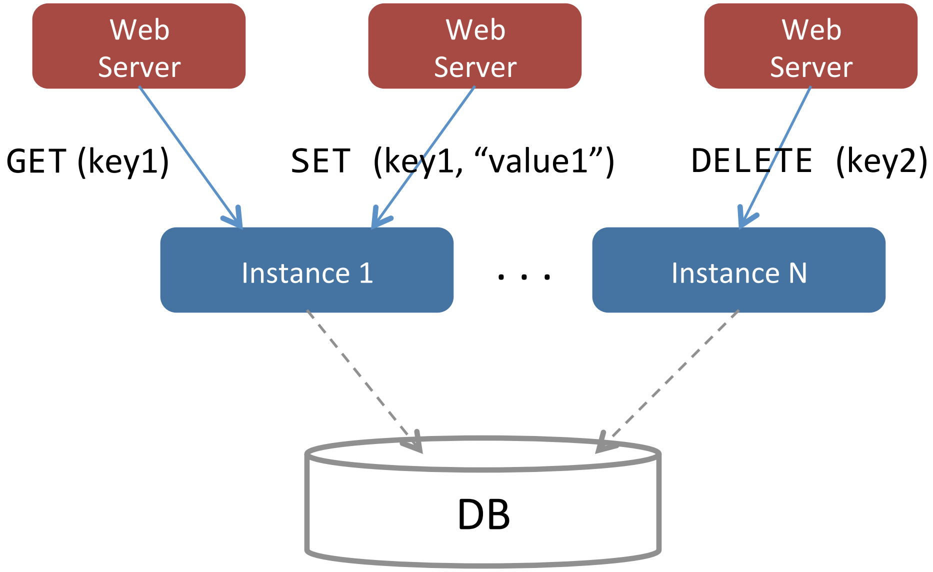
Mercury: Bringing Efficiency to Key-value Stores

Rohan Gandhi, Y. Charlie Hu
Purdue University

Aayush Gupta, Anna Povzner,
Wendy Belluomini, Tim Kaldewey
IBM Research – Almaden

Overview:
In-memory Key-Value Stores

- Store data in DRAM
- Alleviate database load



Memcached — http://memcached.org/

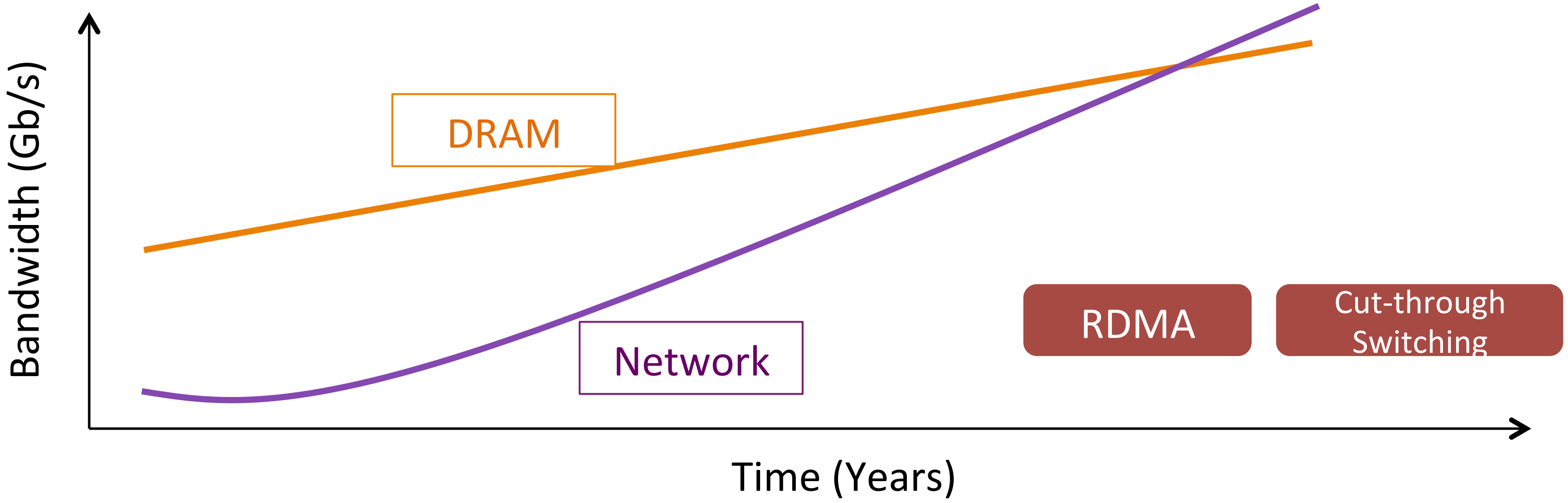
MemC3 — NSDI’ 13

MassTree — EuroSys’ 12

Request Latency = Latency(Network) + Latency(Server)

Bottleneck

Memory Efficiency Matters!



Bottlenecks shifting from network to DRAM !

