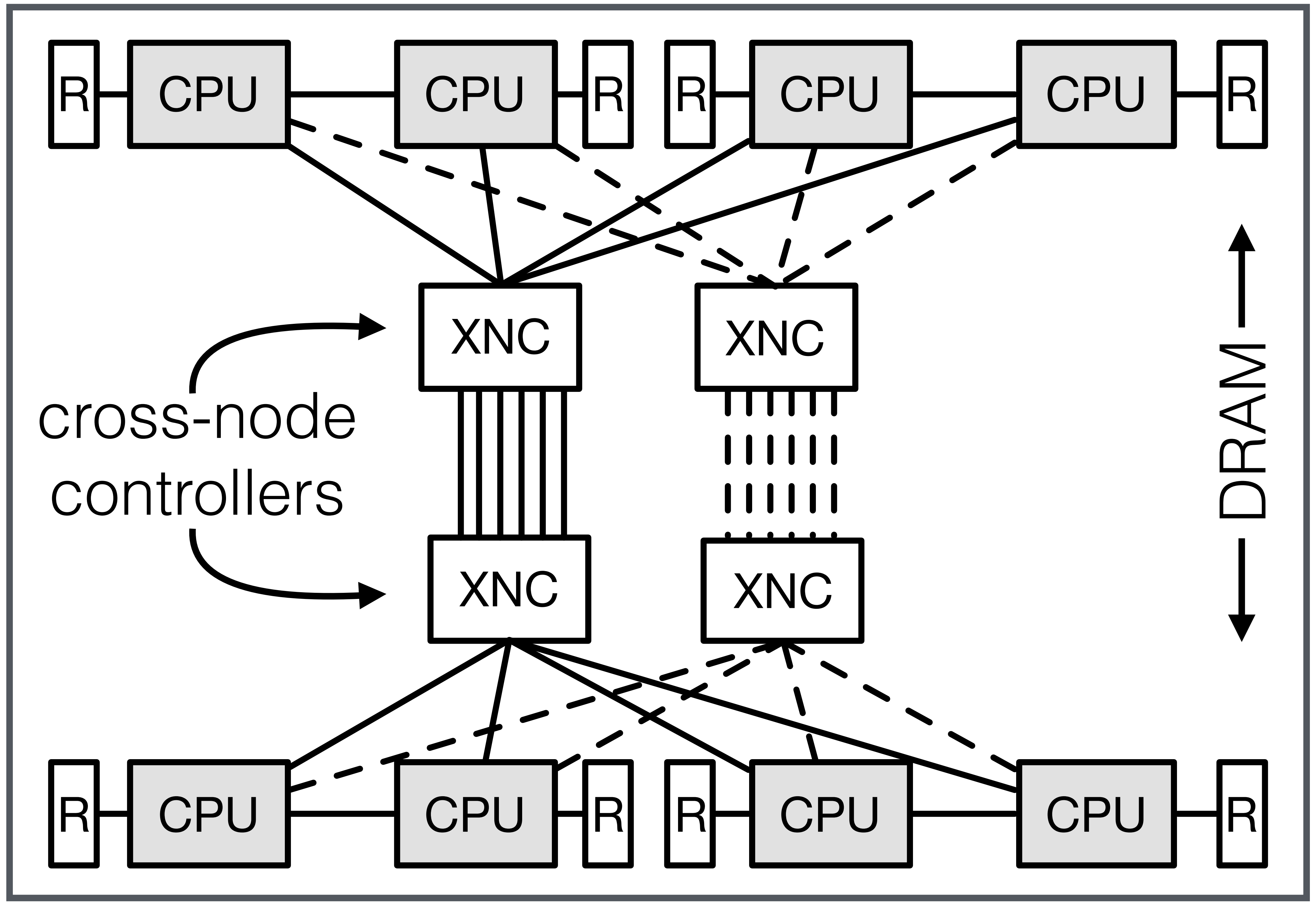
Concurrent Log-Structured Memory for Many-Core Key-Value Stores

Alex Merritt§ Ada Gavrilovska Yuan Chen† Dejan Milojicic
Georgia Institute of Technology Hewlett Packard Labs
mail@alexmerritt.de ada@cc.gatech.edu yuan.chen@jd.com dejan.milojicic@hpe.com

§Now at Intel. Work done as part of PhD at GT. †Now at JD Silicon Valley R&D center.

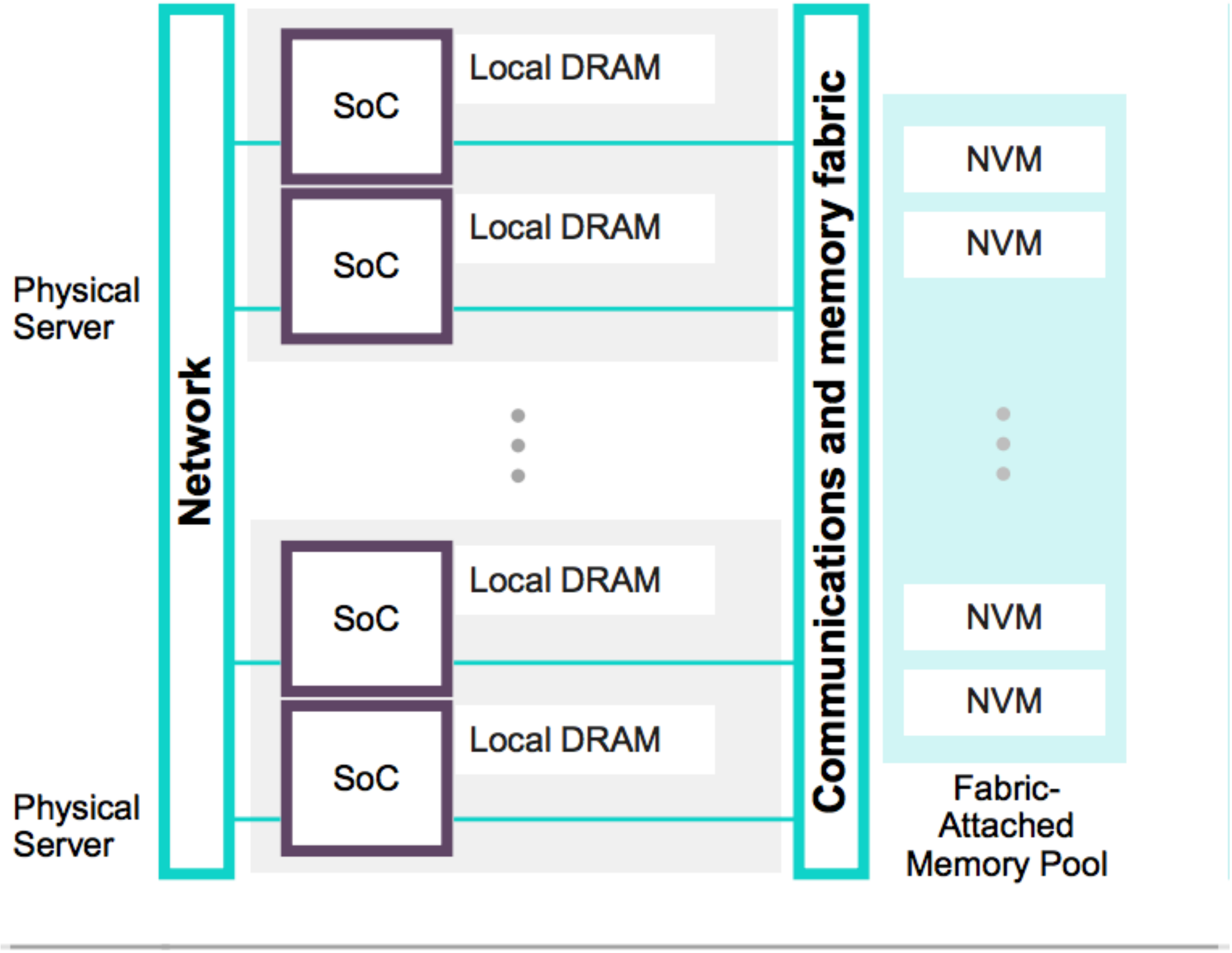
VLDB 2018
Rio de Janeiro, Brazil

Large Data Need Large Machines



Single-machine multi-socket ccNUMA

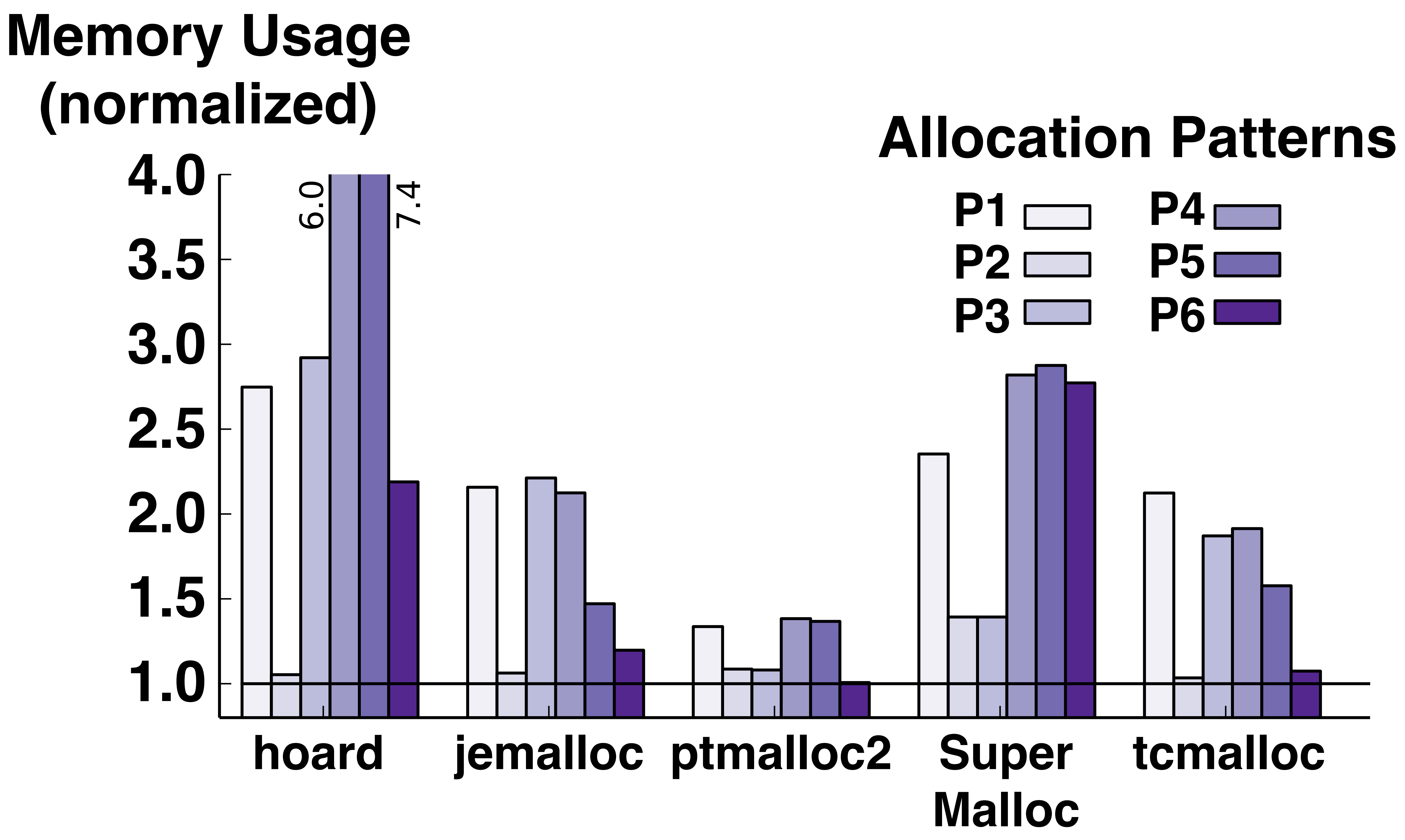
HPE "PREMA" architecture. 2010.



