MemC3: MemCache with CLOCK and Concurrent Cuckoo Hashing

Bin Fan (CMU),

Dave Andersen (CMU), Michael Kaminsky (Intel Labs)

NSDI 2013

Goal: Improve Memcached

1. Reduce space overhead (bytes/key)

2. Improve performance (queries/sec)

Background

Overview

- Previous Work: Sharding
  - Avoid inter-thread synchronization
    - e.g., dedicated cores [Berezecki11]
  - Hotspot? Memory Efficiency?

- Our Approach: Algorithm Engineering
  - Apply concurrent / space-efficient data structures

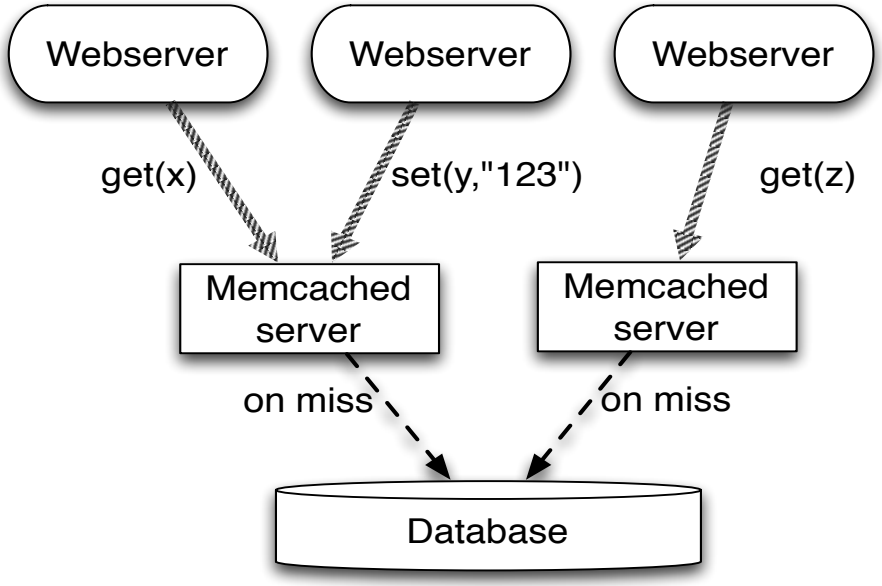
MemC3 vs. Memcached
3x throughput
30% more small key-value items

Background

Memcached Overview

- A DRAM-based key-value store
  - `GET(key)`
  - `SET(key, value)`

- LRU eviction for high hit rate
- Typical use:
  - Speed up webservers
  - Alleviate db load



Background

Typical Workloads

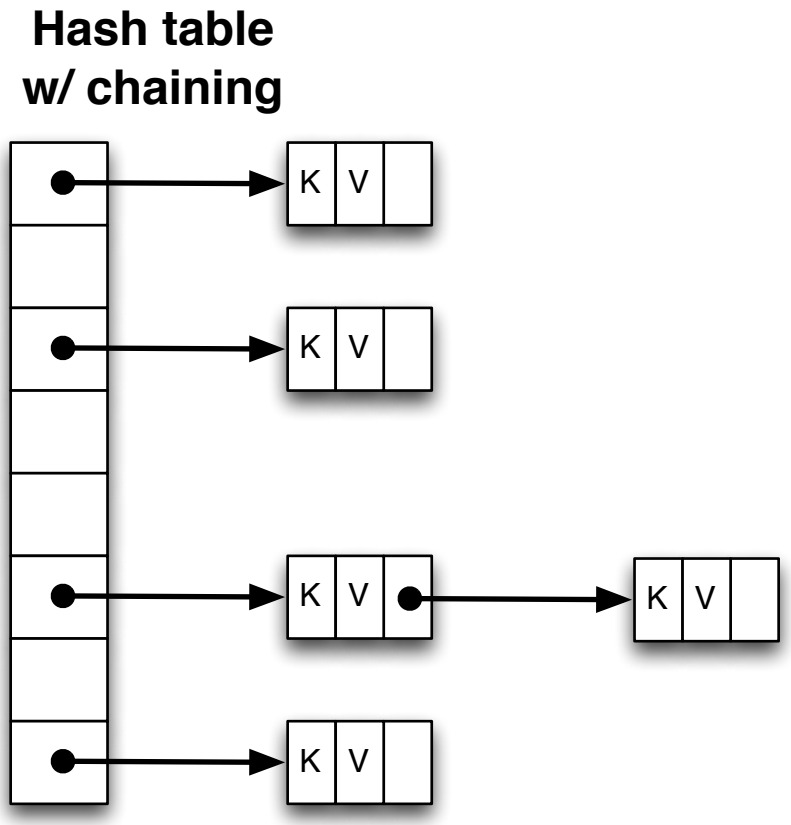
- Watch the next talk!
  - Often used for small objects (Facebook[Atikoglu12])
    - 90% keys < 31 bytes
    - Some apps only use 2-byte values
  - Tens of millions of queries per second for large memcached clusters (Facebook[Nishtala13])