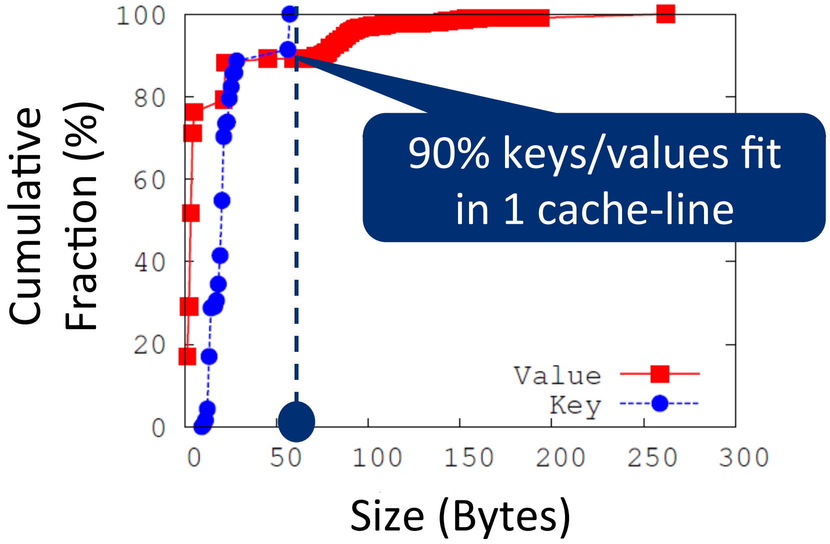
Request Latency = Latency(Network) + Latency(Server)

Bottleneck

But we have CPU caches (L1:L3) to alleviate DRAM performance !

Is it true with key-value stores ?

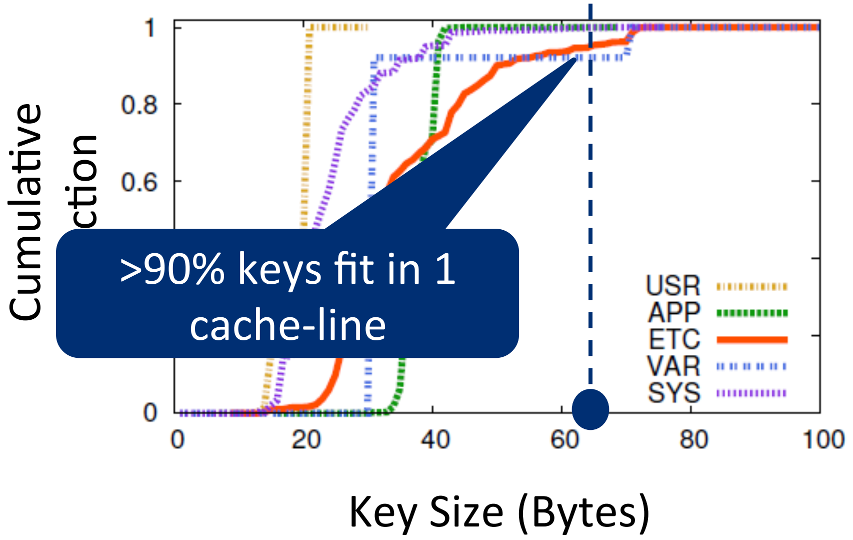
Workloads:
Key-values pairs are becoming small



90% keys/values fit in 1 cache-line

Write-heavy

Facebook Trace: Sigmetrics’12



>90% keys fit in 1 cache-line

Read-heavy

Implications:

No cache prefetching

Working set > cache size

Large number of key-value pairs / node

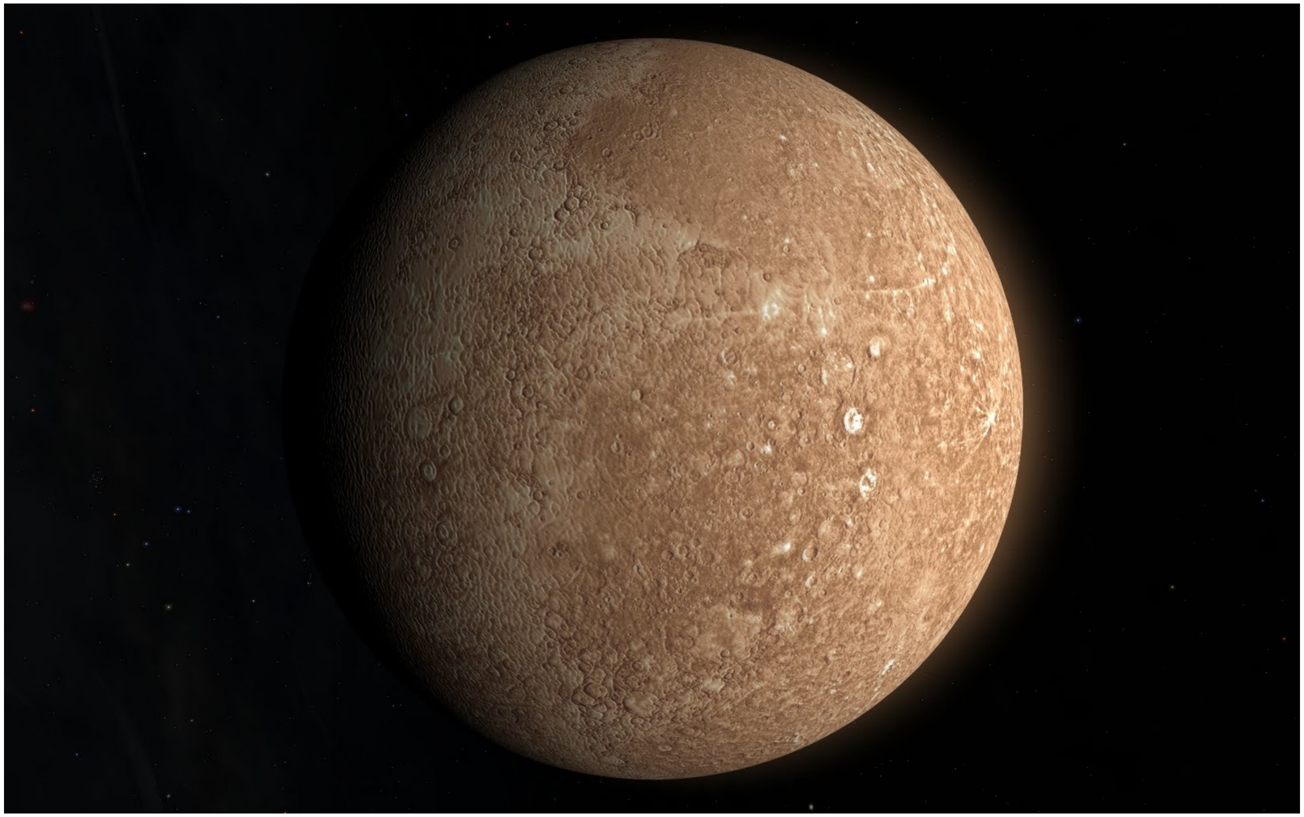
Metadata > cache size

Multi-threading → Synchronization overheads

Performance Comparison

	~Zero Synchronization Overhead ?	**Minimum DRAM Accesses ?**	**Workload Independent ?**
Memcached	**No**	Yes	Yes
MassTree (Eurosys'12)	Yes	**No**	Yes
MemC3 (NSDI'13)	Yes	Yes	**No**
Mercury	**Yes**	**Yes**	**Yes**

Mercury



Closest planet to sun !

Mercury:
Low latency-High Throughput Key-Value Store

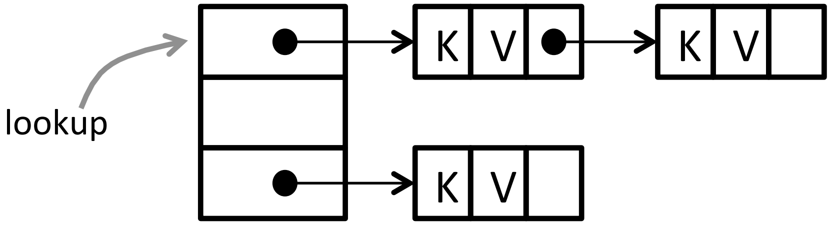
- Improves Memory system for a single server
- Operations: GET/SET/DELETE

- Workload independent
- Scalability
  - Store millions of key-value pairs
  - Support multiple threads and multiple writers
- Minimum DRAM accesses
  - 1 access: Hash-table
  - 1 access: Key-value pair

Design Choices

Lookup: Hash-table or Tree?

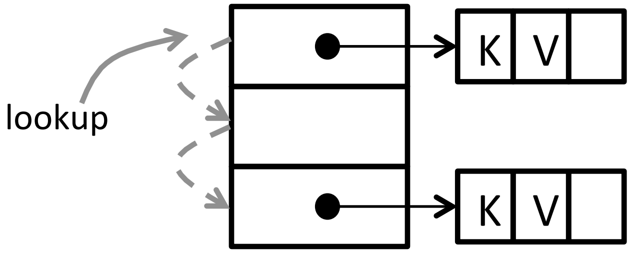
Chain Hashing



Advantage:
Lock granularity: Single bucket

Challenge:
Keep chain length small

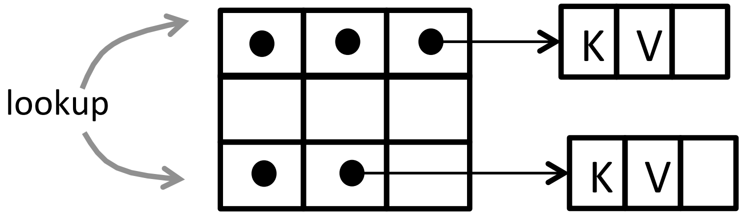
Linear Probing



Advantage:
Cache-friendly

Challenge:
Locking ?

Cuckoo Hashing



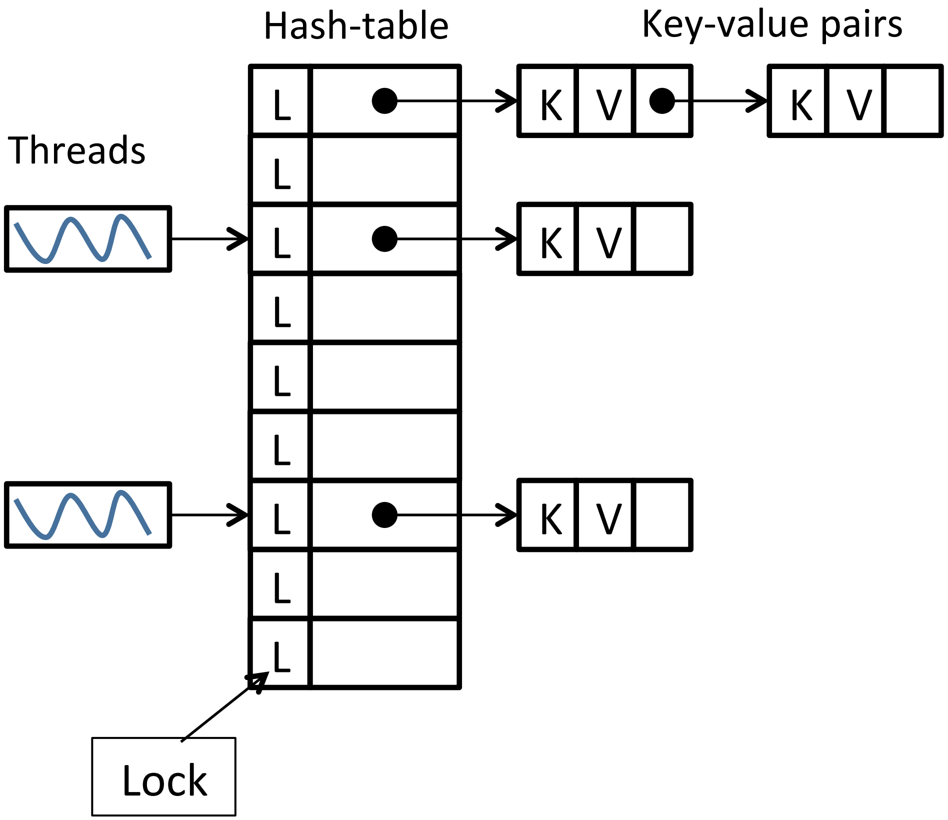
Advantage:
Higher hash-table occupancy

Challenge:
Locking? Expansion?