Multi-server non-coherent NUMA

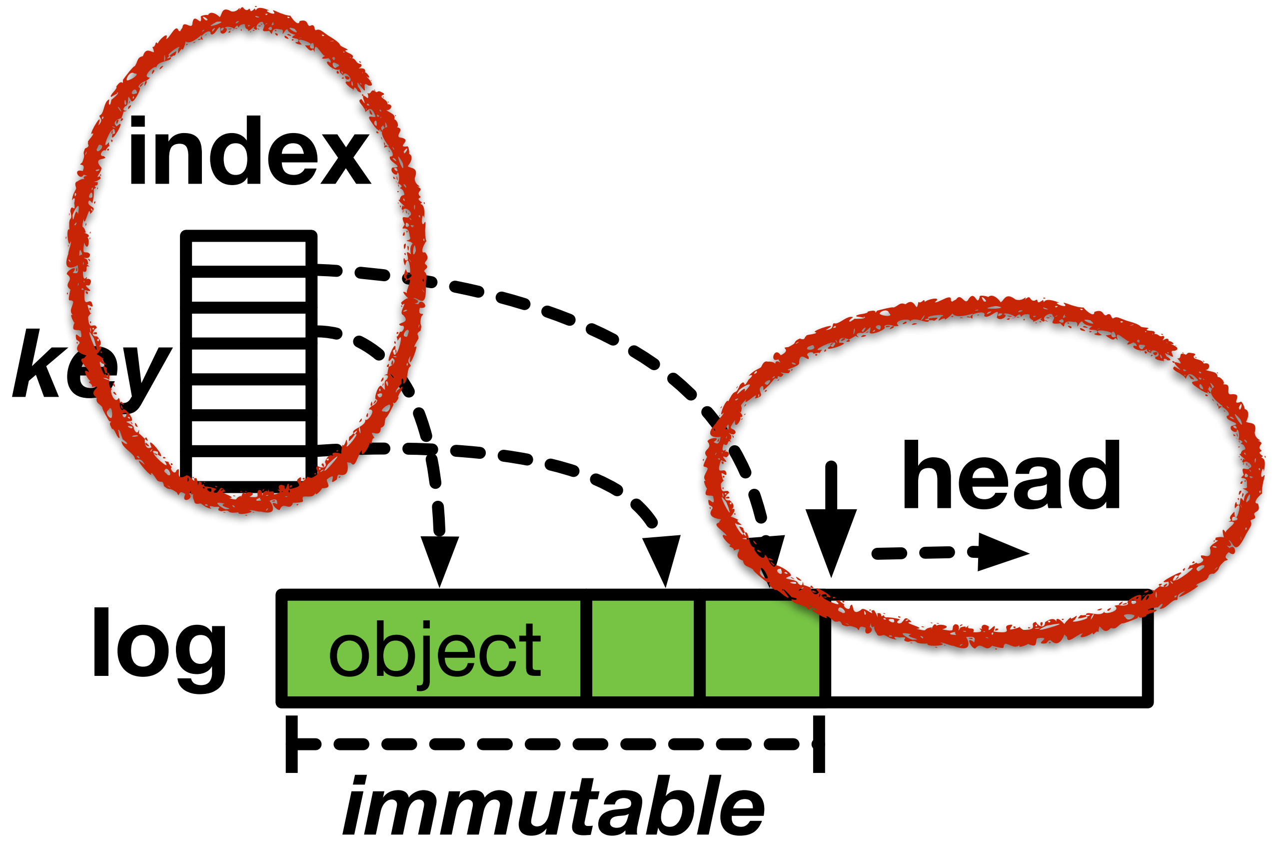
"Shared something." Kim Keeton. Keynote, FAST2017

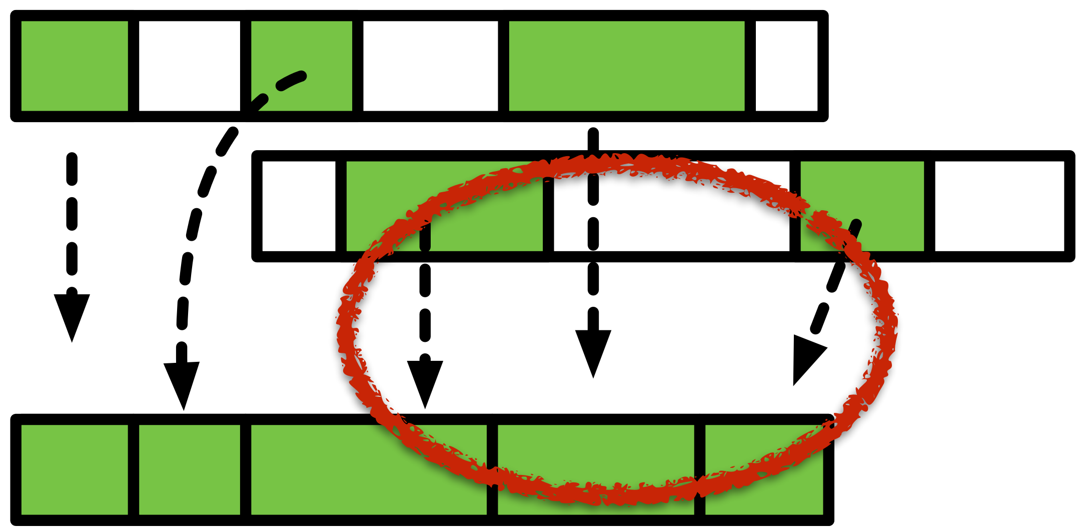
Managing Large Main Memory



Label	Pattern	Label	Pattern
P1	60 ➟ 70 B	P4	1 ➟ 10 KiB
P2	1000 ➟ 1024 B	P5	10 ➟ 100 KiB
P3	1000 ➟ 1030 B	P6	500 ➟ 600 KiB