Small Objects, High Rate

Background

Memcached: Core Data Structures

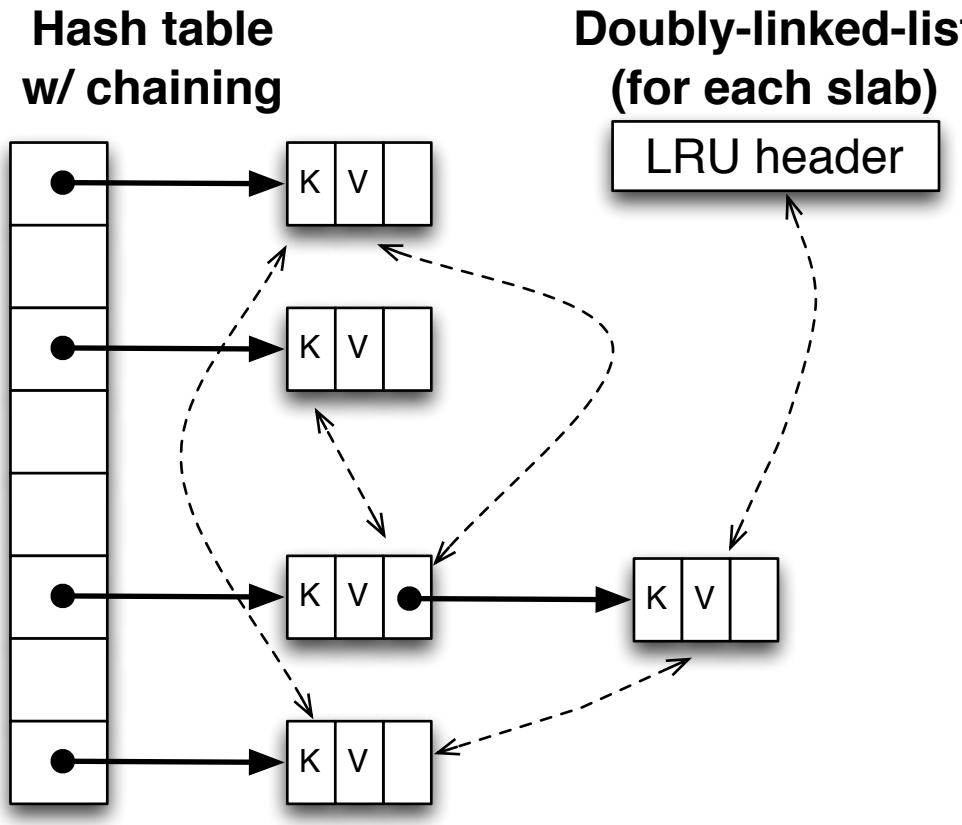
- **Key-Value Index**:
  - Chaining hash table



Background

Memcached: Core Data Structures

- **Key-Value Index**:
  - Chaining hash table
- **LRU Eviction**:
  - Doubly-linked lists



Problems We Solve

- Single-node scalability
  - Accessing hash table and updating LRU are serialized
- Space overhead
  - 56-byte header per object
    - Including 3 pointers and 1 refcount
    - For a 100B object, overhead > 50%

Solutions

Optimistic cuckoo hashing

- Better memory efficiency: 95% table occupancy
- Higher concurrency: single-writer/multi-reader

focus of this talk

CLOCK-based LRU eviction

- Better space efficiency and concurrency

Additional algo & tuning improvements

described in paper

Solutions

Optimistic cuckoo hashing

- Better memory efficiency: 95% table occupancy
- Higher concurrency: single-writer/multi-reader

focus of this talk

CLOCK-based LRU eviction

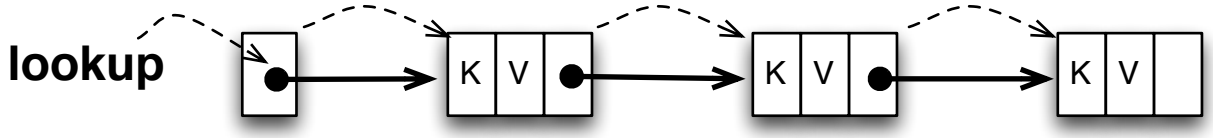
- Better space efficiency and concurrency

Additional algo & tuning improvements

Problem

Memcached Default Hash Table

- Chaining items hashed in same bucket:



Good: simple (Data Structure 101)

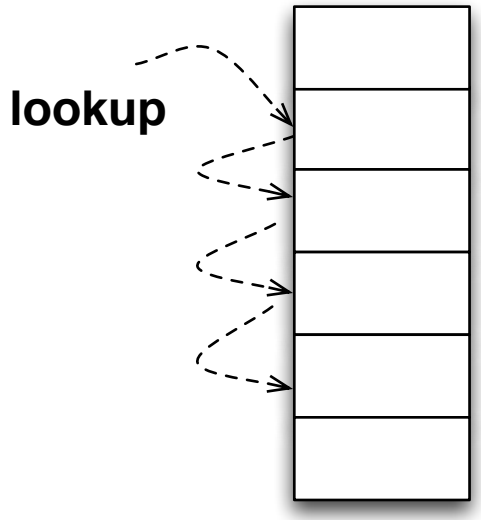
Bad: low cache locality:
(dependent pointer dereference)

Bad: pointer costs space

Problem

Linear Probing

- Probing consecutive buckets for vacancy



Good: simple

Good: cache friendly

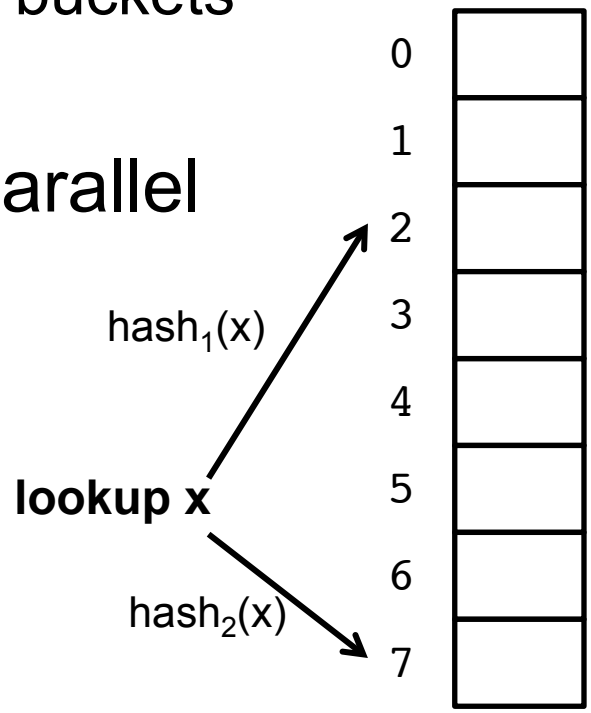
Bad: poor memory efficiency:

(if occupancy > 50%, lookup needs to search a long chain)

Solution

Cuckoo Hashing[Pagh04]