Mercury uses chain hashing
Dynamically expands hash-table size to keep chain lengths small

Mercury: Core data-structure

Chained Hash-table + each bucket protected with a single lock



Operations:

1) Acquire lock
2) Read/update hash-table entry

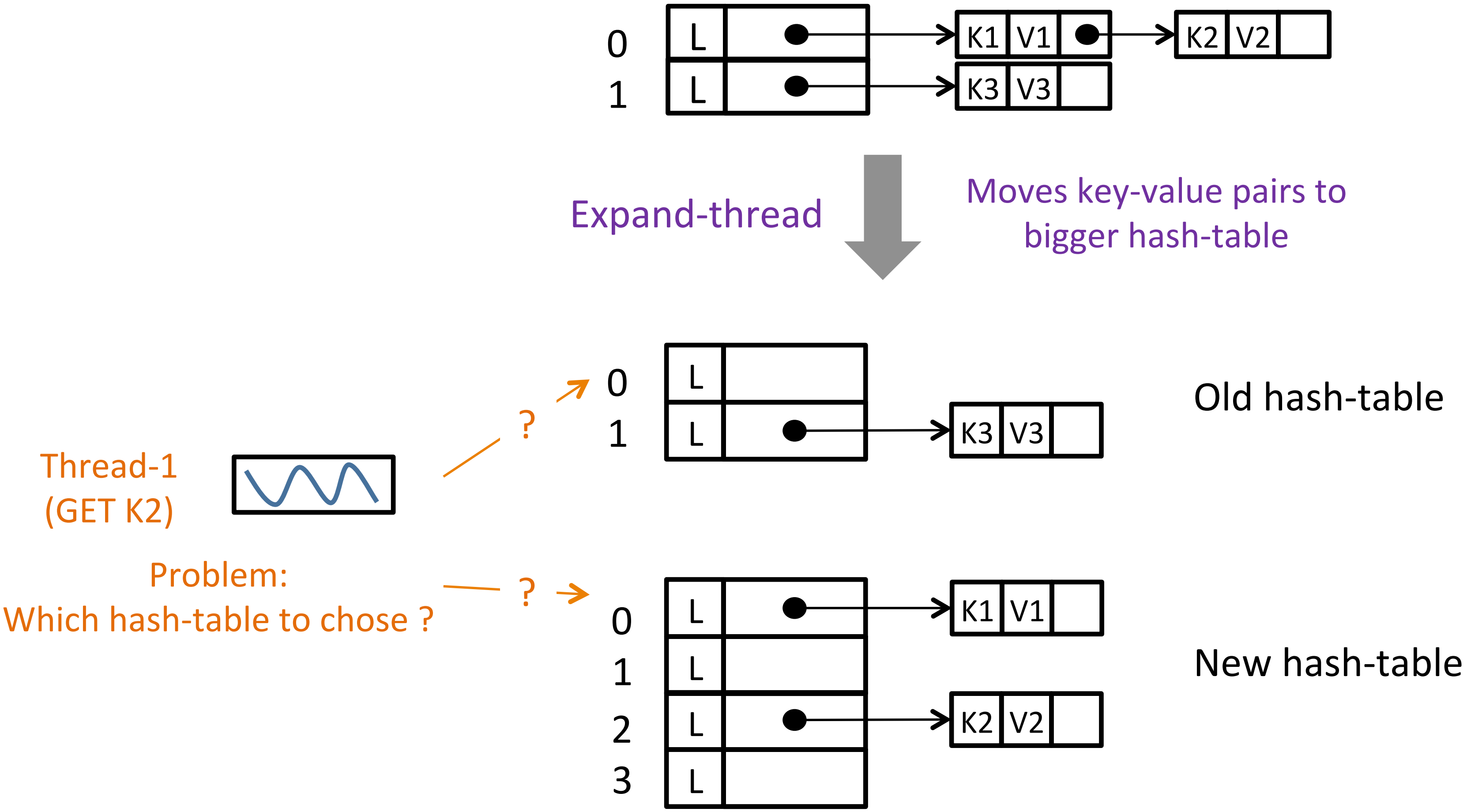
1 DRAM access due to 64-byte cache line

Contention probability@
(1 million locks, 12 thread)
= 0.07 x 10^{-3}

All locks implemented using TAS
(not pthread)

How to expand hash-table?
How to lookup key-value pairs during expansion?