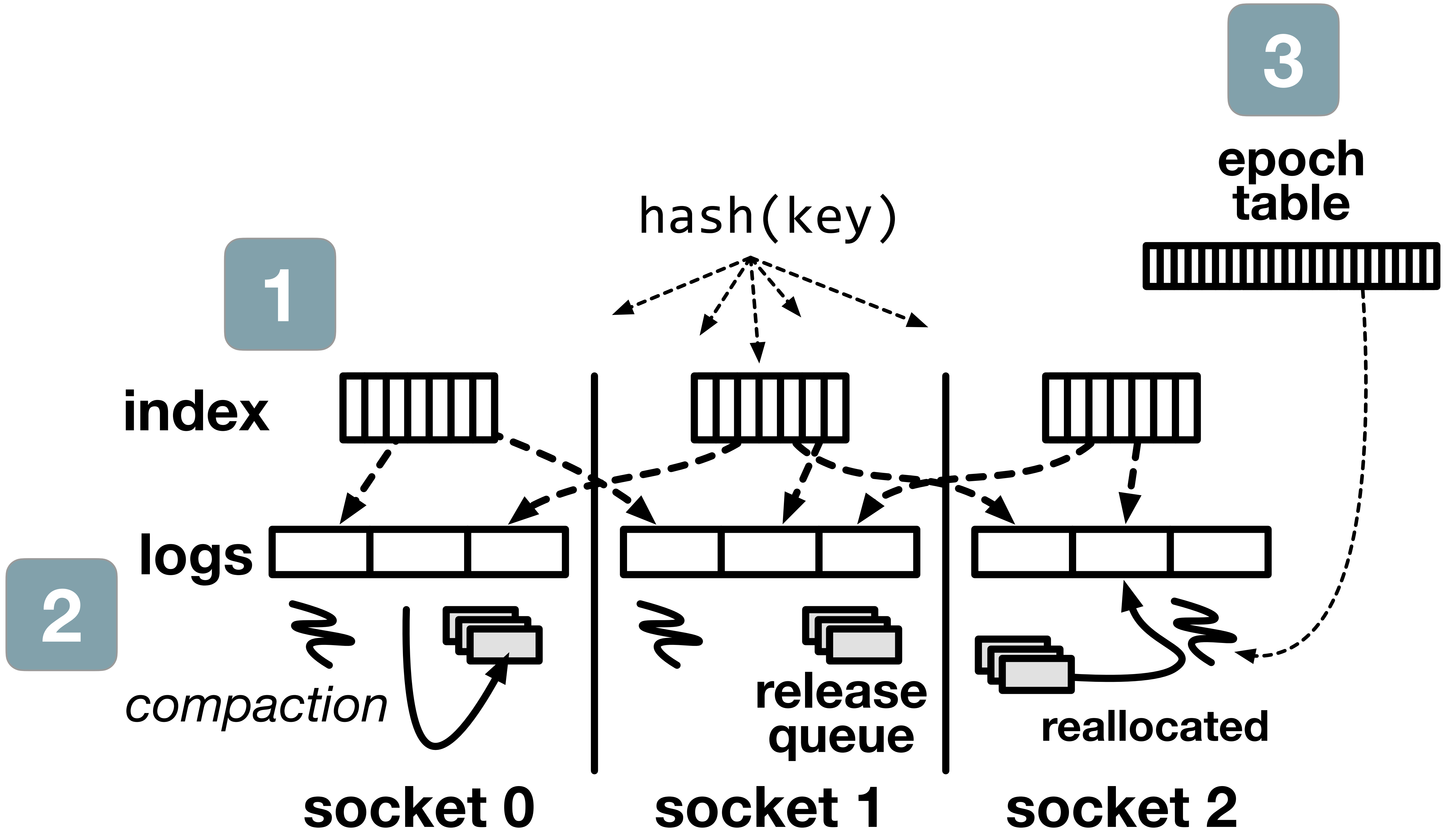
Allocation Patterns





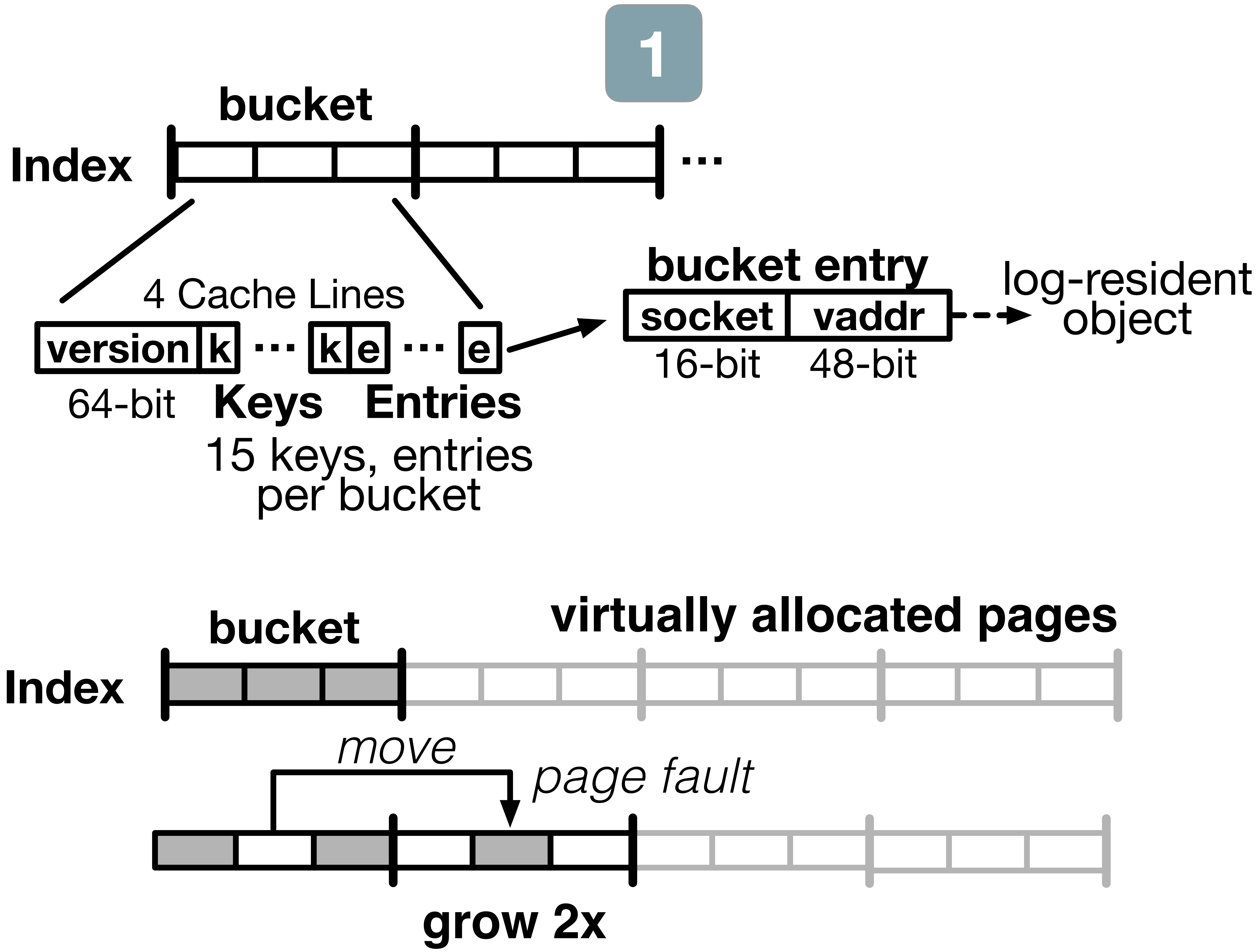
- Unstructured data stores
  - Heaps, buddy list, slab cache
  - Evolving data: fragmentation
- Log allocation — scalability issues

Nibble: Concurrent Log-Based KVS



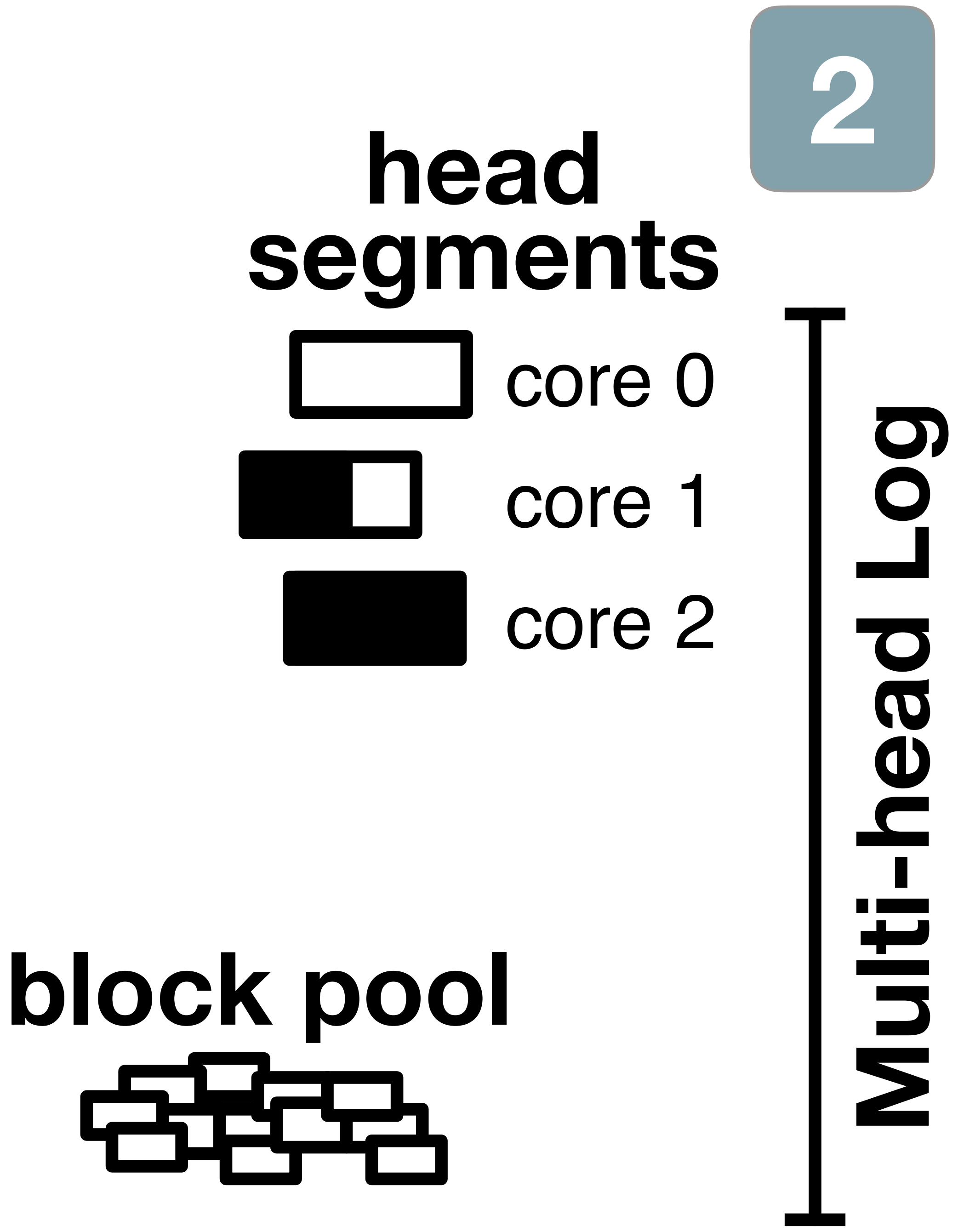
- In-process KVS, high concurrency
    - Concurrent index (partitioned)
    - Multi-head log (partitioned)
    - Consistency via hardware epoch
- Per-socket isolation
    - Index, compaction, allocation
- Written in ~4000 lines of Rust
    - https://www.rust-lang.org

