HotRing: A Hotspot-Aware In-Memory Key-Value Store

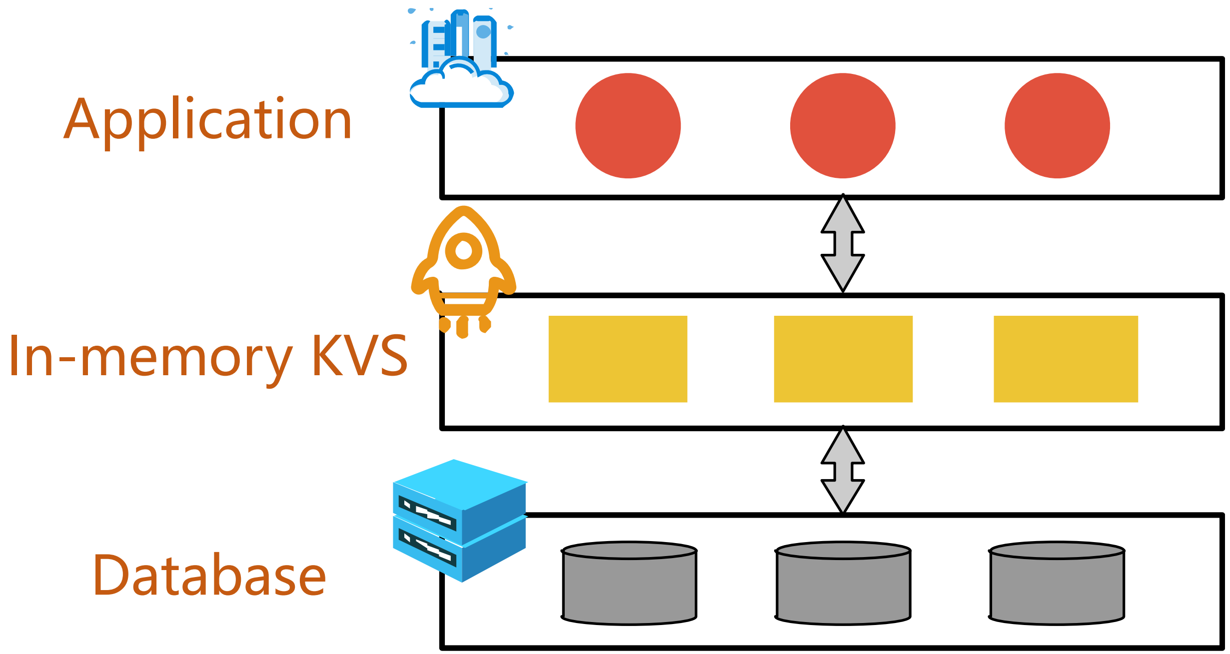
Le Cai

Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu,

Yuanyuan Sun, Huan Liu, Feifei Li

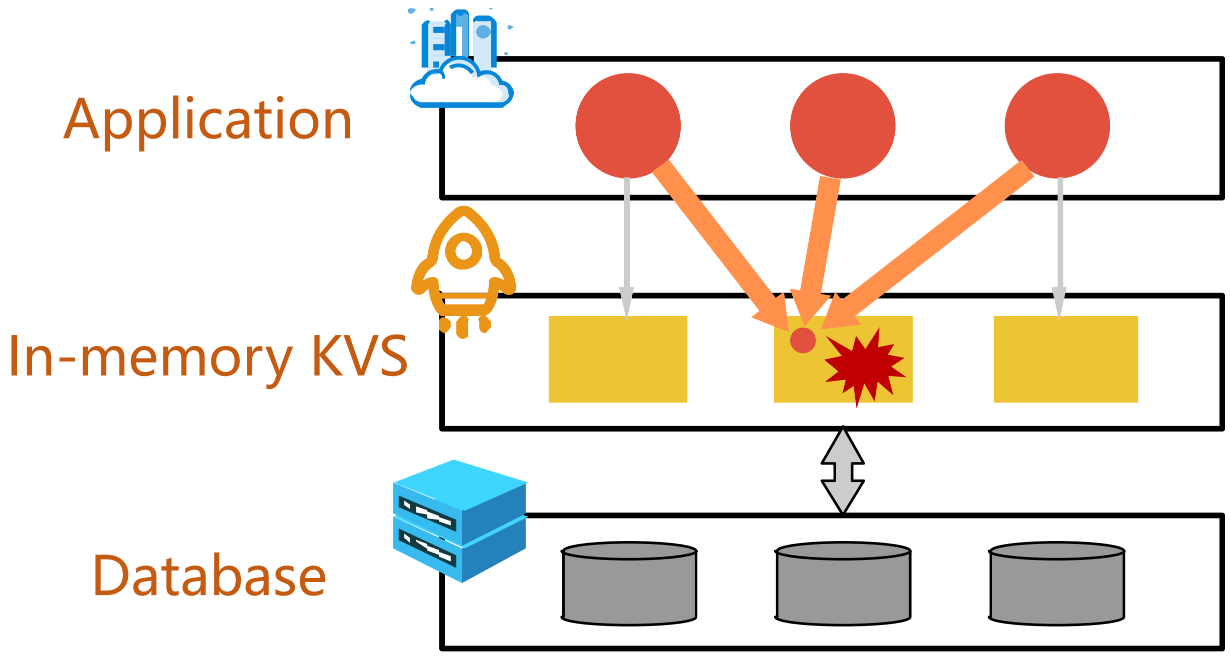
In-memory KV store

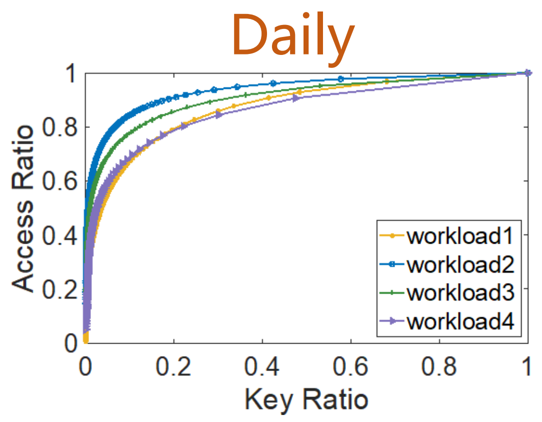
- In-memory KVSes have become an essential component in storage infrastructures
  - Cloud storage: Tair @ Alibaba
  - Social networks: Memcached @ Facebook