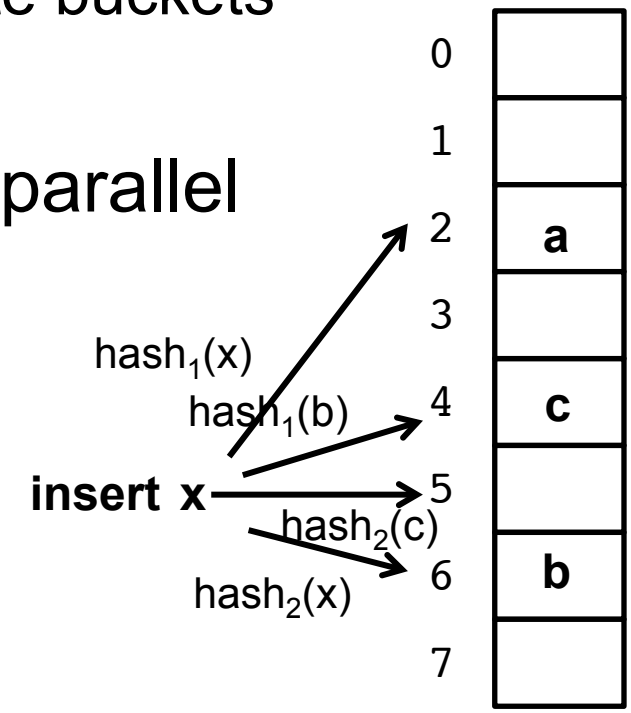
- Each key has two candidate buckets
  - Assigned by $hash_1(key)$, $hash_2(key)$
  - Stored in one of its candidate buckets
- Lookup: read 2 buckets in parallel



Solution

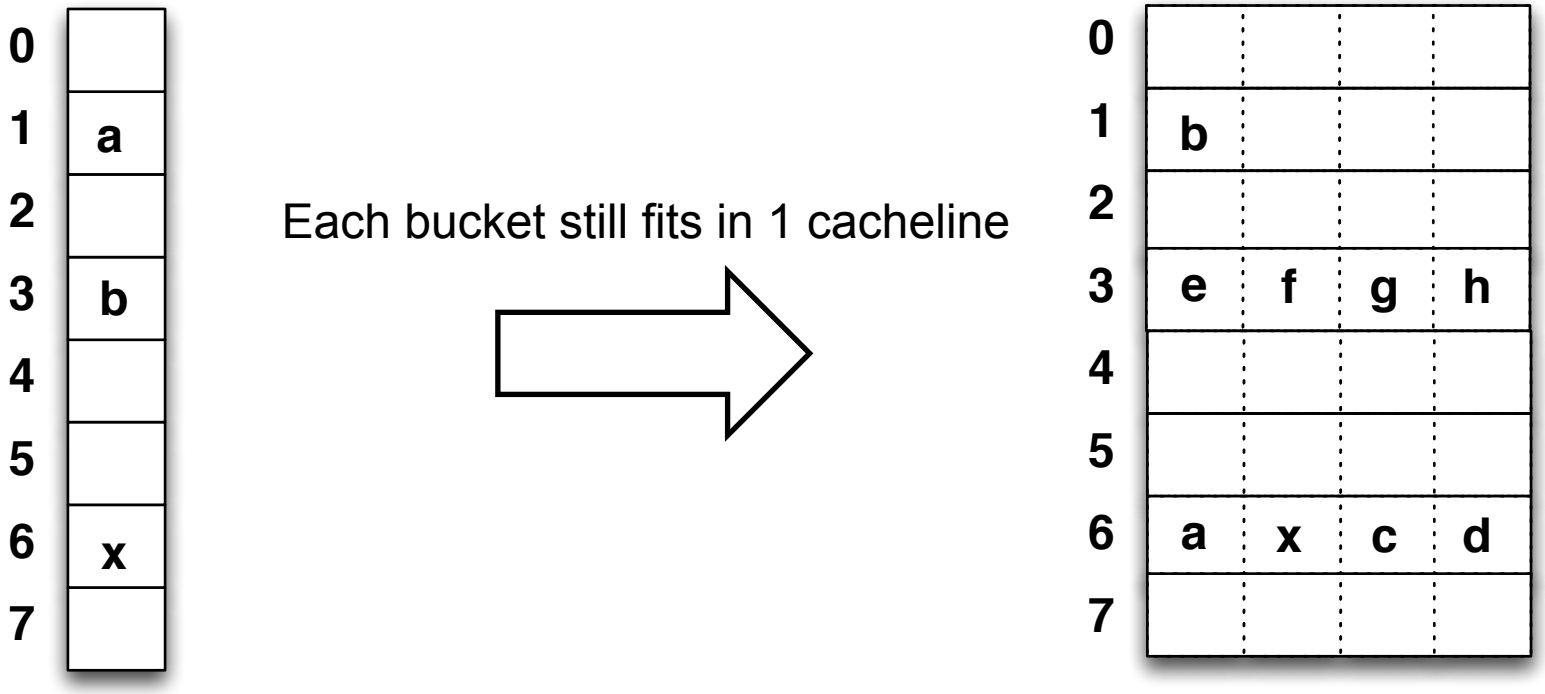
Cuckoo Hashing[Pagh04]

- Each key has two candidate buckets
  - Assigned by $hash_1(key)$, $hash_2(key)$
  - Stored in one of its candidate buckets
- Lookup: read 2 buckets in parallel
- Insert:
  - Perform key displacement recursively
  - Still O(1) on average [Pagh04]



Solution

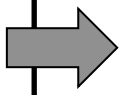
Increase Set-Associativity



- 2 cacheline-sized reads per lookup
- **50%** space utilized

- 2 cacheline-sized reads per lookup
- **95%** space utilized!