Optimistic Concurrency Index



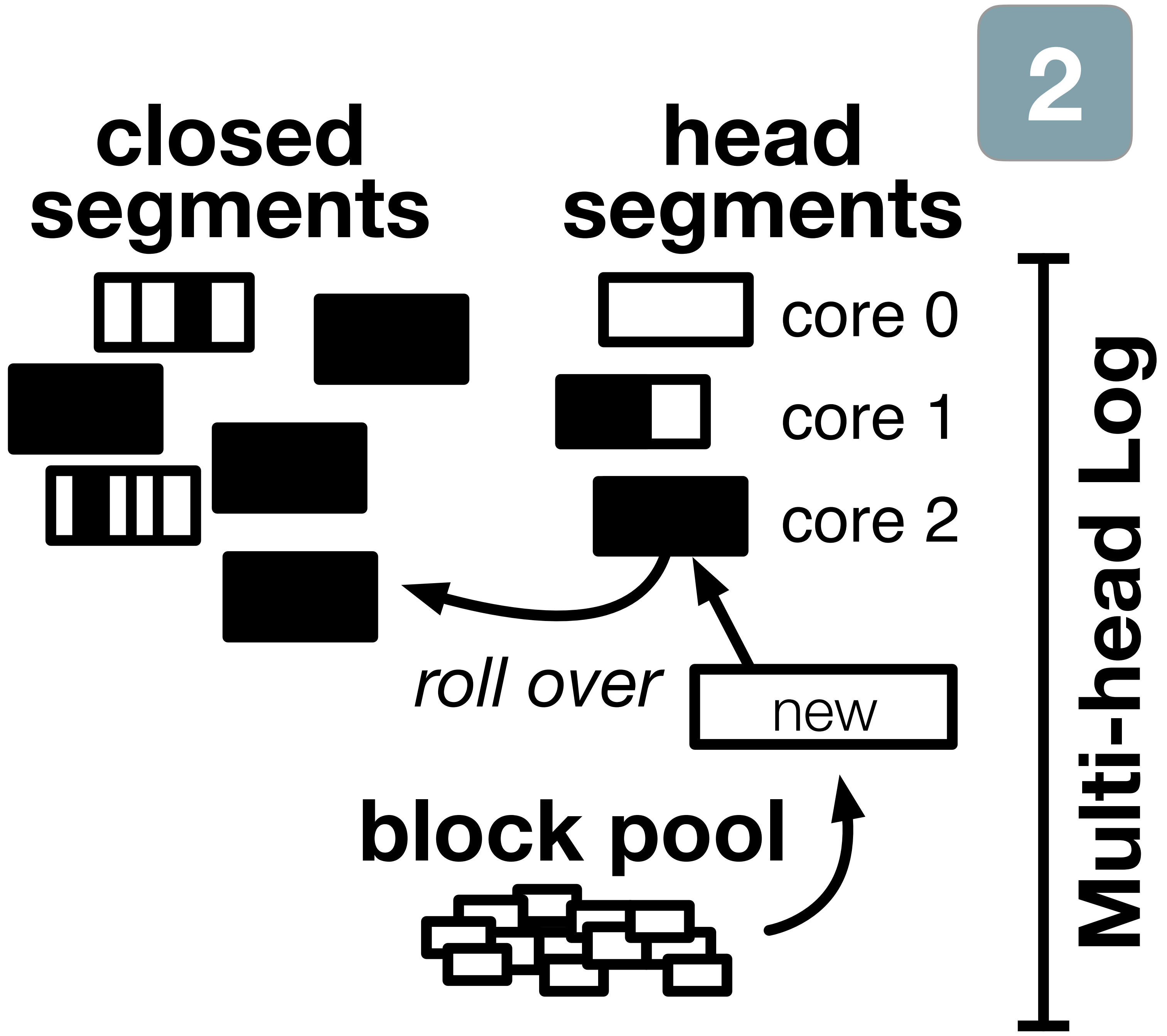
- Open addressing with linear probing
    - 8-byte keys, 8-byte values
- Buckets guarded by atomic version
    - Lookups record twice: before and after reading value. Retry if changed
    - Writer locks with `cmpxchg` to odd, and again to even value to release
- Over-allocate virtual memory
    - Grow by faulting in physical pages

Multi-head Logs for Parallel Writes



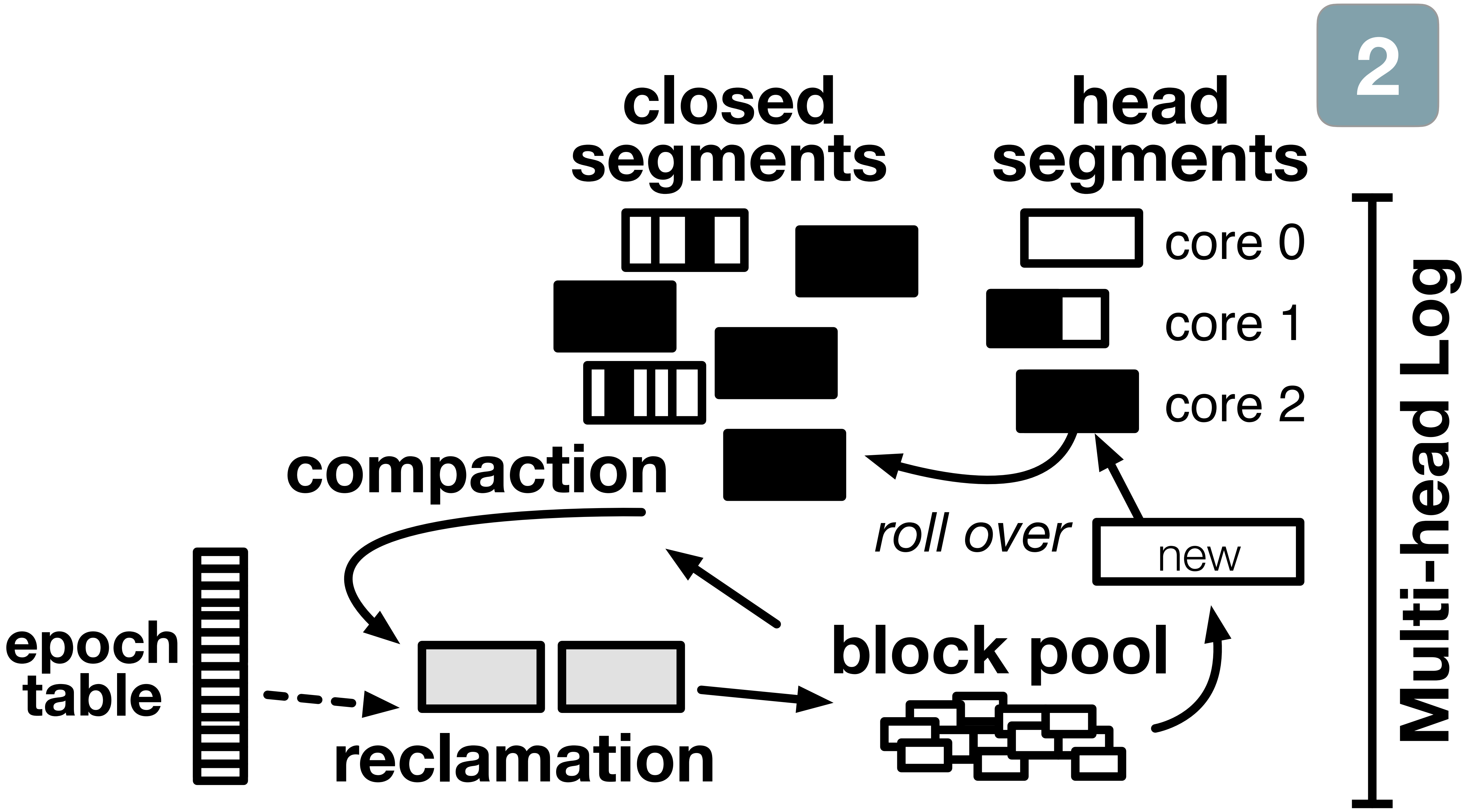
- Memory allocated as blocks
    - Segment = container of blocks
- Multiple heads: one per core
    - Thread append to core's head segment via (`rdtscp`)

Multi-head Logs for Parallel Writes



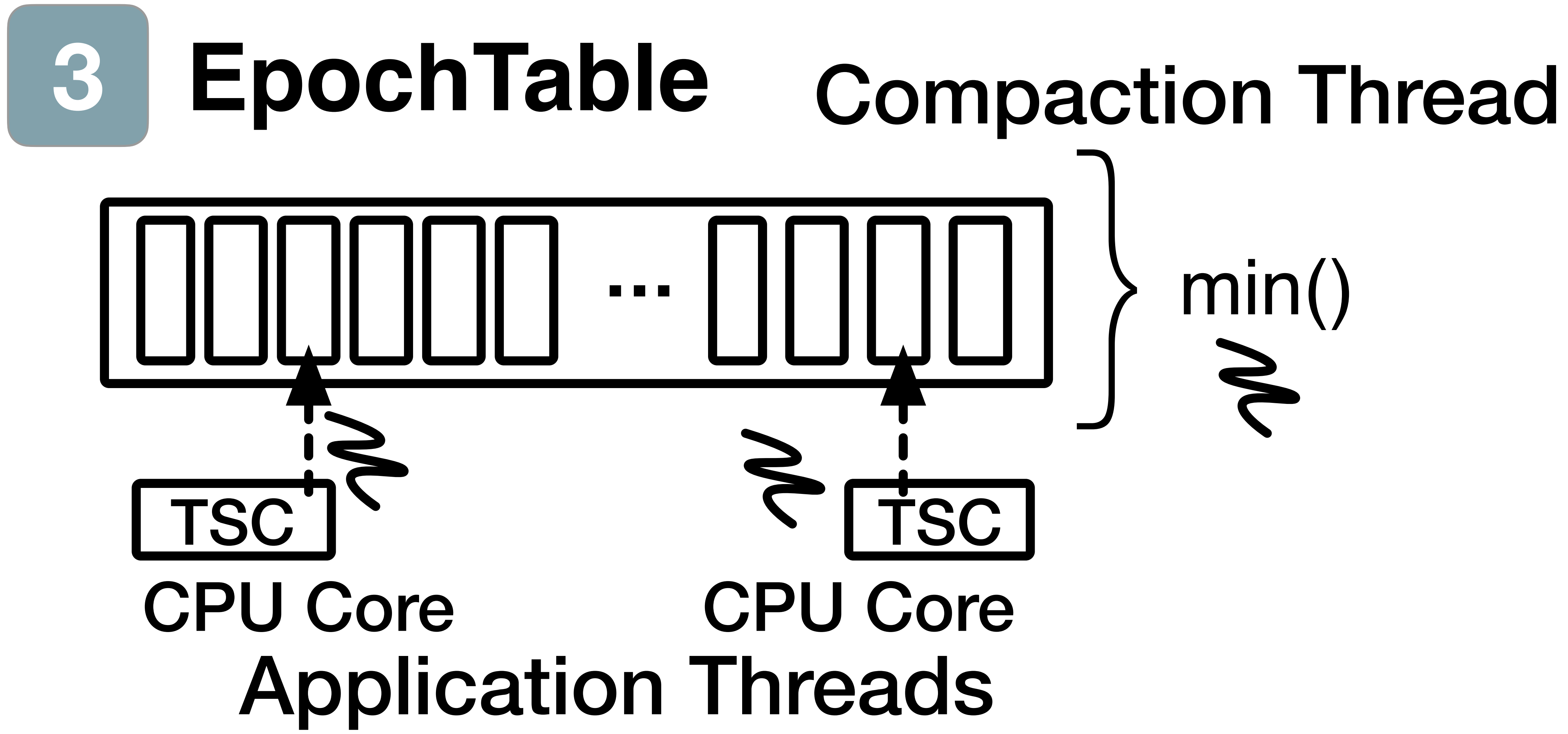
- Full head is closed and replaced with new segment