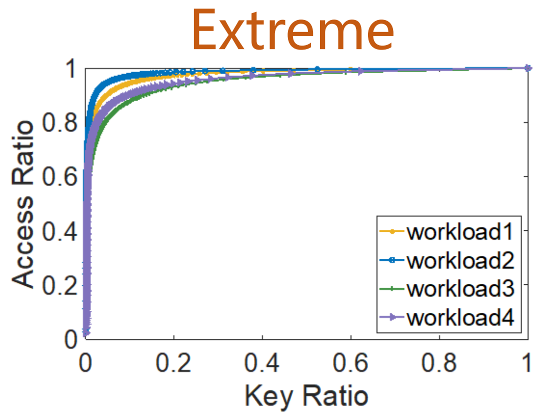


Hotspot issue

- A small portion of items that are frequently accessed
  - Daily distribution: 1% data holds 50% accesses
  - Extreme distribution: 1% data holds 90% accesses
  - E.g., iPhone 11 releases





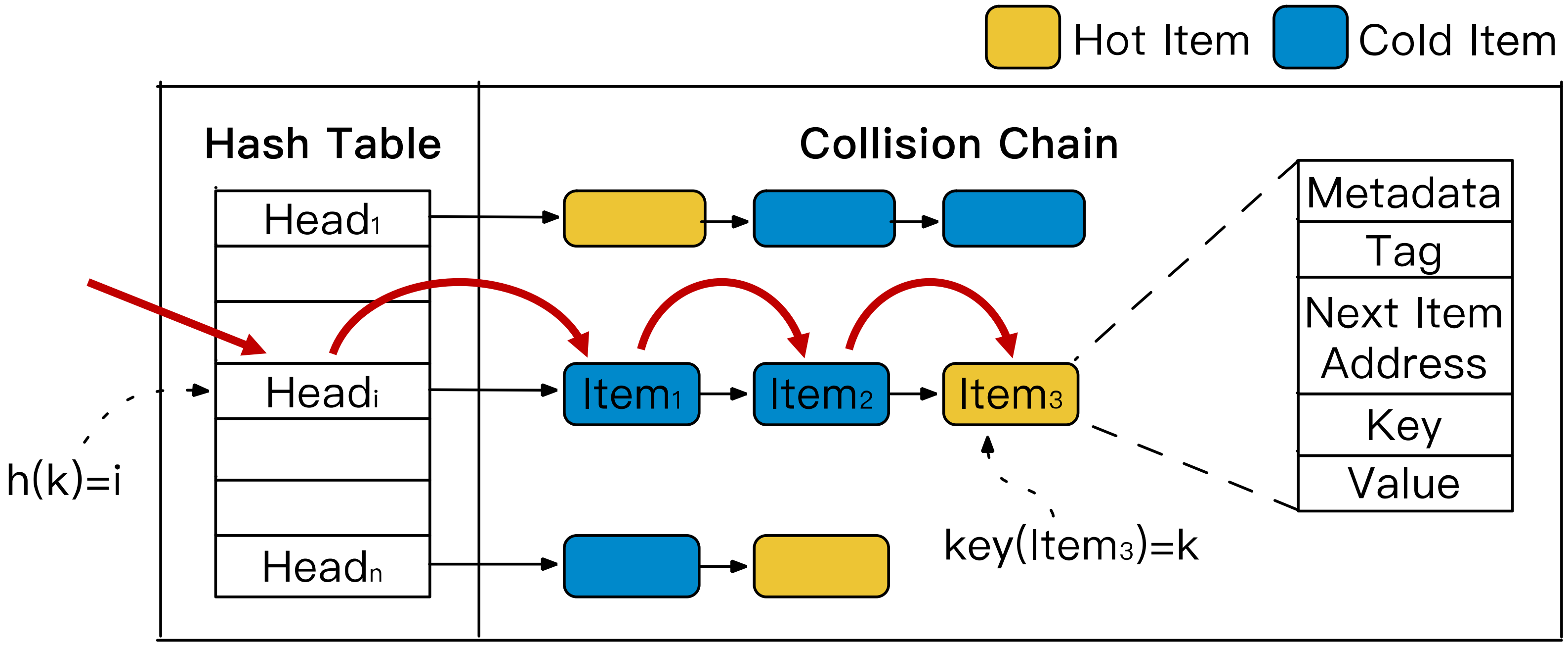


Solution to mitigate the hotspot issue

- Scale out using consistent hashing
  - Reduce the resource utilization of a single node
- Replication in multi-node
  - Large system and storage overhead
- Front-end cache
  - Inefficient for write-intensive data
- Improve single node's ability to handle hotspots

Prior work: Chain-based hash index

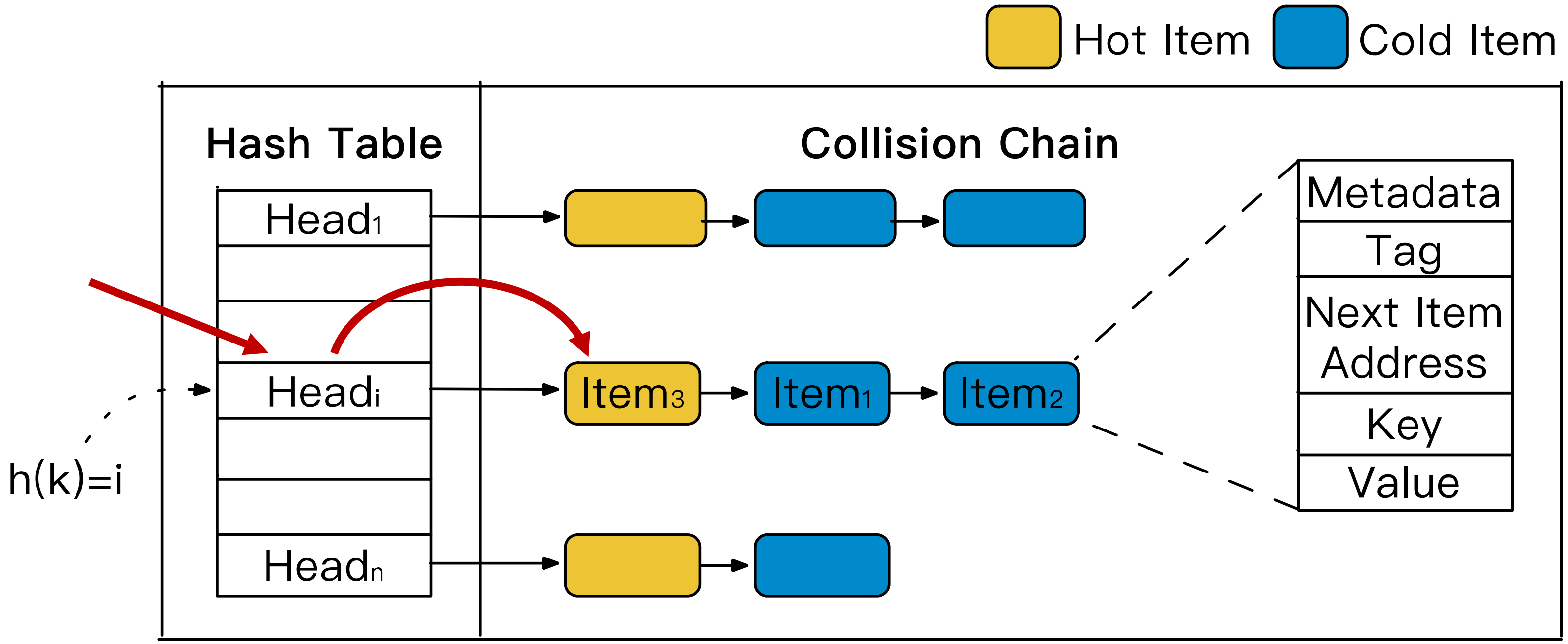
- Hot items are randomly placed in the collision chain