Solutions



Optimistic cuckoo hashing

- Better memory efficiency: 95% table occupancy
- Higher concurrency: single-writer/multi-reader

focus of this talk

CLOCK-based LRU eviction

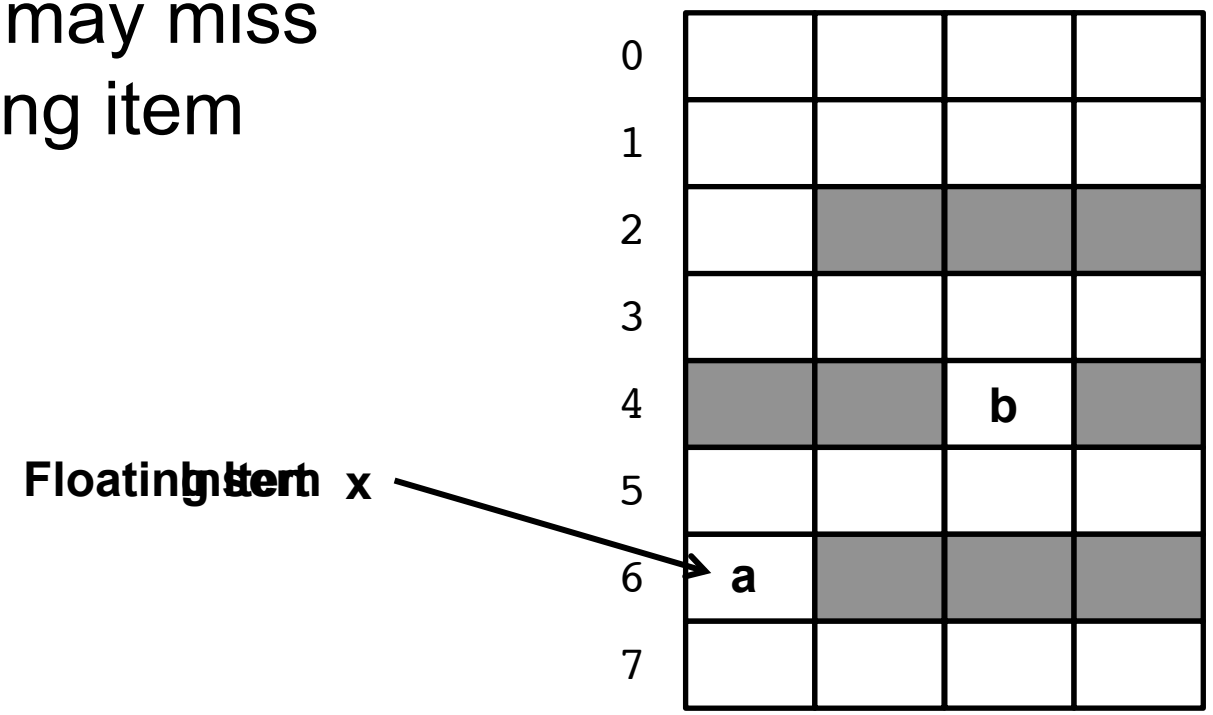
- Better space efficiency and concurrency

Additional algo & tuning improvements

Problem

False Miss Problem

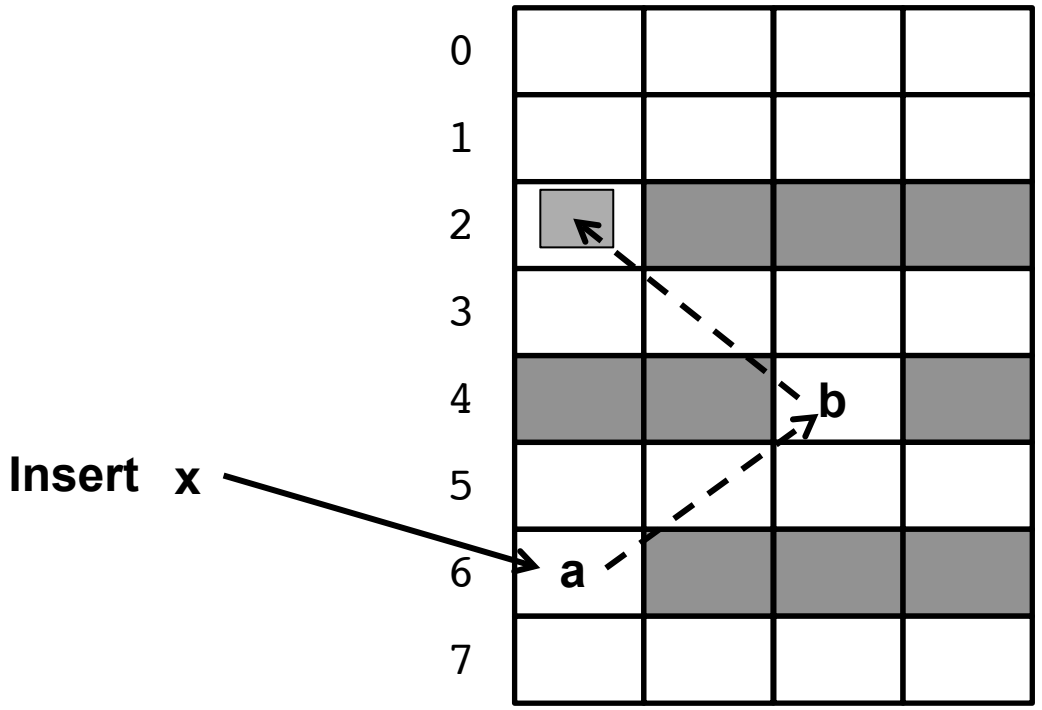
- During insertion:
  - always a “floating” item during insertion
  - a reader may miss this floating item