Multi-head Logs for Parallel Writes



- Compaction relocates objects in closed segments
- Each compaction thread works on dedicated segments
  - Parallel sorting and compaction
- Segment selection (compaction) based on cost-benefit metric

Concurrent Reference Tracking

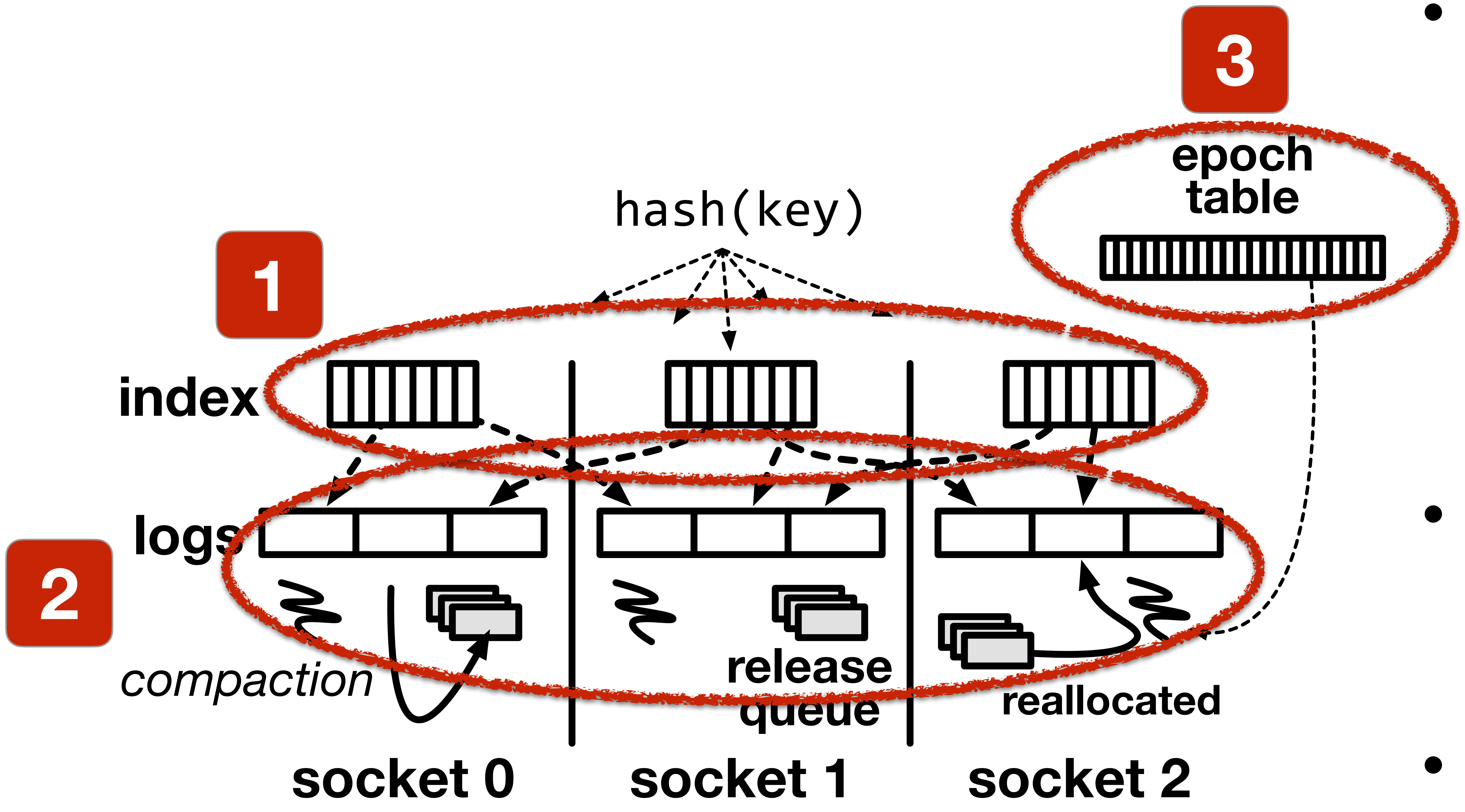


Entries are cacheline-sized

- Application thread assigned unique entry
- On `PUT`, `GET`, `DEL` thread records local core TSC: $\mathtt{e}_{\mathtt{op}}$
  - On return, thread records NIL (0)
  - No atomics, no locking
- Compacted Segments are stamped: $\mathtt{e}_{\mathtt{seg}}$
- Segments released when

  $\mathtt{e}_{\mathtt{seg}}$ `< min(EpochTable)`

Nibble: Concurrent Log-Based KVS



- In-process KVS, high concurrency
  - Concurrent index (partitioned)
  - Multi-head log (partitioned)
  - Consistency via hardware epoch
- Per-socket isolation
  - Index, compaction, allocation
- Written in ~4000 lines of Rust
  - https://www.rust-lang.org

Goals of Evaluation

- How far do modern systems scale?
  - Real measurements on enormous machine
  - Broader workloads: static and dynamic data
- Push concurrency of log-allocation:
  - How we compare — at low utilization, and at high?
  - Can we handle pure writes?
- Did our design decisions make sense?

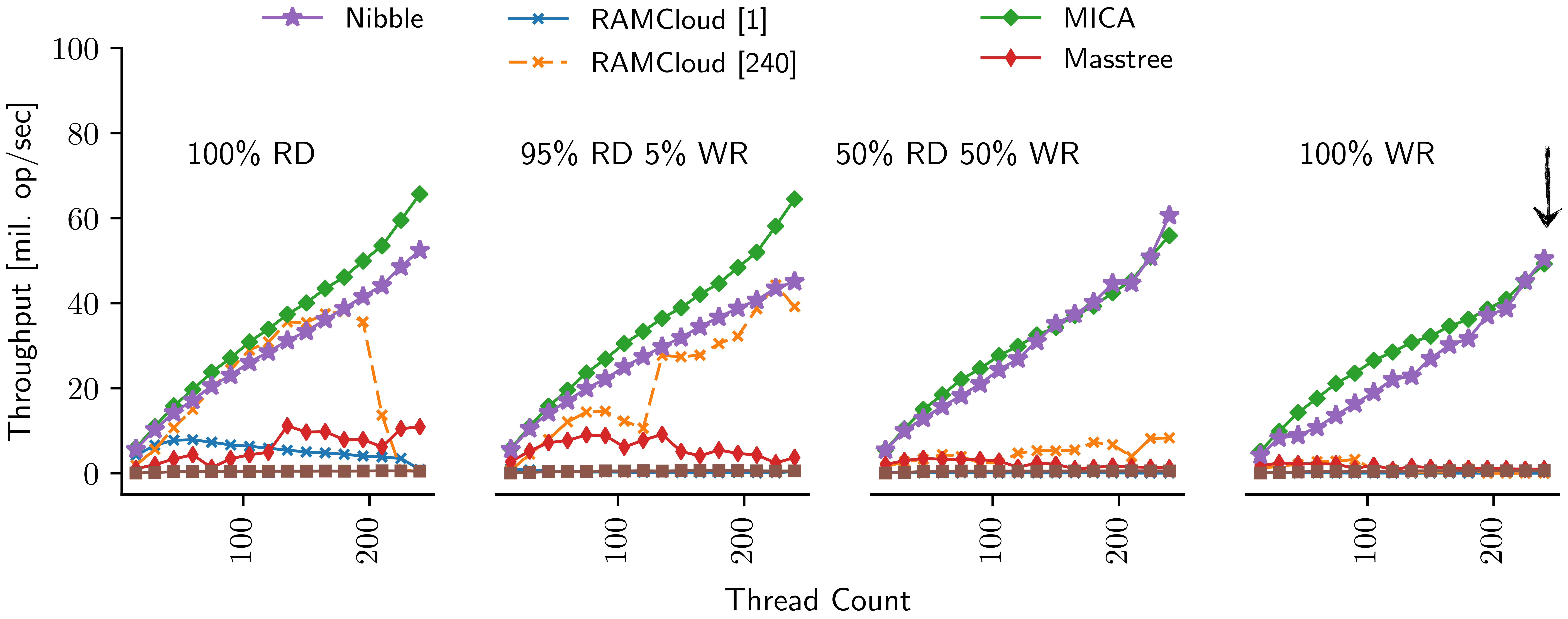
Evaluation Overview

HPE SuperDome X — 12 TiB DRAM, 16 Intel Xeon E7-2890 v2 (240 cores)

Workload	Type	Description
Fragmentation	Dynamic	Cycle allocate / release. Six patterns.
Postmark Trace	Dynamic	**500-4096 bytes, 20% ins/del**, 183m. ops., 5.7 GiB/thread
YCSB	Static	**1 TiB (1bn. 1 KiB objects)**, various patterns.