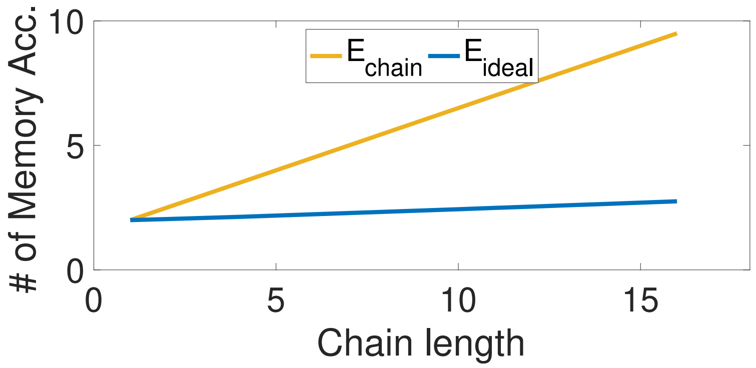


4 memory accesses to find item$_3$, this is not optimal

Ideal: Hotspot-aware hash index

- Memory accesses required to retrieve an item should be (negatively) correlated to this hotness

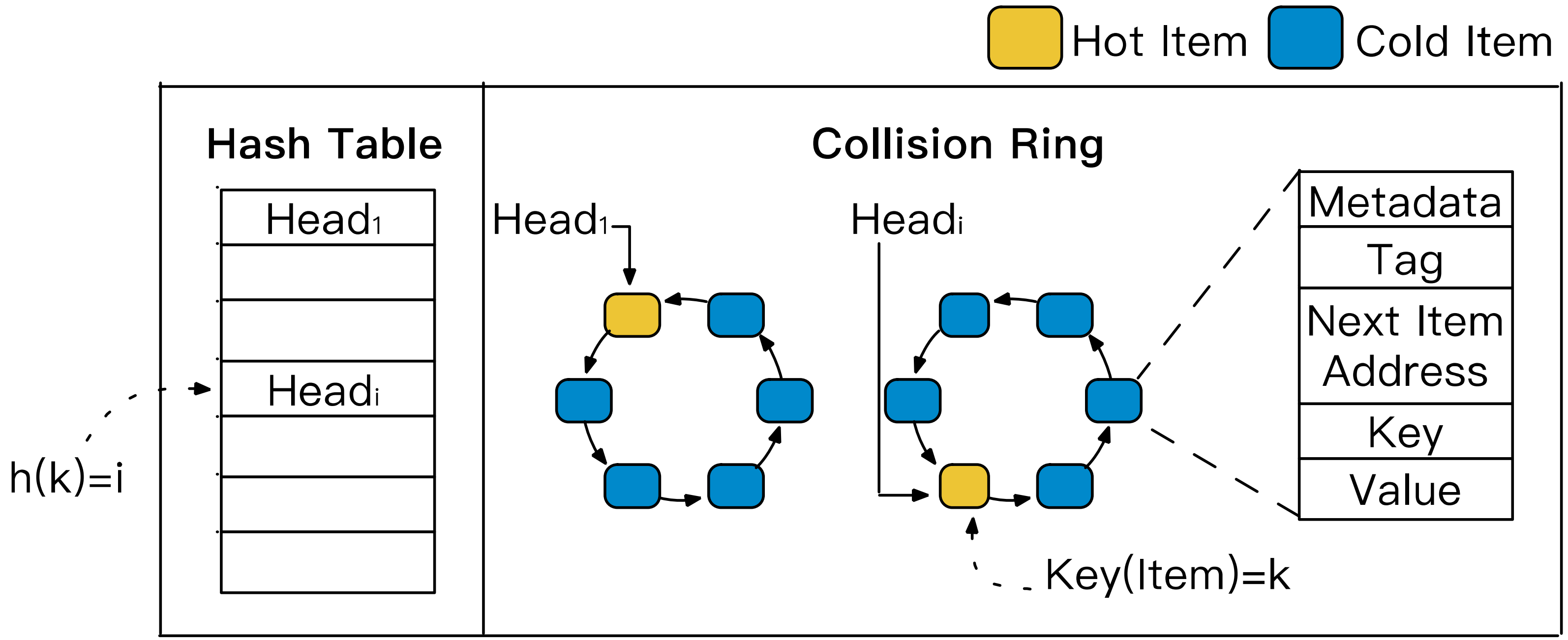




Ensuring dynamic hotspot shift and lock-free access is a challenge

HotRing: Ordered-ring hash index

- The head pointer can point to any items dynamically

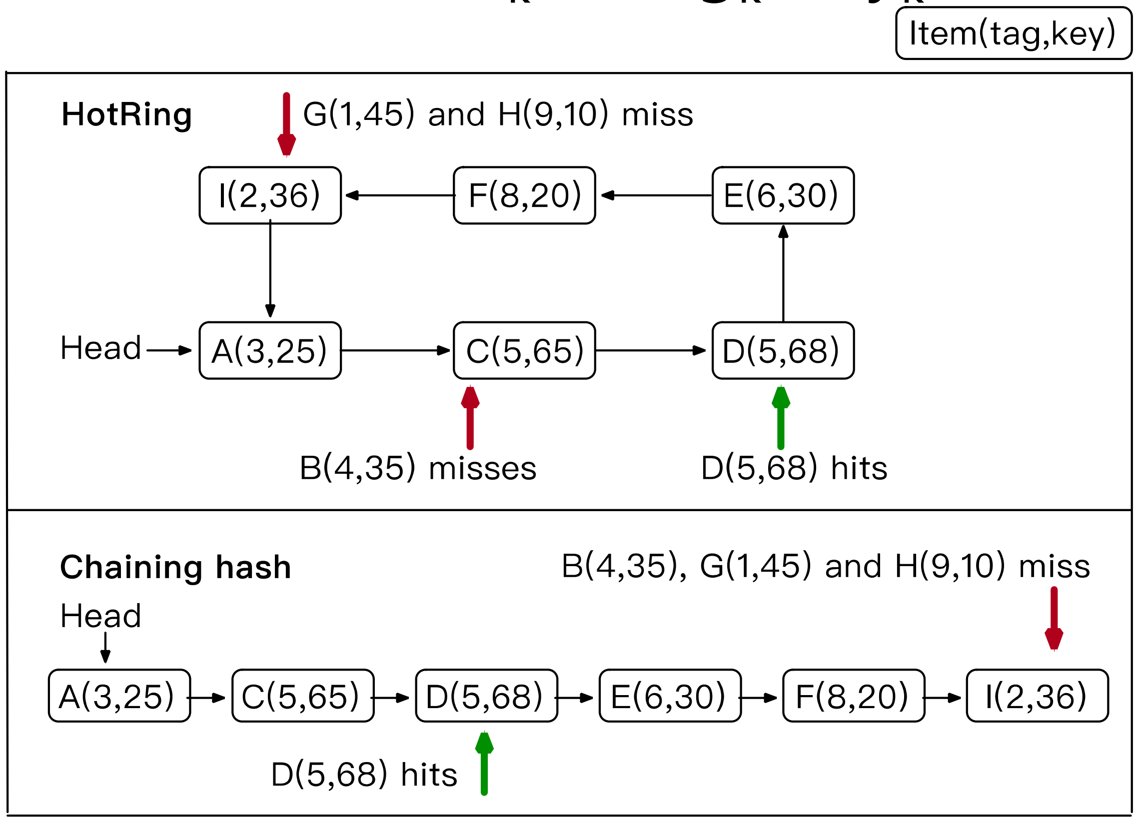


HotRing design

- #1: Why an ordered-ring structure is needed?
- #2: How to identify hotspots and adjust head pointer?
- #3: How to guarantee lock-free concurrent operations?