Design: Hash-table Expansion



Expand-thread

Moves key-value pairs to bigger hash-table

Old hash-table

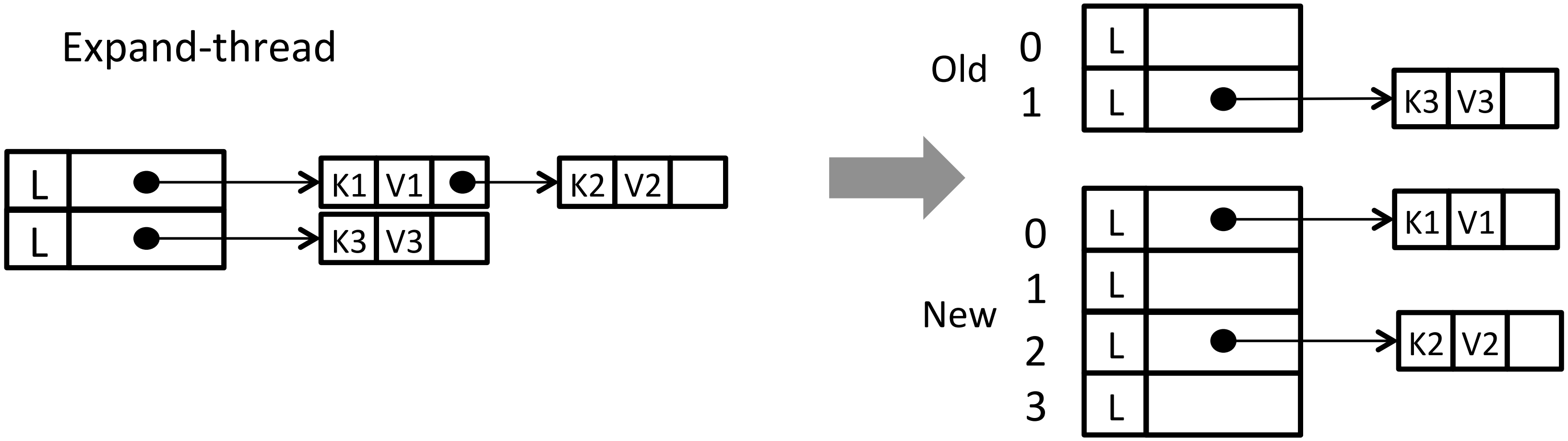
New hash-table

Thread-1 (GET K2)

Problem:
Which hash-table to chose ?

How to achieve lookup during hash-table expansion ?

Design: Lookup During Expansion



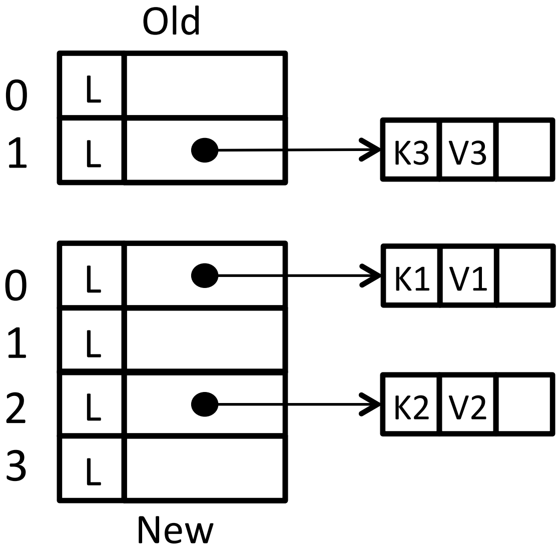
isExpanding = True | currentBucket = 1

Thread-1
(GET K2)

1) h = hash (K2) % 2 = 0

2) (h < currentBucket) = True

3) h = hash (K2) % 4 = 2
Look in **new hash-table**



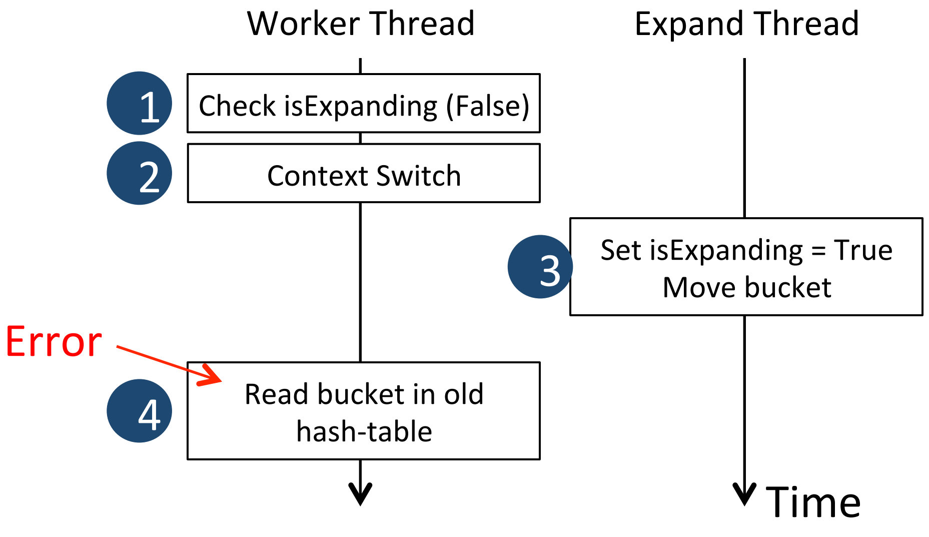
Thread-2
(GET K3)