System	Description	Modifications for Evaluation
Redis	In-memory data structure store	No parallelism. Launch **64 instances**.
RAMCloud	In-memory log-structured KVS	Extract `ObjectManager`. **1 and 240 instances**.
Masstree	Concurrent OCC B^+-tree	Extracted tree for use.
MICA	Concurrent + partitioned KVS	EREW mode. SIPHash.
Nibble	Scalable log-based KVS	8 comp. threads. 64 KiB block 32 MiB segment

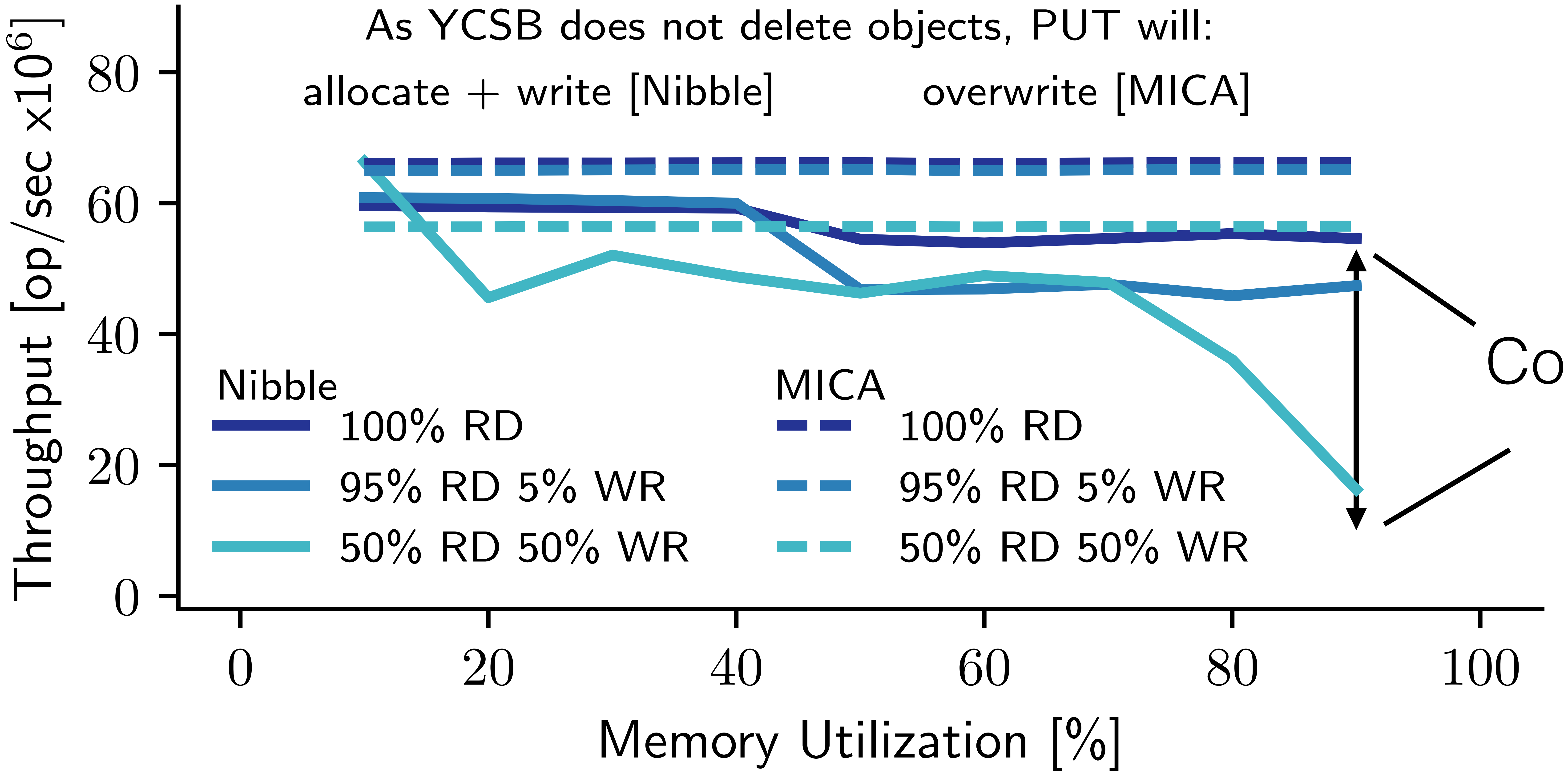
Static Data Sets



YCSB Uniform Distribution: 1bn. (2^{30}) 1KiB objects = 1 TiB, 12% capacity.

YCSB Zipfian discussed in paper.

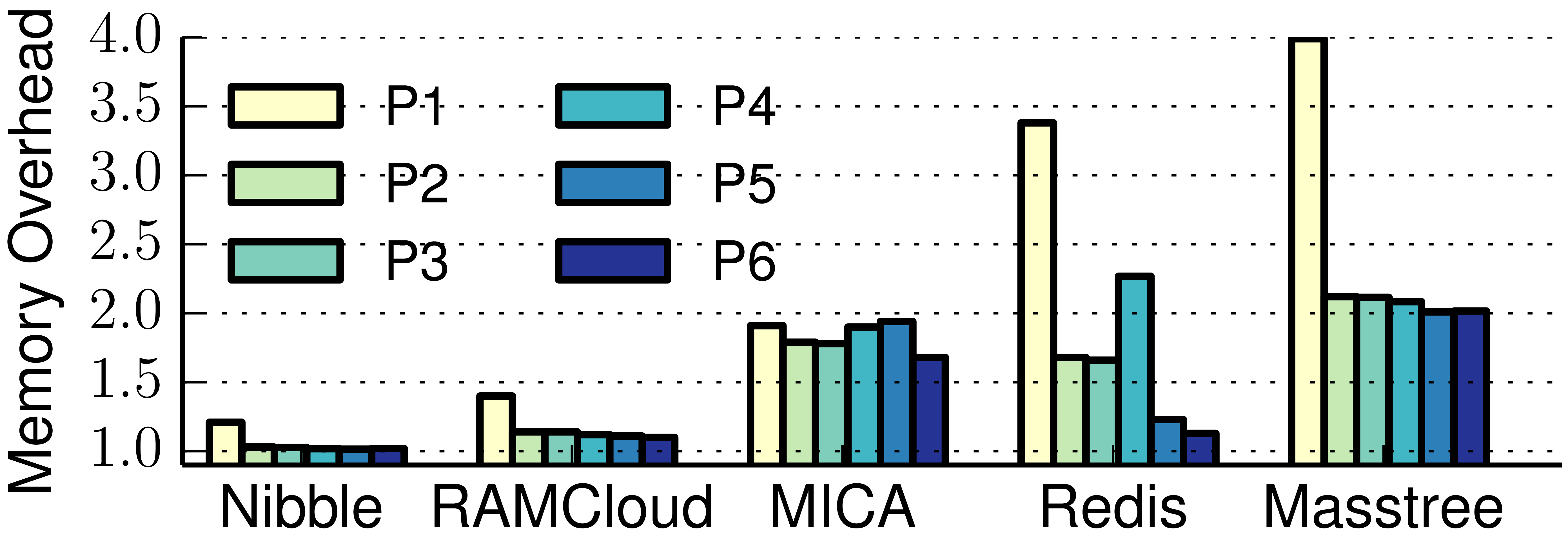
… and with less memory



Compaction costs.

YCSB Uniform Distribution @ 240 threads

Revisit Memory Fragmentation

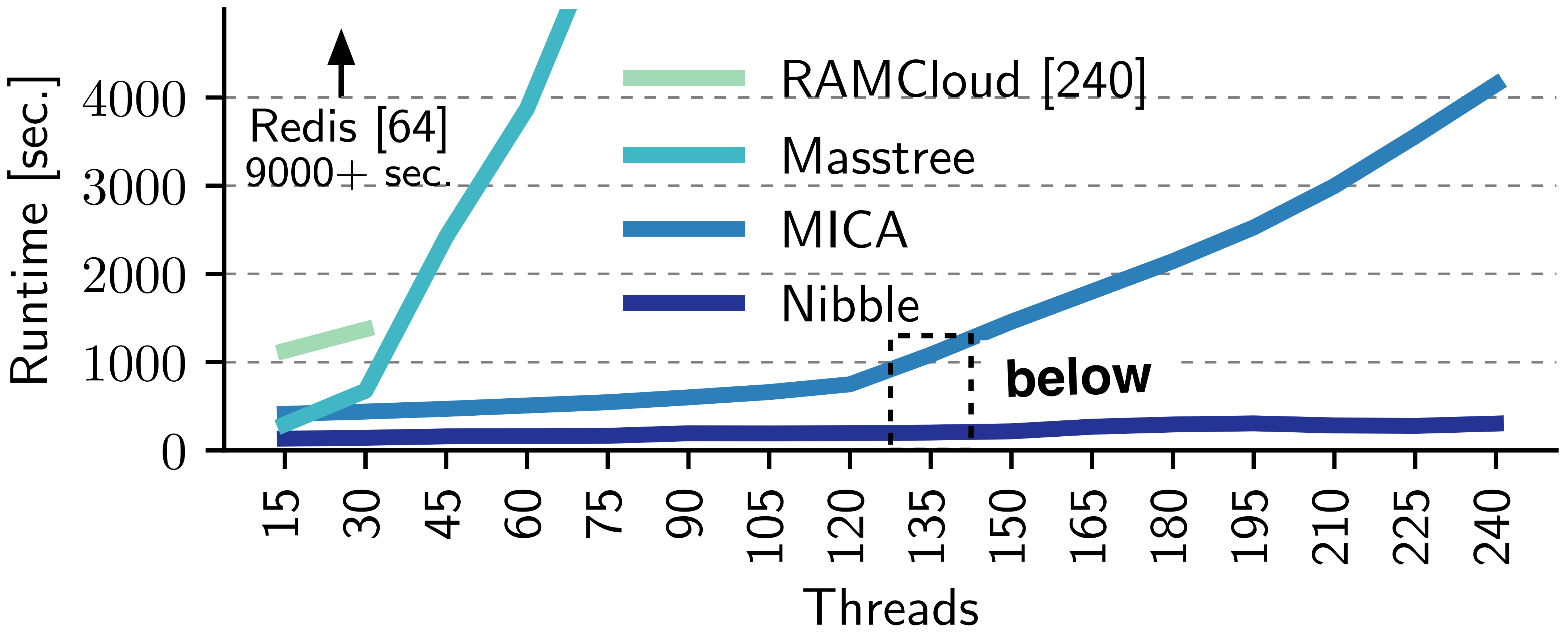


Label	Pattern	Label	Pattern
P1	60→70 b	P4	1→10 KiB
P2	1000→1024 b	P5	10→100 KiB
P3	1000→1030 b	P6	500→600 KiB