- #4: How to rehash to adapt to hotspot volumes increase?

#1: Ordered-ring structure

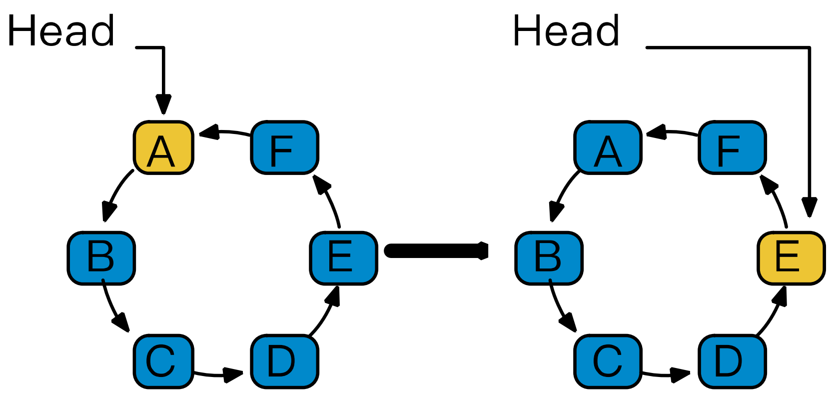
- To determine the termination of lookup processes
  - Define order of item k: $order_k = (tag_k, key_k)$



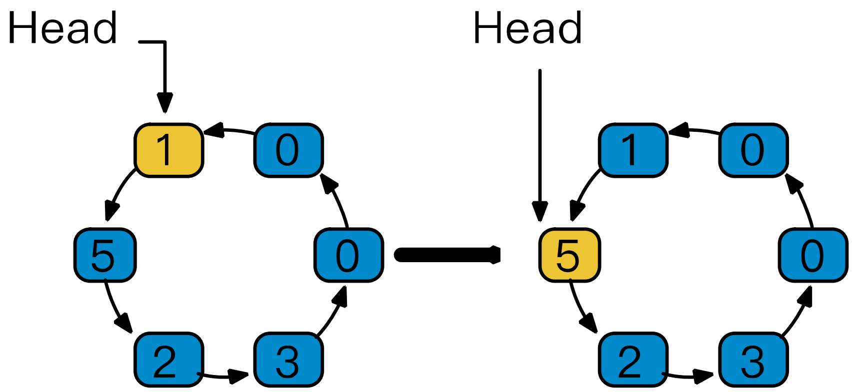
#2: Identify hotspots and adjust head pointer

- The head pointer is periodically moved to a potential hotspot from the strategy

- Random Movement Strategy
  - ✓ After *R* requests, moving the head pointer to the *R*-th access items

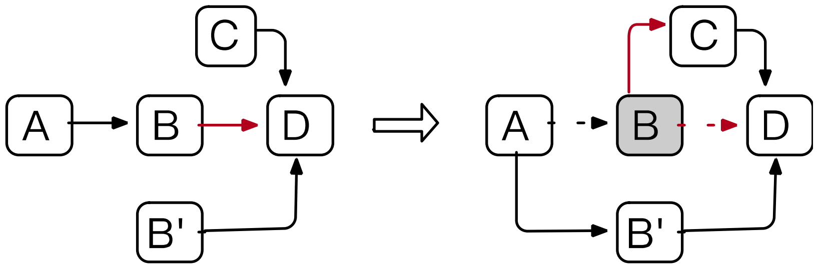


- Statistical Sampling Strategy
  - ✓ After *R* requests, launching a new round of sampling to find the best position