1) h = hash (K3) % 2 = 1

2) (h < currentBucket) = False

3) h = 1
Look in **old hash-table**

Expanding Hash-table: Race Condition



Worker Thread

Expand Thread

1 Check isExpanding (False)

2 Context Switch

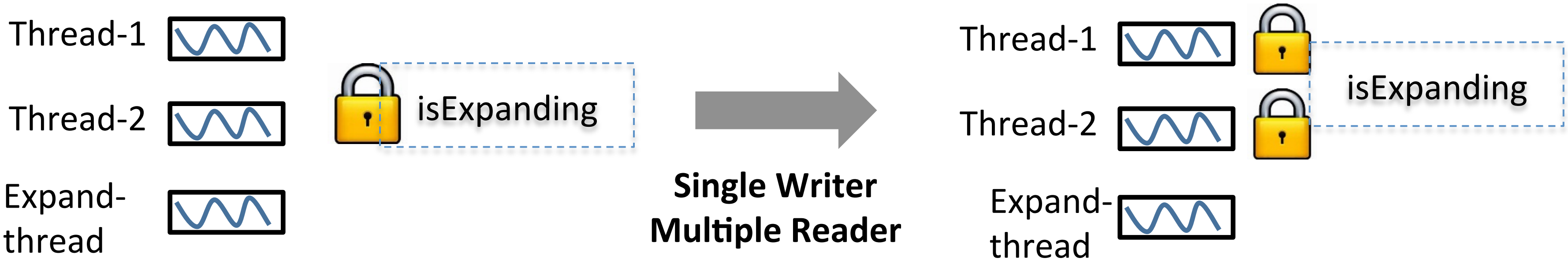
3 Set isExpanding = True
Move bucket

Error

4 Read bucket in old hash-table

Time

isExpanding bit needs synchronization



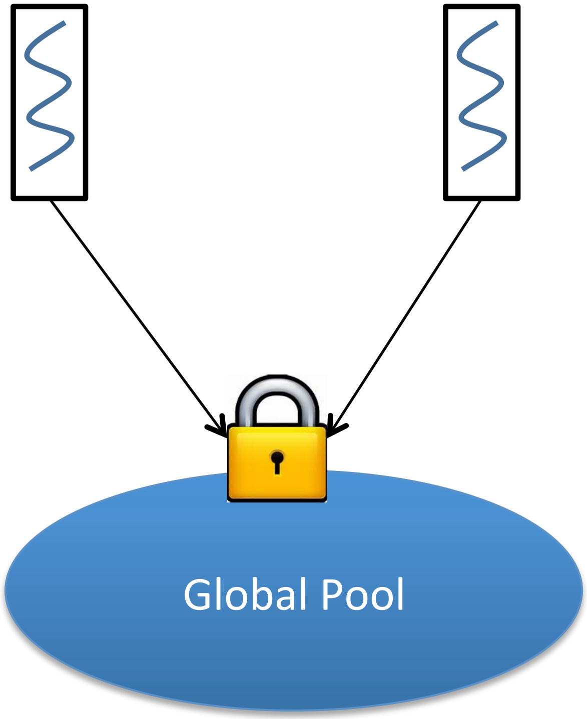
Single Writer Multiple Reader

Contention: <1 usec

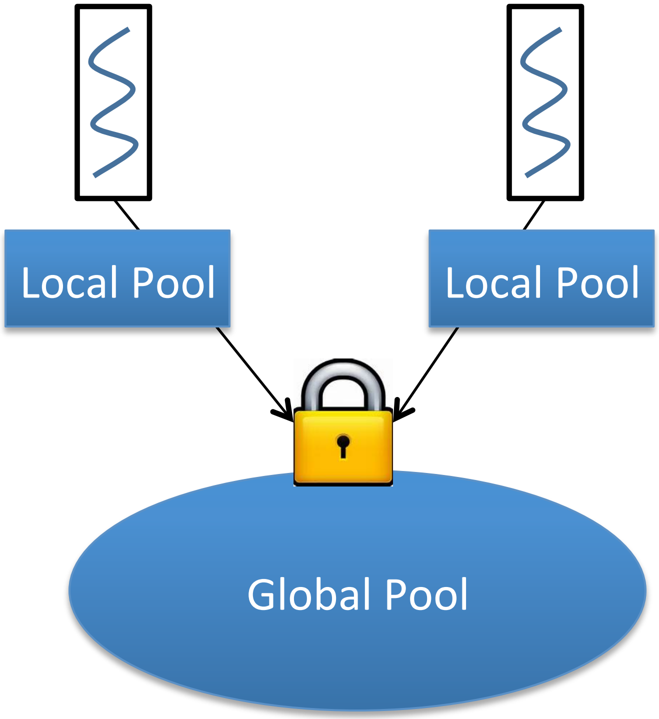
No cache invalidations between threads