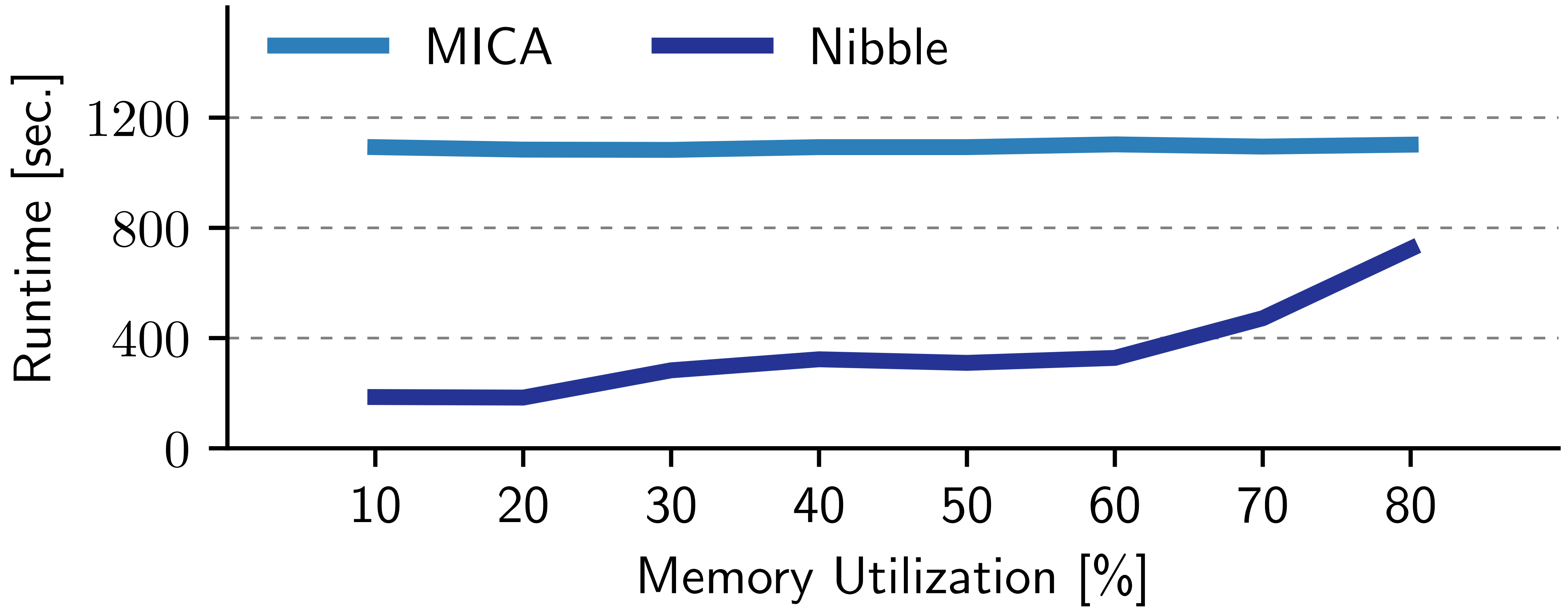
- Compaction resists fragmentation
- **Nibble ≤ 10%** additional memory
- MICA, Redis, Masstree ca. **1.5 - 2x**
- P1 is high as objects are small
    - Metadata per object is fixed

Dynamic Data Sets

Time to complete trace @17% utilization



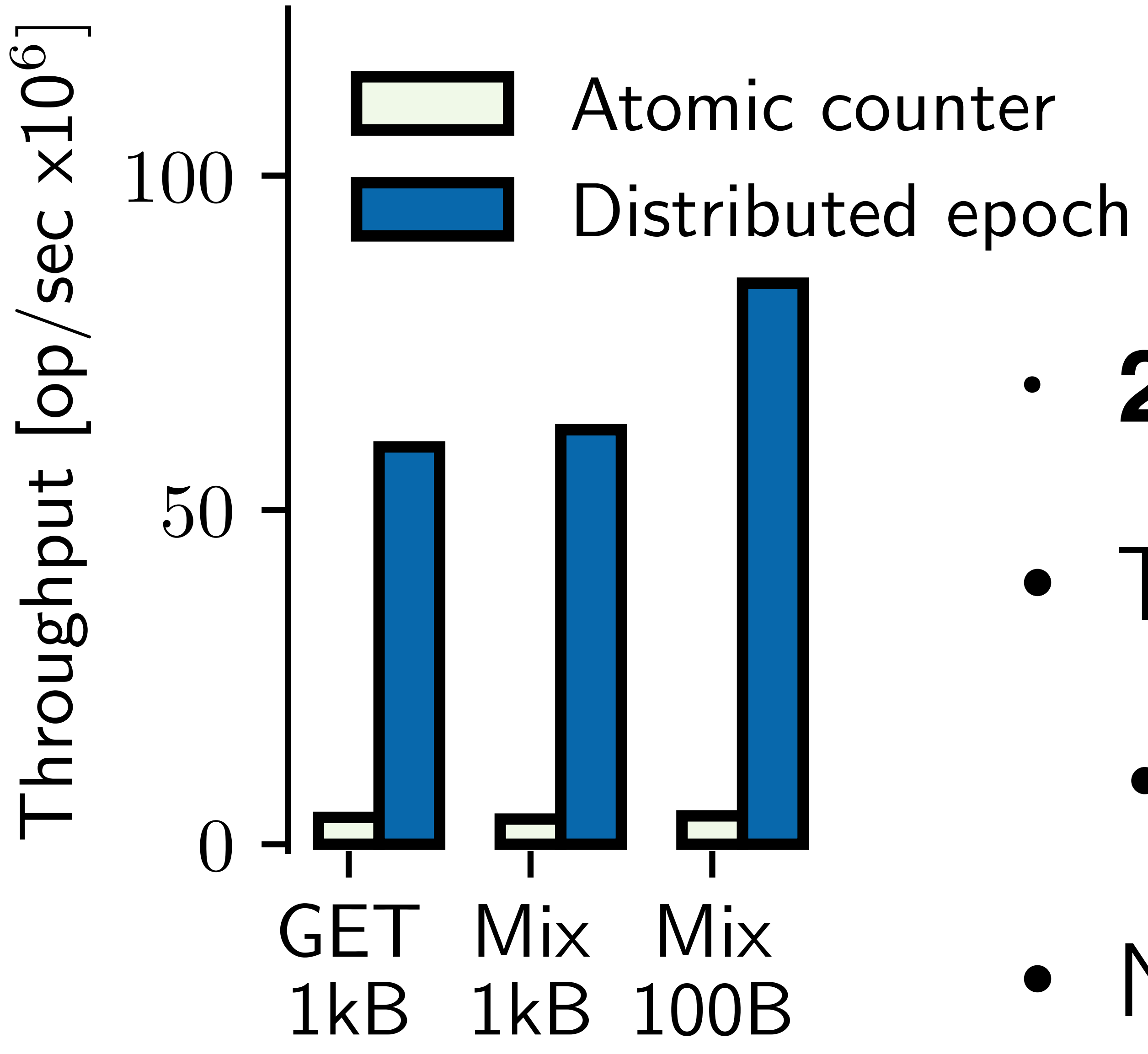
... and with less memory (135 threads)



- Captured trace from TableFS + **Postmark**
  - Object sizes **500 - 4096 bytes**
  - **10% insertion 10% deletion**
  - 18mil. objects, 5.7 GiB working set
- Each thread executes trace in isolation
- Measure total time to complete

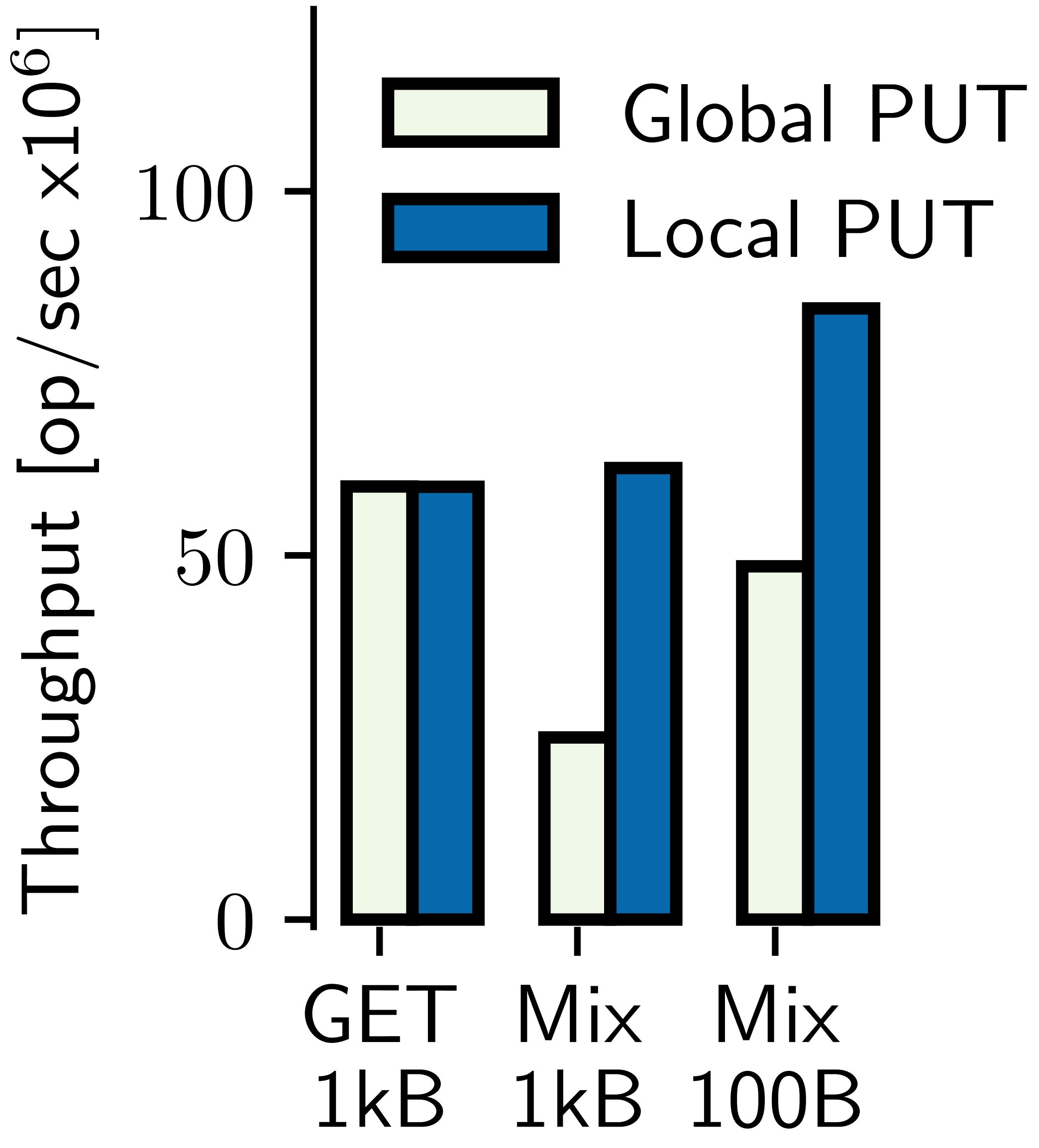
Micro-Evaluation

How much does TSC help?