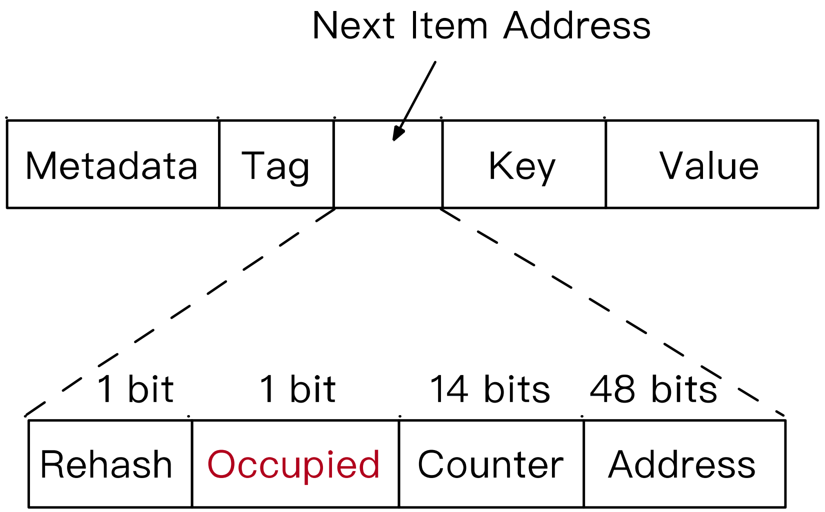


#3: Lock-free concurrent operations

- HotRing has a complete set of lock-free designs, which has been rigorously introduced by previous work[1,2]



concurrency issue

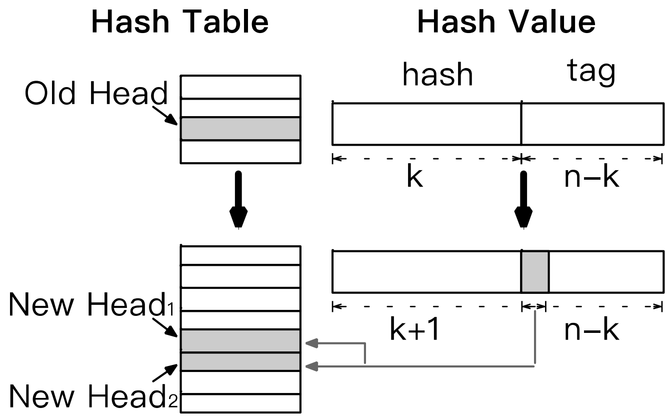


Item format

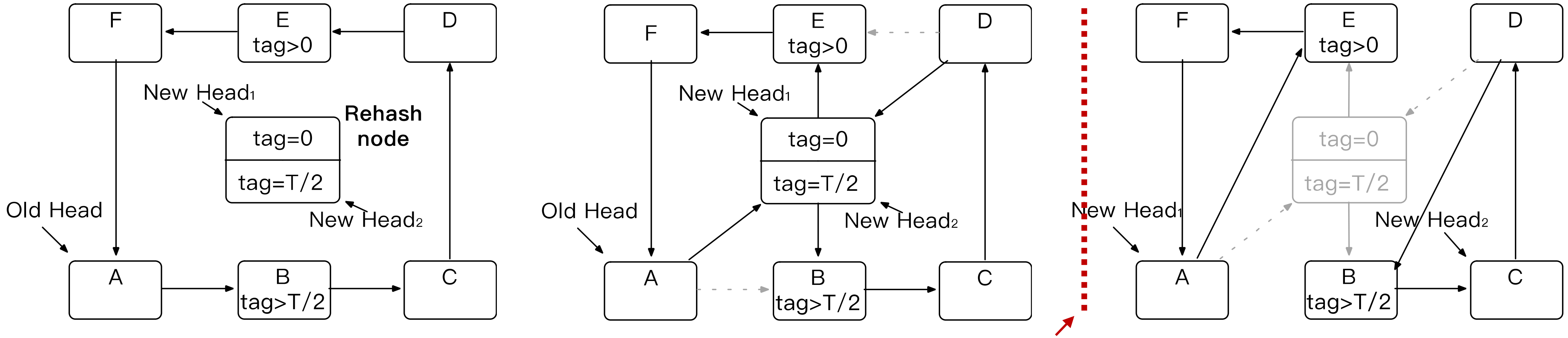
[1] John D Valois. Lock-free linked lists using compare-and-swap. (PODC 1995)
[2] Timothy L Harris. A Pragmatic Implementation of Non-blocking Linked-lists. (DISC 2001)

#4: Lock-free rehash adapts to hotspot volume

- **Access overhead** instead of Load factor
  - ➢ average number of memory accesses to retrieve an item



- Three steps to rehash



Initialization　　Split　　Transition period　　Deletion

Performance evaluation

- Experiment setting
  - Environment

CPU	Memory
2.50GHz Intel Xeon(R) E5-2682 v4 * 2 L2 cache 256KB (512 * 8 way) L3 cache 40MB (32768 * 20 way)	32GB 2133MHz DDR4 DRAM * 8

  - YCSB core workloads, except workload E (scan operations)

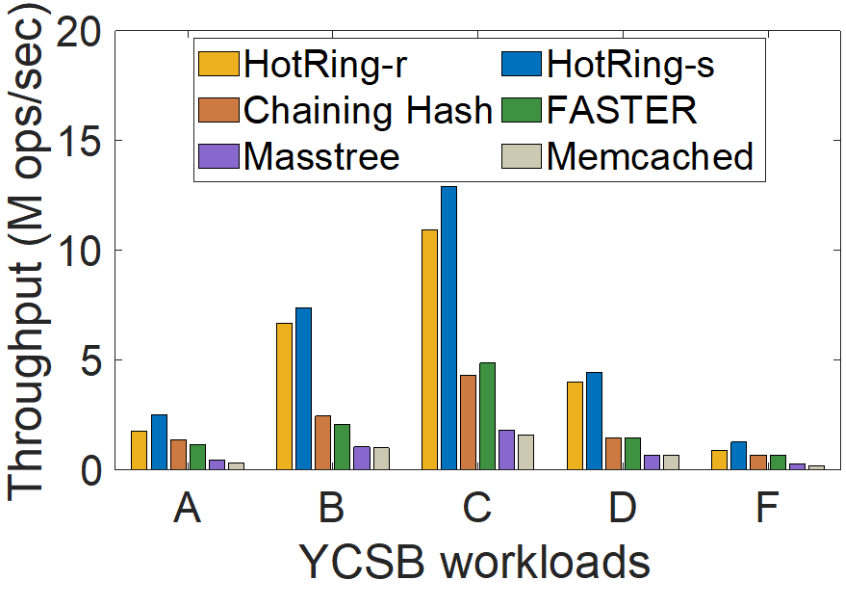
Key size	Value size	# of Loaded keys	Zipfian θ	key-bucket ratio	# of thread
8 bytes	8 bytes	250 millions	1.22	8	64