hash-table size Independent

Design: Memory Management

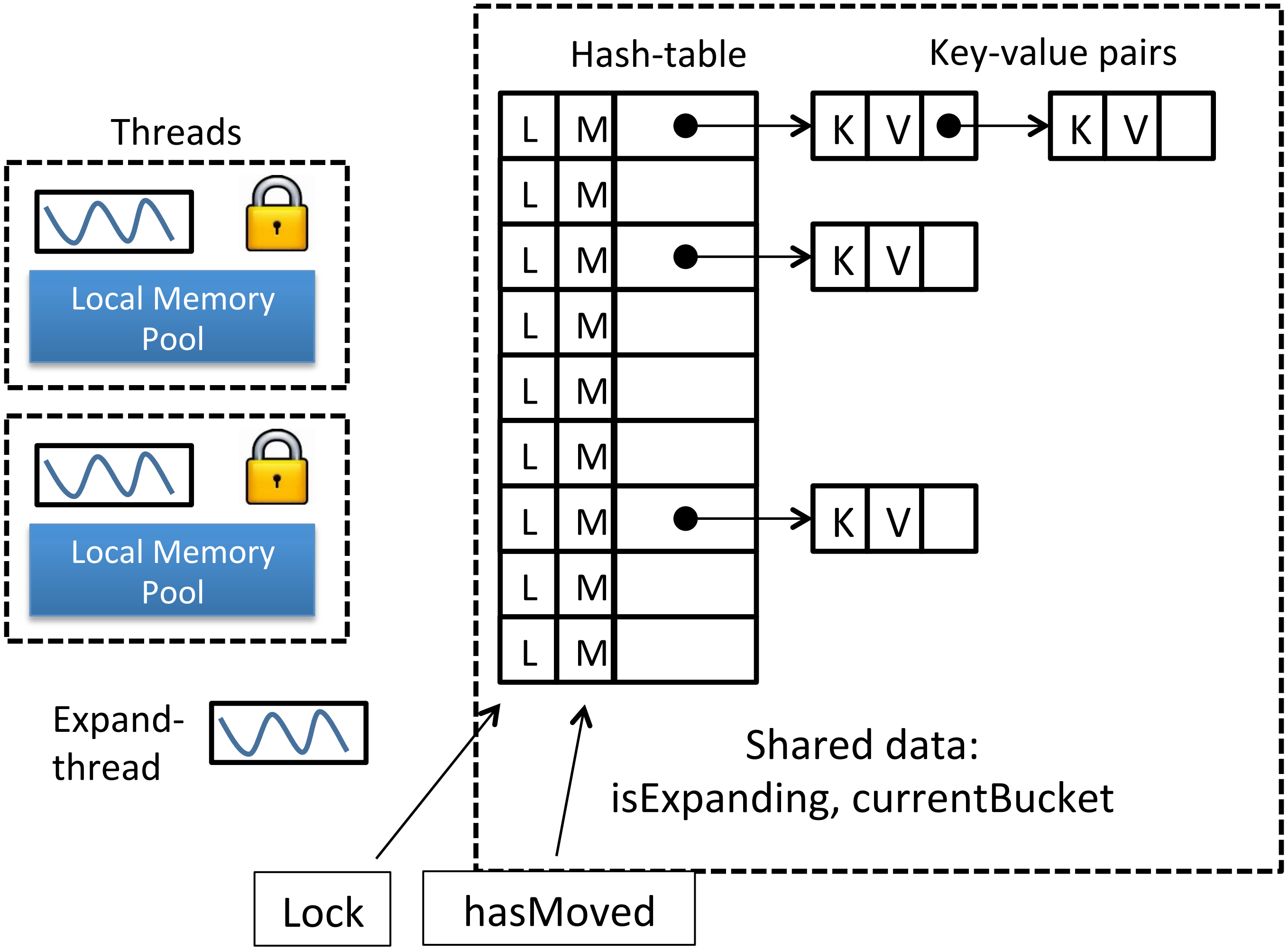


Lock Contention



Lock Contention Reduced

Mercury Design Summary



Threads

Local Memory Pool

Local Memory Pool

Expand-thread

Hash-table

Key-value pairs

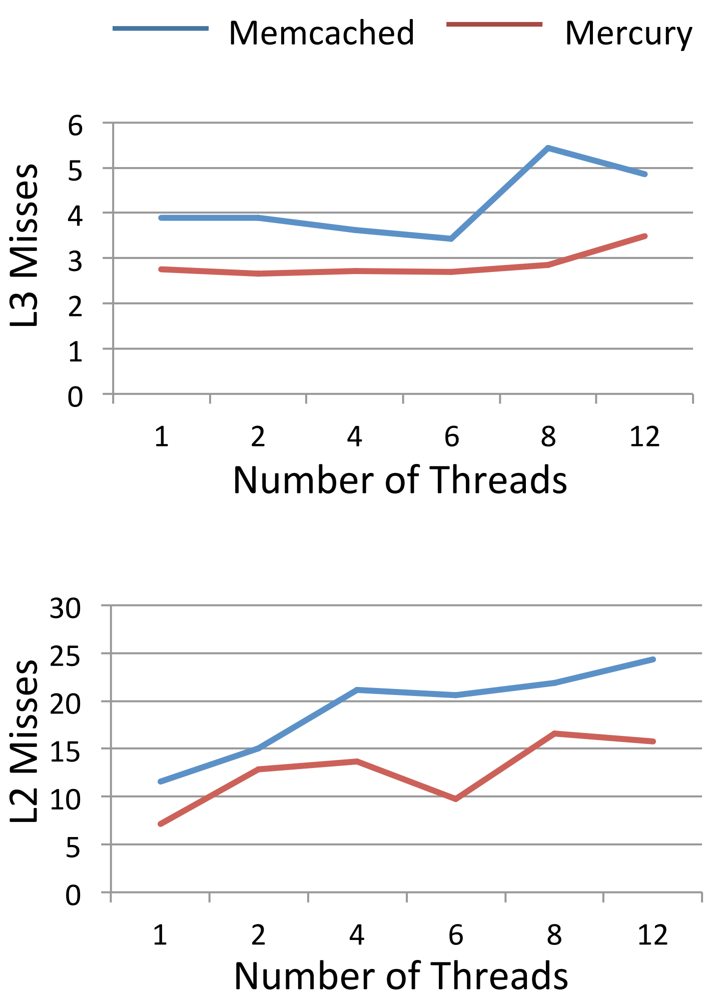
Shared data: isExpanding, currentBucket

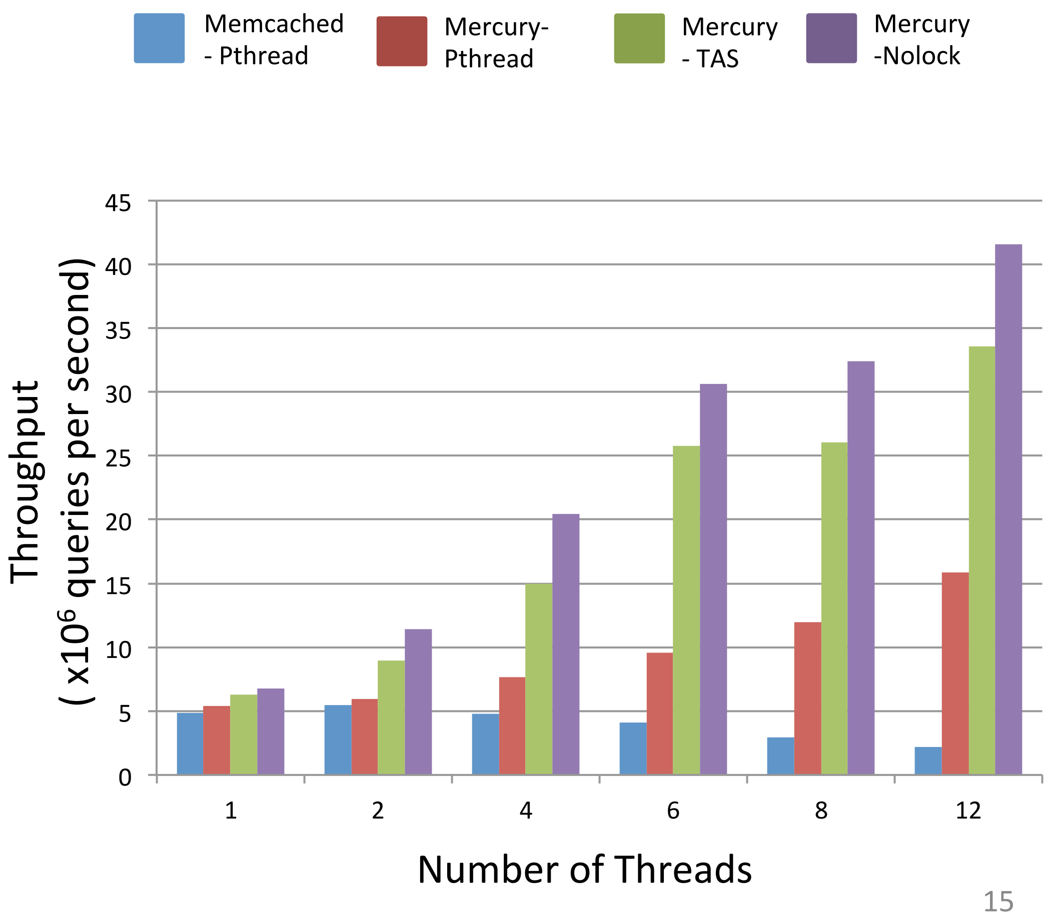
Lock

hasMoved

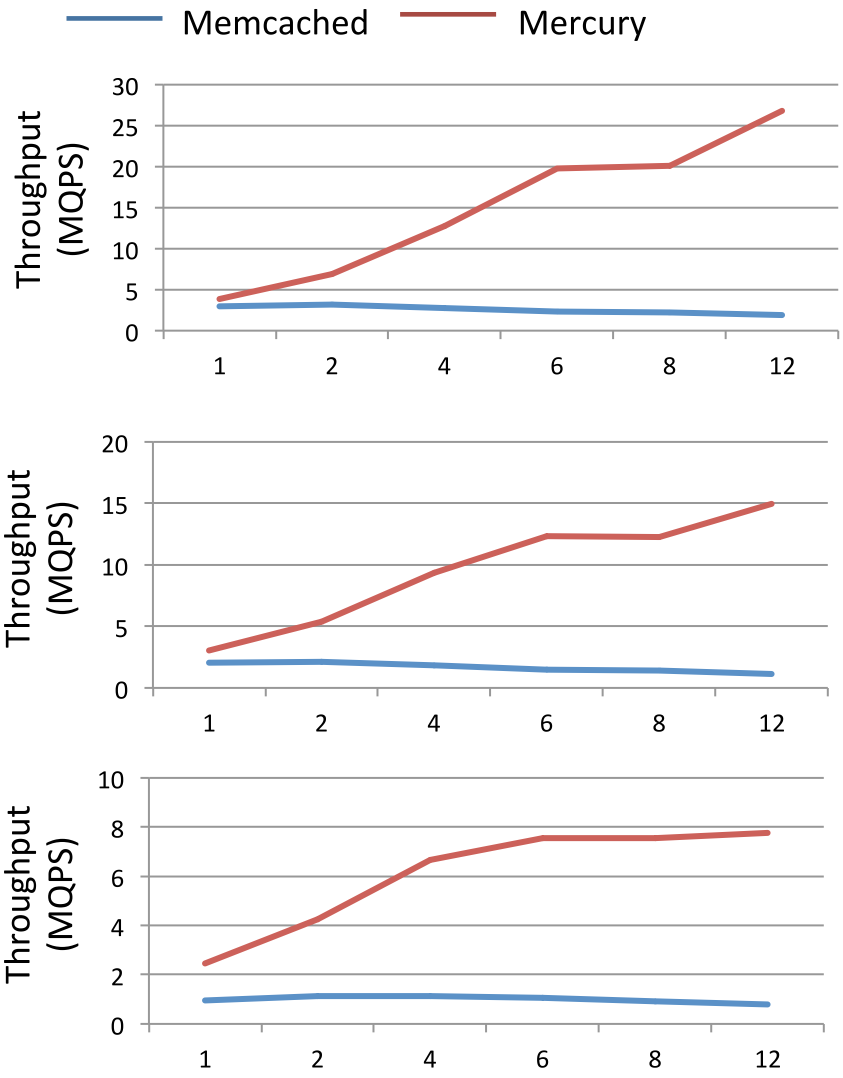
Evaluation: Cache Misses

Setup:
Key/Value size: 8-bytes each
Dataset: 32 million key-value pairs
#Queries: 100 million





Evaluation: YCSB



100% GET

75% GET

50% GET

YCSB Setup:

Key size: 24-bytes
Value size: 10-byte

Dataset: 24 million key-values
#Queries: 100 million

Conclusion

- Mercury
  - Memory efficient key-value store
  - Improves Memcached throughput by 14x
  - Workload Independent
  - Scales for number of key-value pairs and threads

- Design Choices
  - Chain-hashing (with expansion): 2 DRAM Access latency
  - Fine-grained locking: ~Zero contention

Thank You!

Special thanks to Pin Zhou, Ronen Kat, Sivan Toledo, Mary Baker and remaining organization committee !