Solution

Our Solution: 2-Step Insert

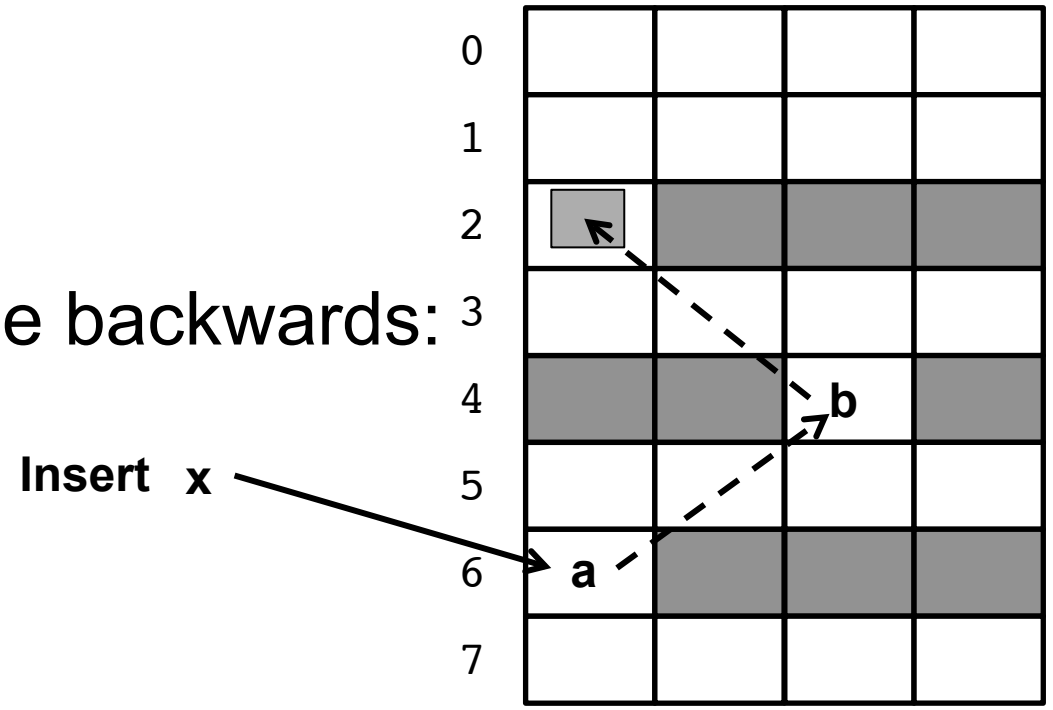
- Step1: Find a cuckoo path to an empty slot without editing buckets



Solution

Our Solution: 2-Step Insert

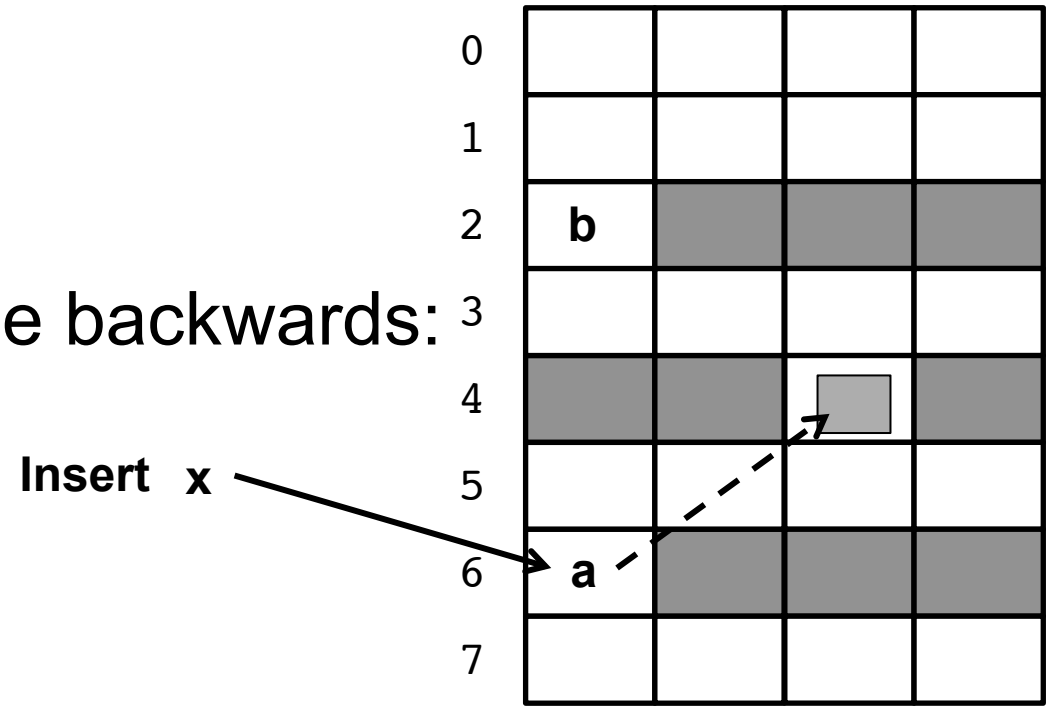
- Step1: Find a cuckoo path to an empty slot without editing buckets
- Step2: Move hole backwards:



Solution

Our Solution: 2-Step Insert

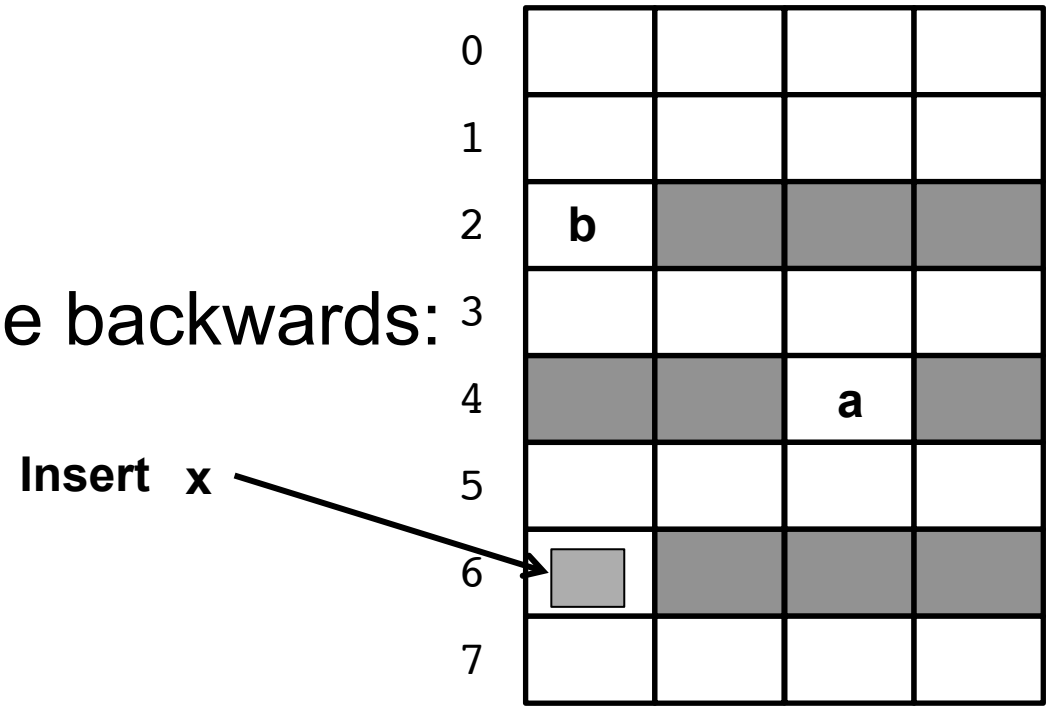
- Step1: Find a cuckoo path to an empty slot without editing buckets
- Step2: Move hole backwards:



Solution

Our Solution: 2-Step Insert

- Step1: Find a cuckoo path to an empty slot without editing buckets
- Step2: Move hole backwards:

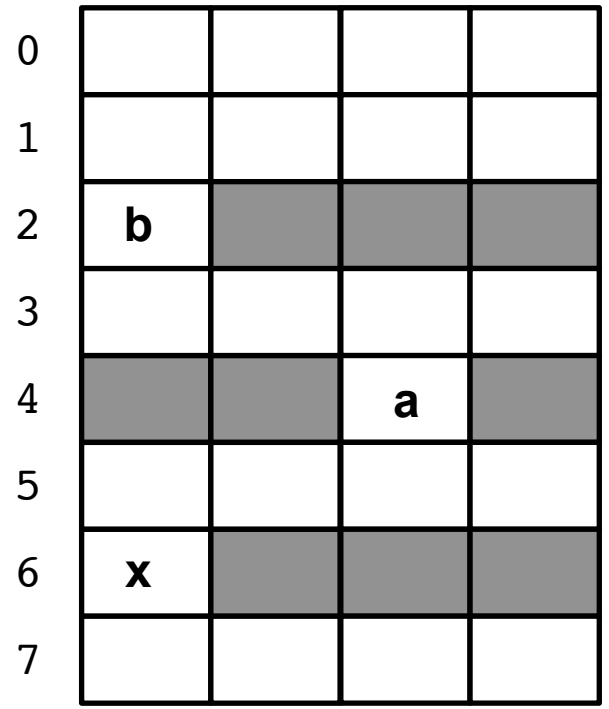


Solution

Our Solution: 2-Step Insert

- Step1: Find a cuckoo path to an empty slot without editing buckets

- Step2: Move hole backwards:



Only need to ensure each move is atomic w.r.t. reader

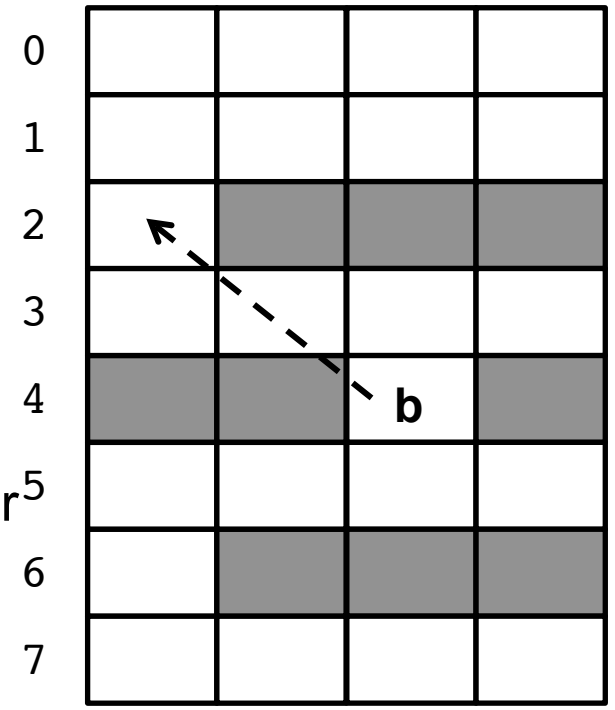
How to Ensure Atomic Move

- e.g., move key “b” from bucket 4 to bucket 2

- A simple implementation:

  Lock bucket 2 and 4
  Move key
  Unlock bucket 2 and 4

- Our approach: Optimistic locking
  - Optimized for read-heavy workloads
  - Each key mapped to a version counter
  - Reader detects version change (described in paper)



Problem

How to Coordinate Writers

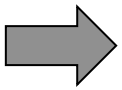
- Simple (current) solution:
  - Serialize inserts
  - Works fine with read-heavy workload
- Ongoing work: allow multiple writers

Solutions

Optimistic cuckoo hashing

- Better memory efficiency: 95% table occupancy
- Higher concurrency: single-writer/multi-reader

focus of this talk



CLOCK-based LRU eviction

- Better space efficiency and concurrency

2ptr/key => 1bit/key, concurrent update

Additional algo & tuning improvements

Solutions

Optimistic cuckoo hashing

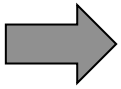
- Better memory efficiency: 95% table occupancy
- Higher concurrency: single-writer/multi-reader