Performance evaluation

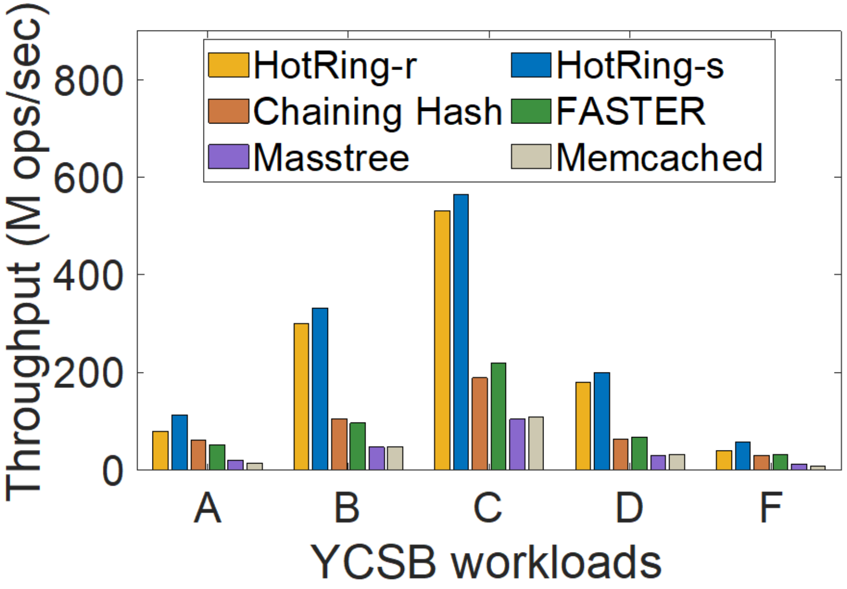
- Deployment
  - HotRing-r
    - random movement strategy
  - HotRing-s
    - sampling statistics strategy
- Baselines
  - Chaining Hash
    - lock-free chain-based hash index, that is modified based on Memcached
  - FASTER *(SIGMOD 2018)*
  - Masstree
  - Memcached

YCSB benchmarks

- YCSB benchmarks



(a) Single thread

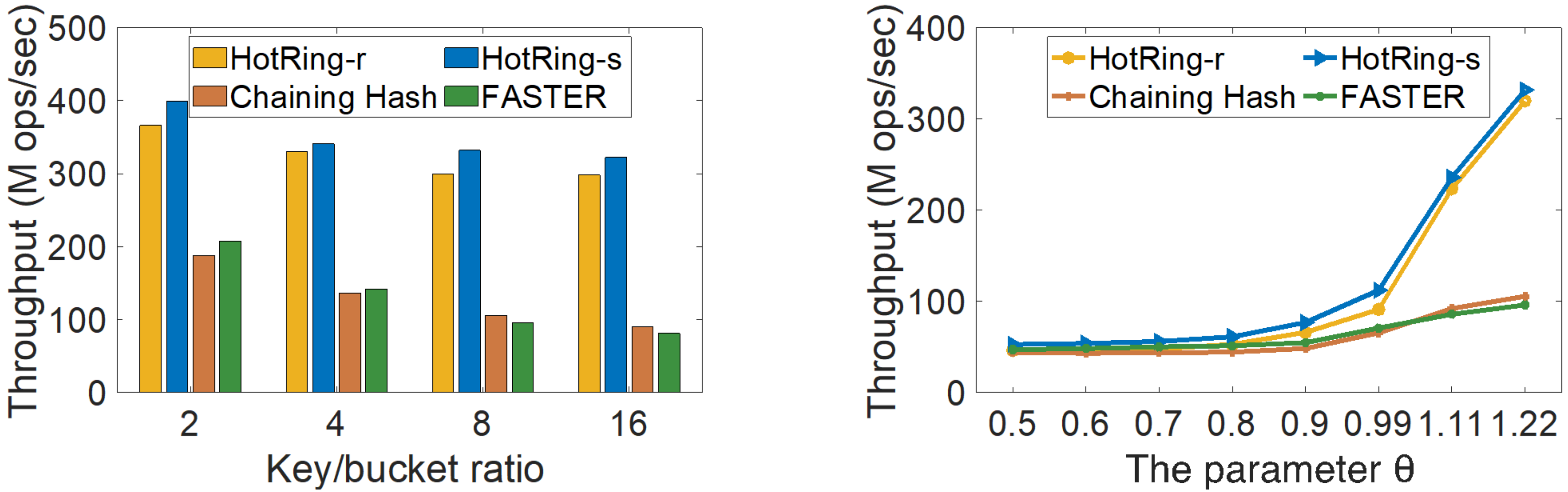


(b) 64 threads

HotRing achieves better performance in throughput under all workloads, especially for workloads B (95% read) and C (100% read).

