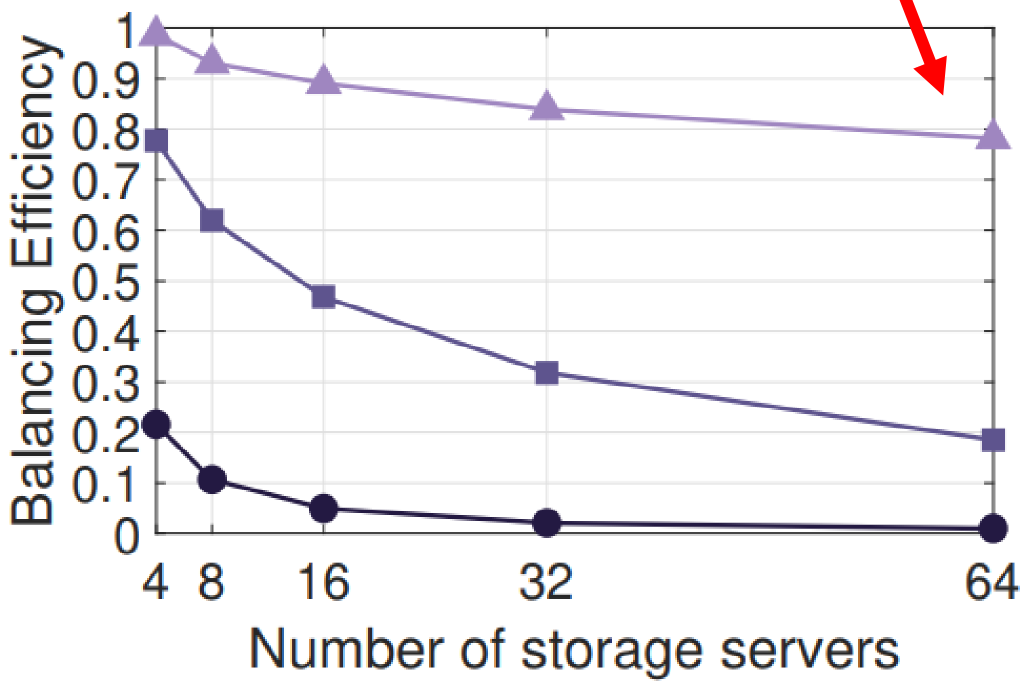


OrbitCache shows the best performance for all the workloads

Scalability

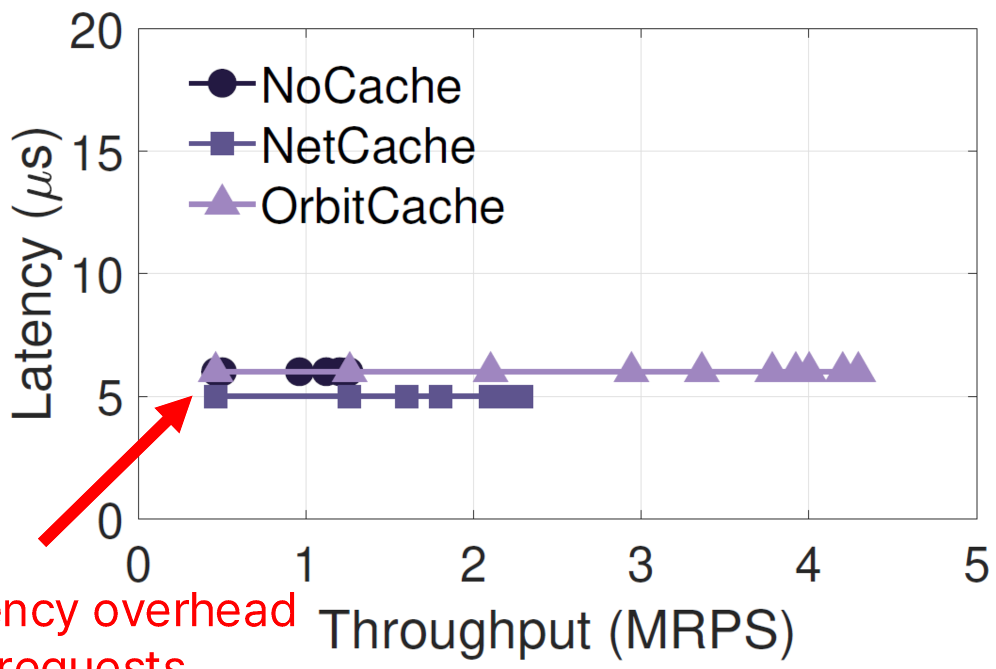
Still good balancing
efficiency with 64 servers