focus of this talk

CLOCK-based LRU eviction

- Better space efficiency and concurrency

2ptr/key => 1bit/key, concurrent update



Additional algo & tuning improvements

Avoid unnecessary full-key comparisons on hash collision

Problems We Solve

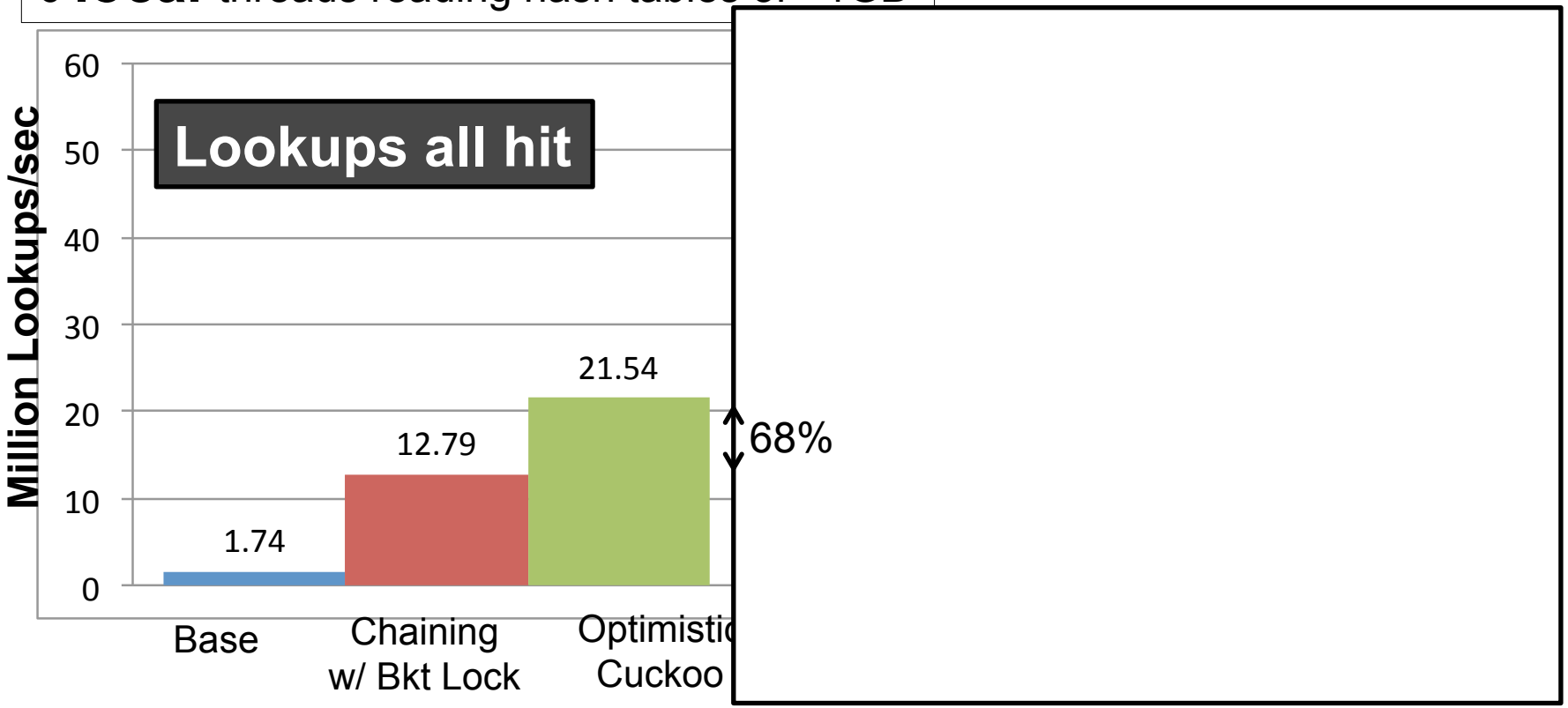
- Single-node scalability
  - ~~Accessing hash table and updating LRU are serialized~~
  - GET requires no mutex
  - Single-writer/multiple-reader

- Space overhead
  - ~~56-byte header per object~~
  - 3 pointers + 1 refcount => 1 pointer + 1 refbit

Hash Table Microbenchmark

Server: Low Power Xeon CPU w/ 12 cores, 12 MB L3 cache

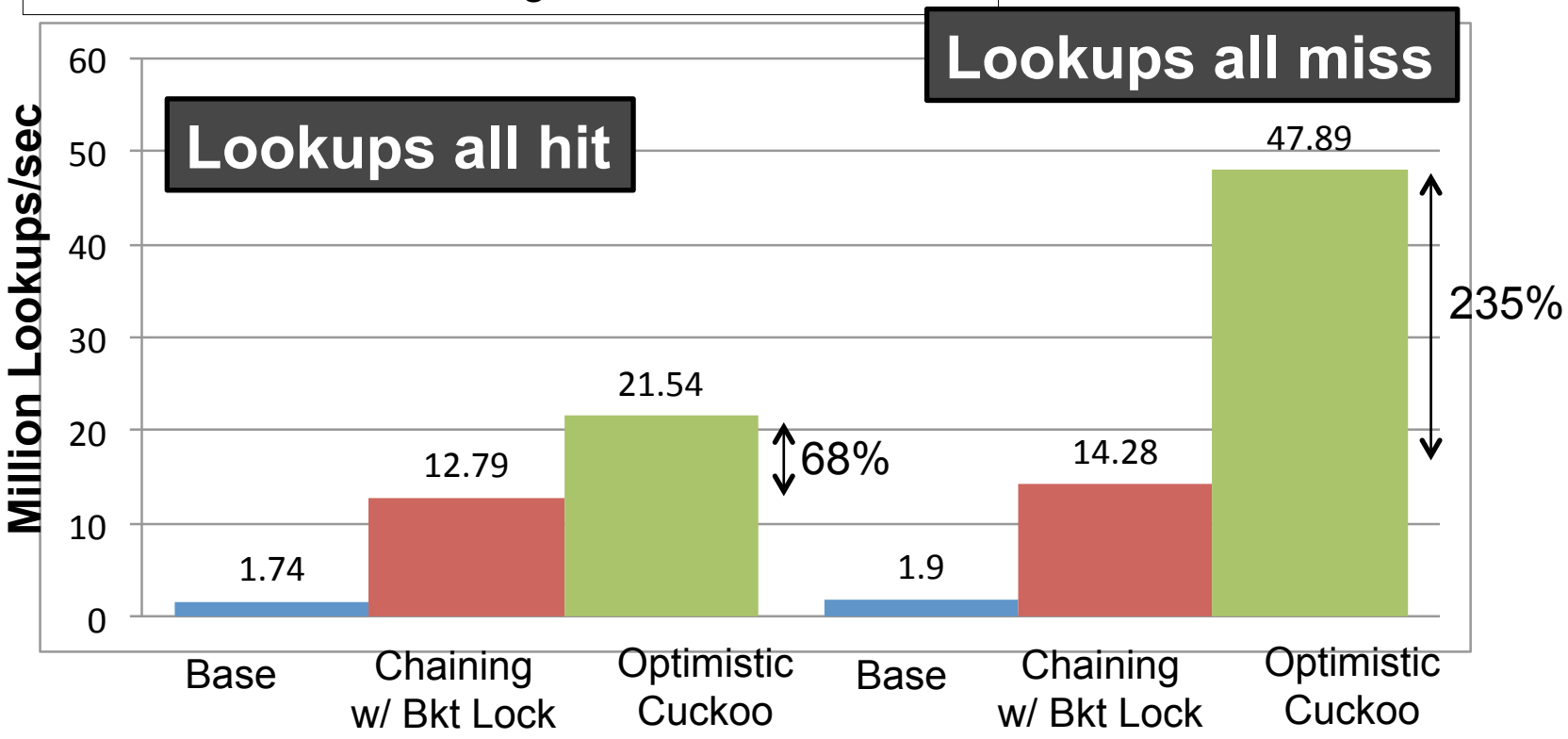
6 **local** threads reading hash tables of ~1GB