Micro-benchmarks

- Chain length & Access skewness

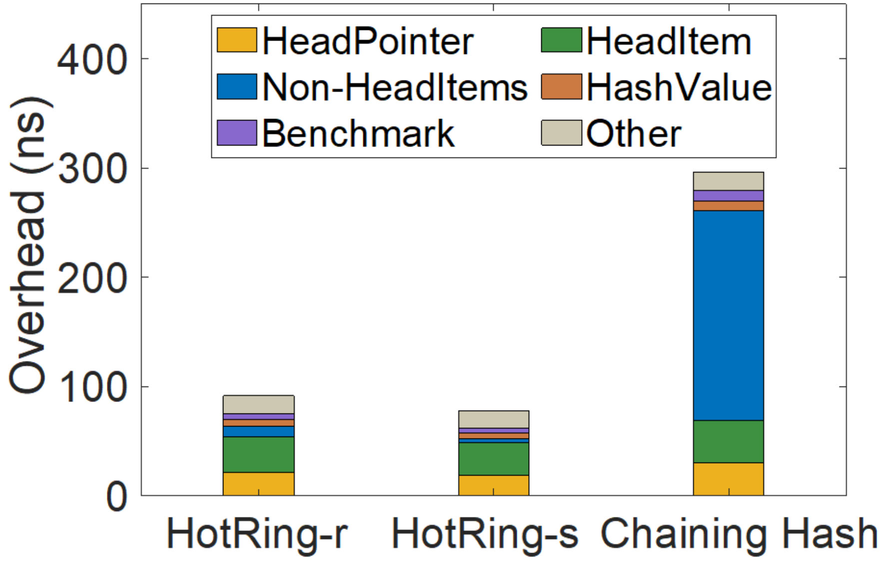


(a) Impact of chain length (b) Impact of Zipfian parameter θ

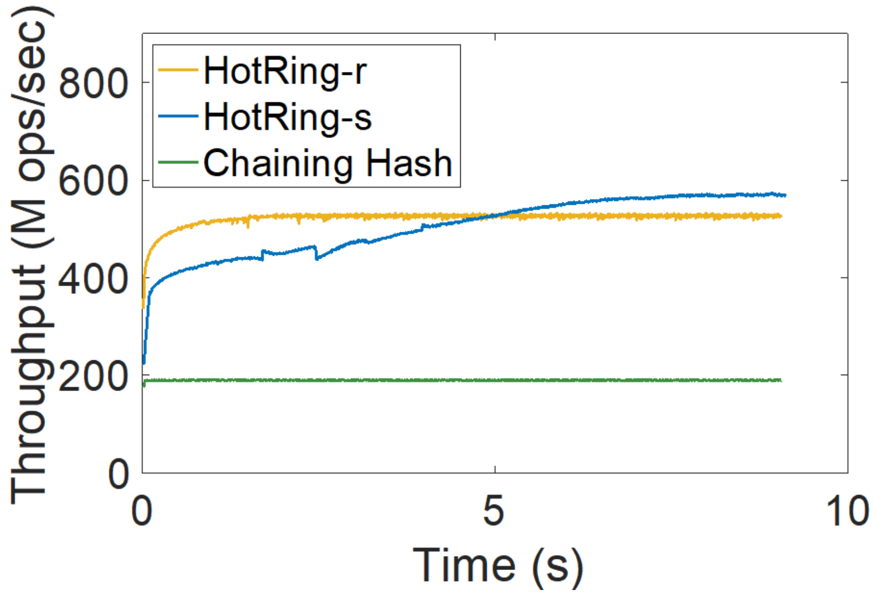
(a) HotRing retains satisfactory performance even for long chains.
(b) HotRing achieves significantly performance improves as θ increases.

Micro-benchmarks

- Break-down cost & Reaction delay



(a) Break-down cost

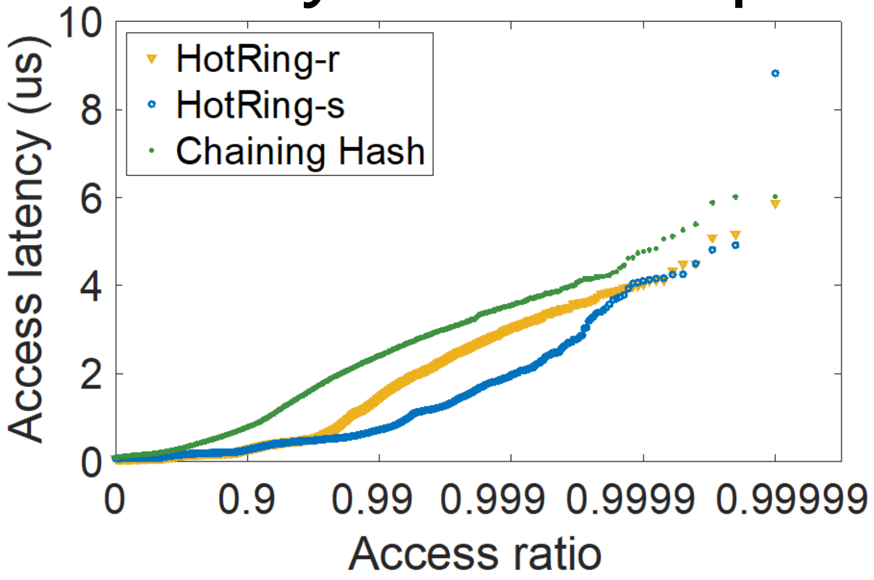


(b) Reaction delay

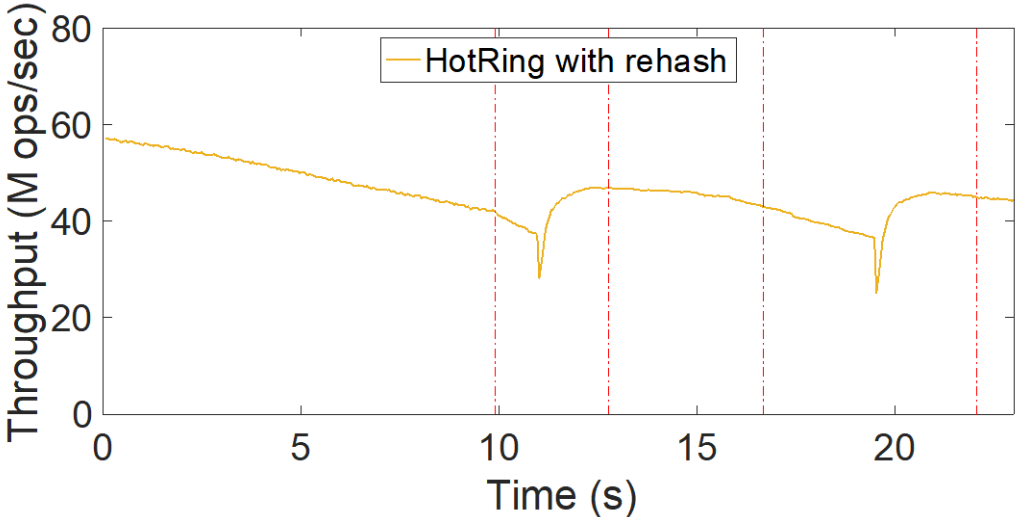
(a) HotRing-s greatly reduces the overhead ratio of Non-HeadItems.
(b) HotRing-r enables faster response after hotspots have shifted.

Micro-benchmarks

- Tail latency & Rehash performance



(a) Tail latency



(b) Two lock-free rehash processes

(a) The 99.999-percentile response time is less than 6us.

(b) Two consecutive rehash operations help to retain the throughput as data volume continuously grows.