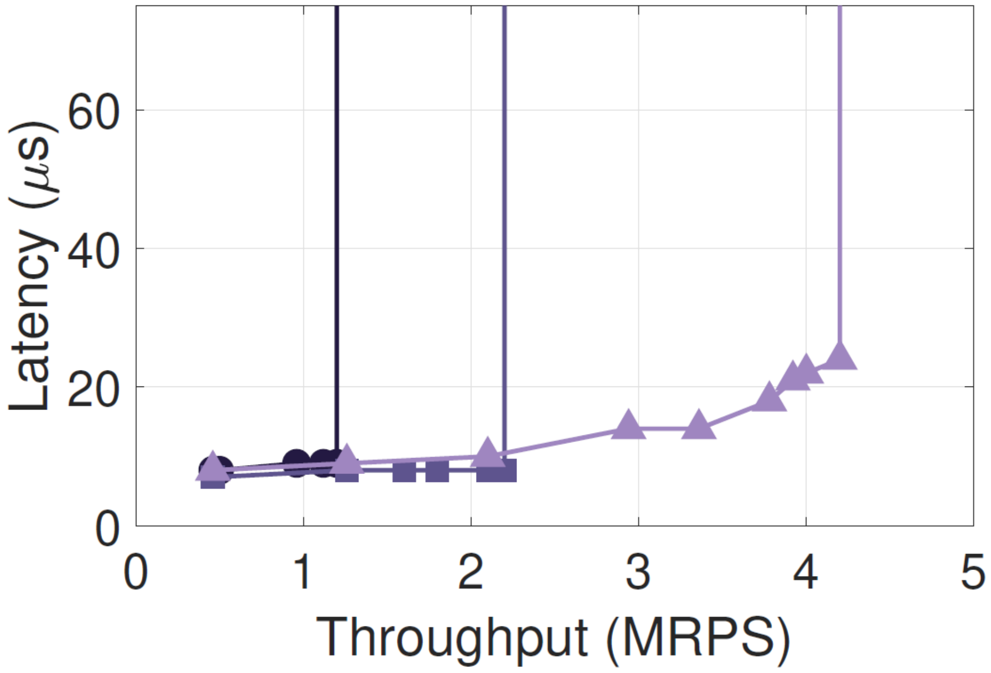


Scalable throughput while maintaining reasonable balancing efficiency

Latency vs. Throughput



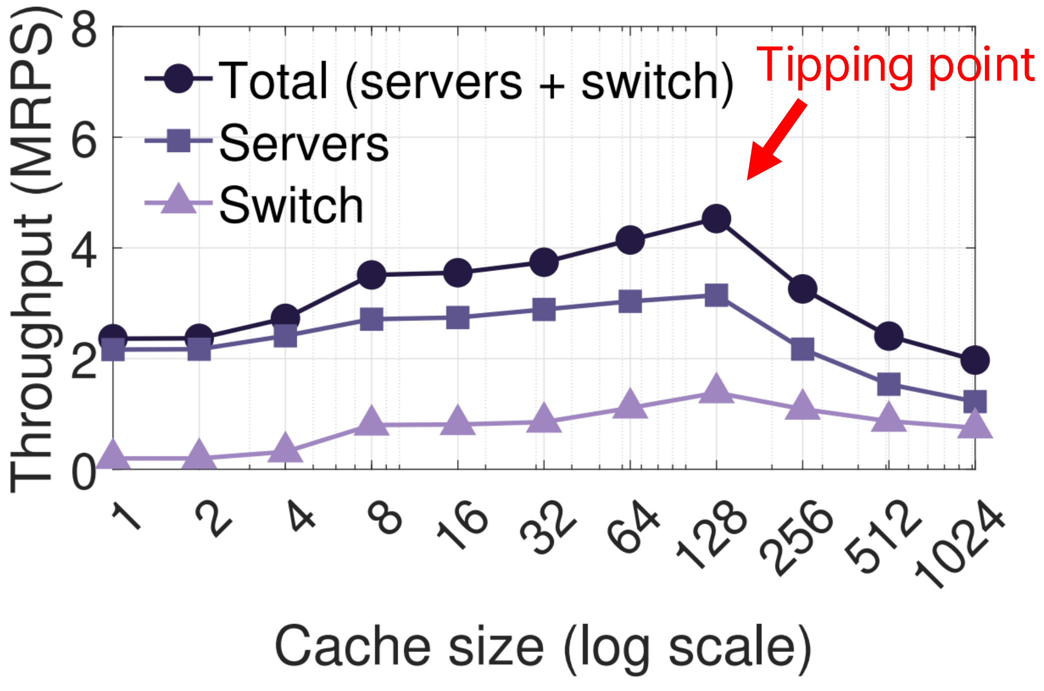
Median



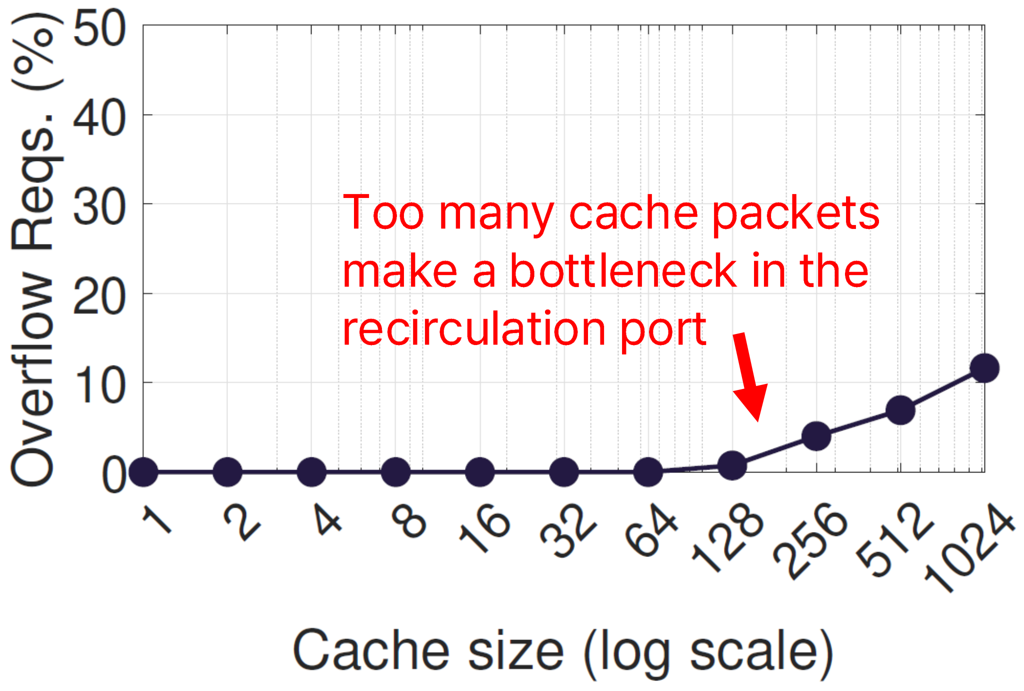
99th percentile

OrbitCache achieves the best throughput while provding comparable latency

Impact of Cache Size



Saturated throughput



Overflow request ratio

OrbitCache has a trade-off in the cache size but supports enough cache size to balance server loads

Conclusion

- OrbitCache efficiently uses packet recirculation to balance distributed key-value stores
  - Avoids hardware limitations by recirculating cache data in the form of cache packets
- Experimental results demonstrate the efficiency of OrbitCache for highly skewed workloads
- We provide insights that built-in switch features have great potential to make in-network computing mechanisms more effective

Thank you!

Questions?

gykim@sungshin.ac.kr
https://nslab.sungshin.ac.kr