Hash Table Microbenchmark

Server: Low Power Xeon CPU w/ 12 cores, 12 MB L3 cache

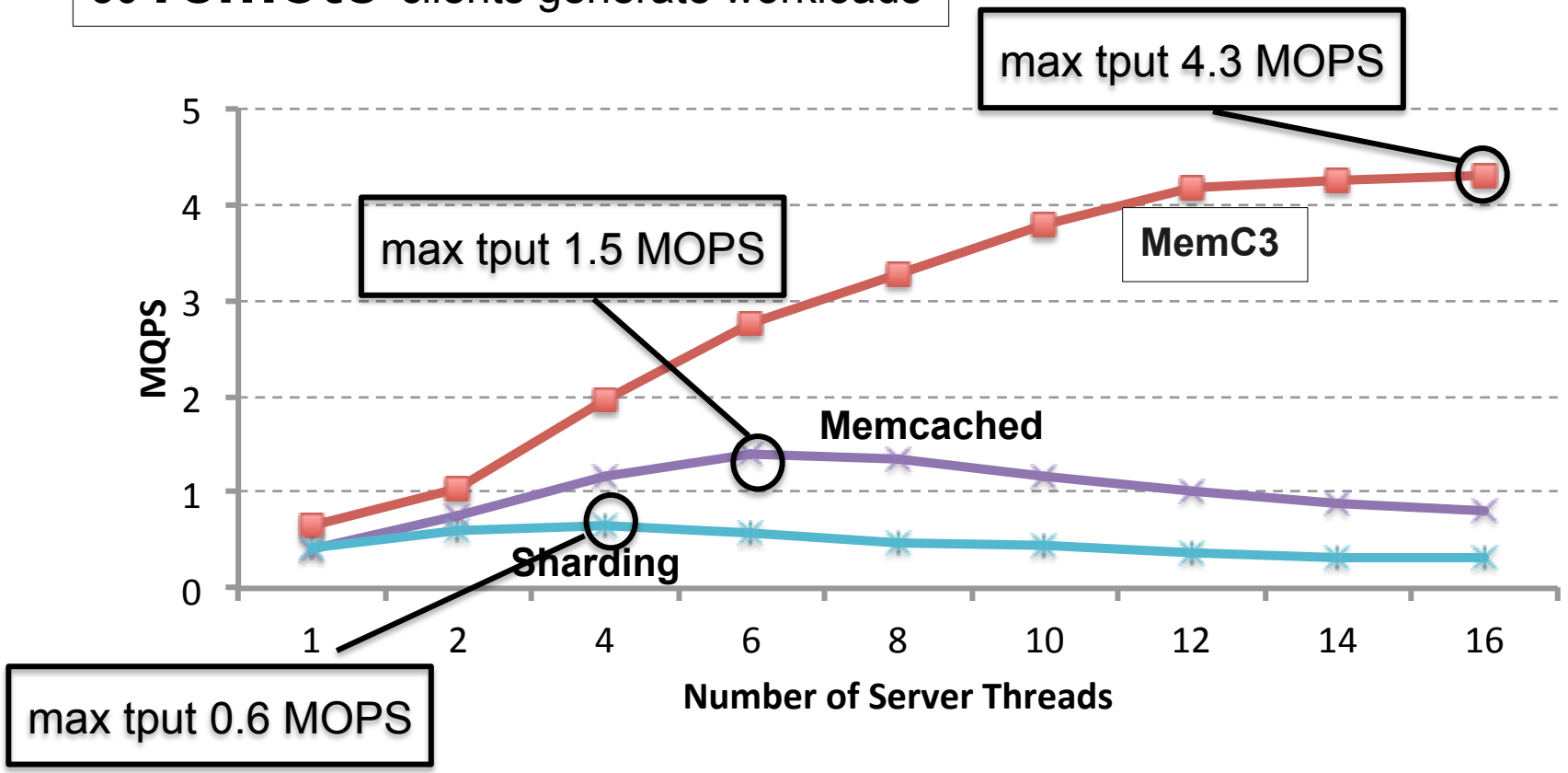
6 **local** threads reading hash tables of ~1GB



End-to-end Performance

16-Byte key, 32-Byte Value, 95% GET, 5% SET, **zipf** distributed

50 **remote** clients generate workloads



Conclusion

- Optimistic cuckoo hashing
  - High space efficiency
  - Optimized for read-heavy workloads
  - Source Code available:
    github.com/efficient/libcuckoo

- MemC3 improves Memcached
  - 3x throughput, 30% more (small) objects
  - Optimistic Cuckoo Hashing, CLOCK, other system tuning

References

[Atikoglu12] Workload analysis of a large-scale key- value store.

[Berezecki11] Many-core key-value store

[Nishtala13] Scaling Memcache at Facebook

[Pagh04] Cuckoo hashing

Q & A

- Thanks!