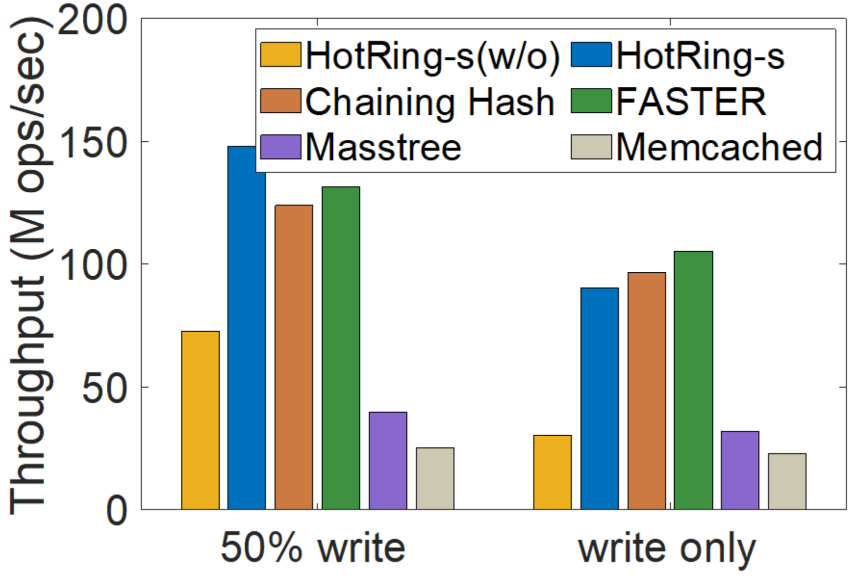
Conclusions

- HotRing is optimized for massively concurrent accesses to a small portion of items.
- HotRing dynamically adapts to the shift of hotspots by pointing bucket heads to frequently accessed items.
- HotRing can retrieve hot items within two memory accesses, and provides near-perfect throughput.

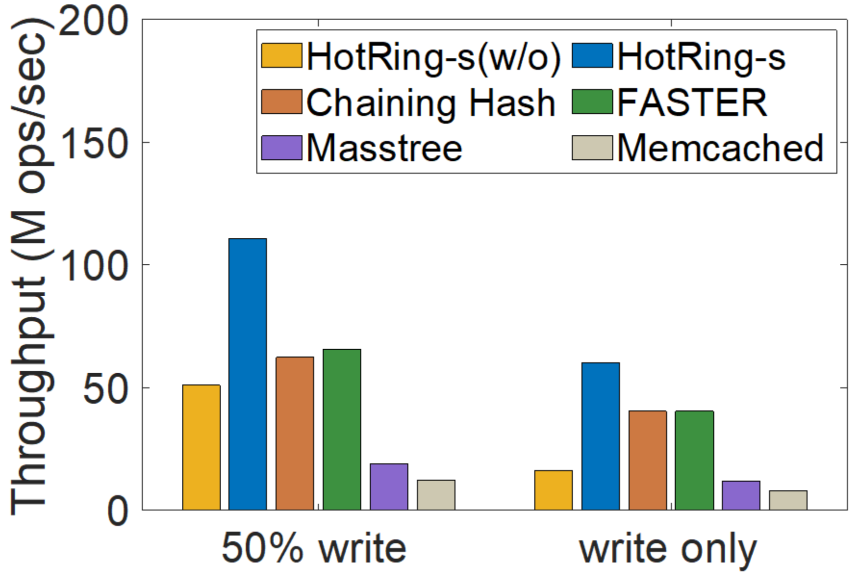
Thank you

Appendix: Micro-benchmarks

- Read-copy-update performance (100-byte value)



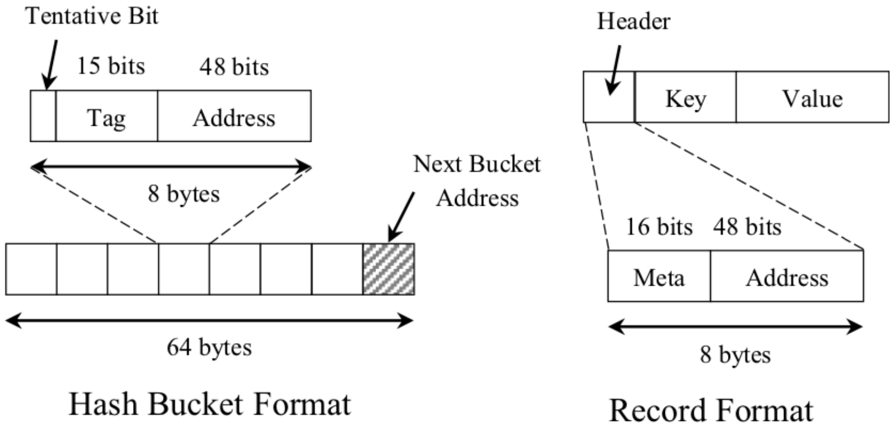
(a) Key-bucket ratio: 2

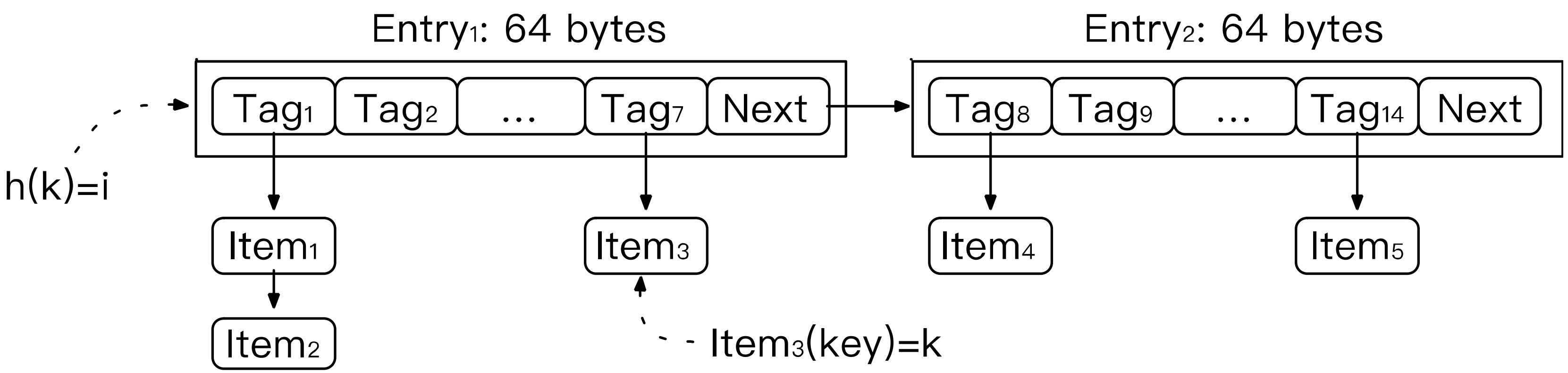


(b) Key-bucket ratio: 8

FASTER: cache affinity to alleviate hot spots

- Chain-based hash index
  - Hash Table
  - Collision Chaining
  - Entry: improve cache affinity





Cache affinity not suit for large-scale data set

the processor cache is only 32MB, while the memory capacity can reach 256GB