- **20x greater throughput**
- TSC auto-increments
  - 30ns to read
- No synchronization

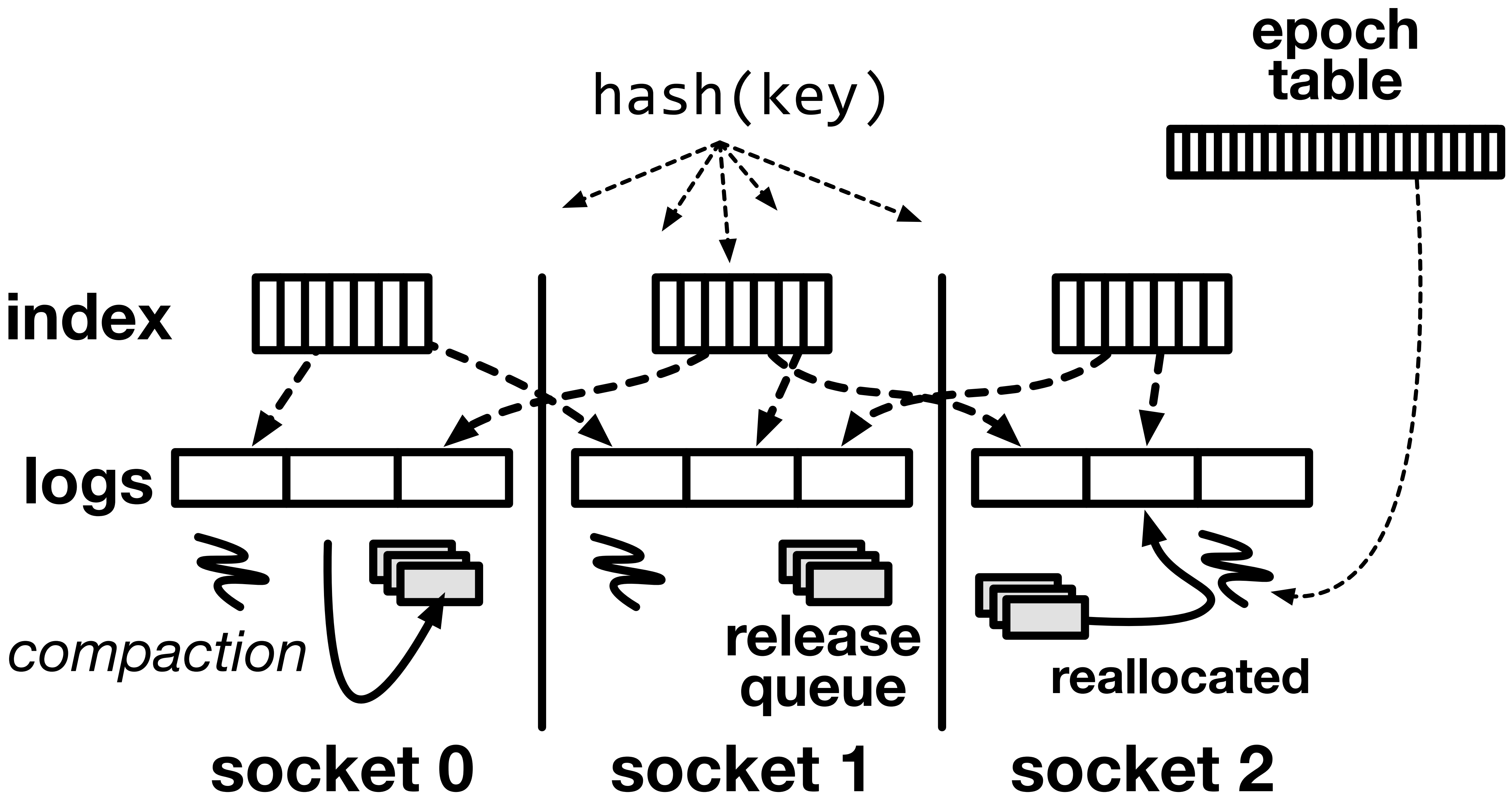
Why exclusively write local memory?



- **1.75x - 2.48x**
- 94% of global ops are remote

Uniform YCSB 240 threads (Mix is a 50:50 ratio of PUT and GET)

Nibble — Summary



Future Work

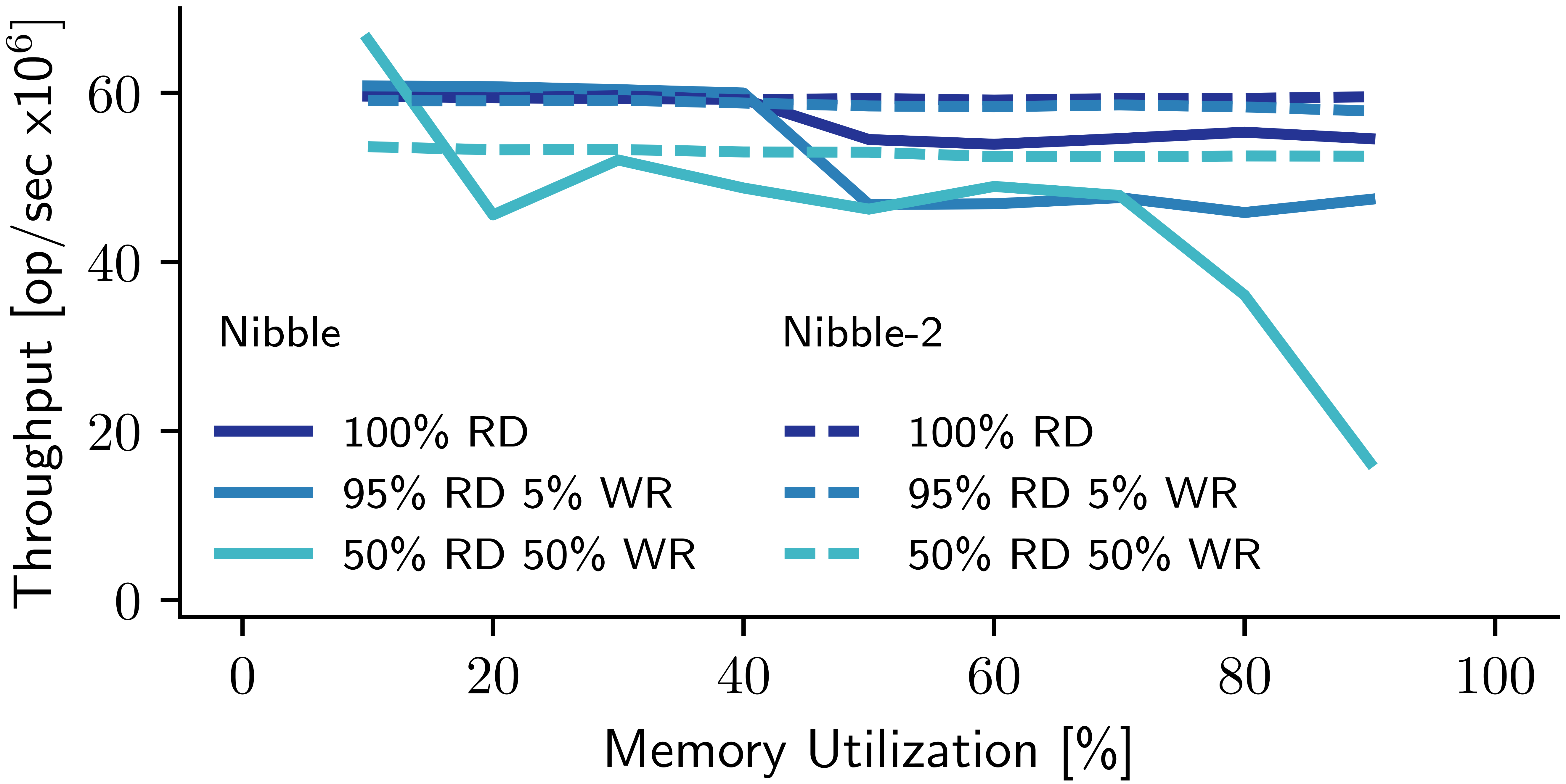
- Multi-object transactions
- Data > DRAM
- Main-memory file systems

Thank you!

`https://github.com/gtkernel/nibble-lsm`

Backup Slides

Can We Improve Performance at High Capacity?



smaug3. Same YCSB configuration as in prior experiments. 135 threads 1 KiB objects 8 compaction threads.

- All PUT **overwrite** existing objects
  - No subsequent compaction
  - Performance similar to MICA
- Subtle difference is updates in Nibble-2 cannot be done to local memory each time

Related Work

Scalable / Concurrent Object Stores

MICA [NSDI'14]
Masstree [Eurosys'12]
OpLog [MIT-TR'14]
CPHash [MIT-TR'11]
Cuckoo [EuroSys'14]
Anna [ICDE'18]
FASTER [SIGMOD'18]

Log-structured Memory

Memshare [ATC'17]
LSM [FAST'14]
Bwtree [ICDE'13]
cLSM [EuroSys'15]
TRIAD [ATC'17]
LogNVMM [ATC'17]

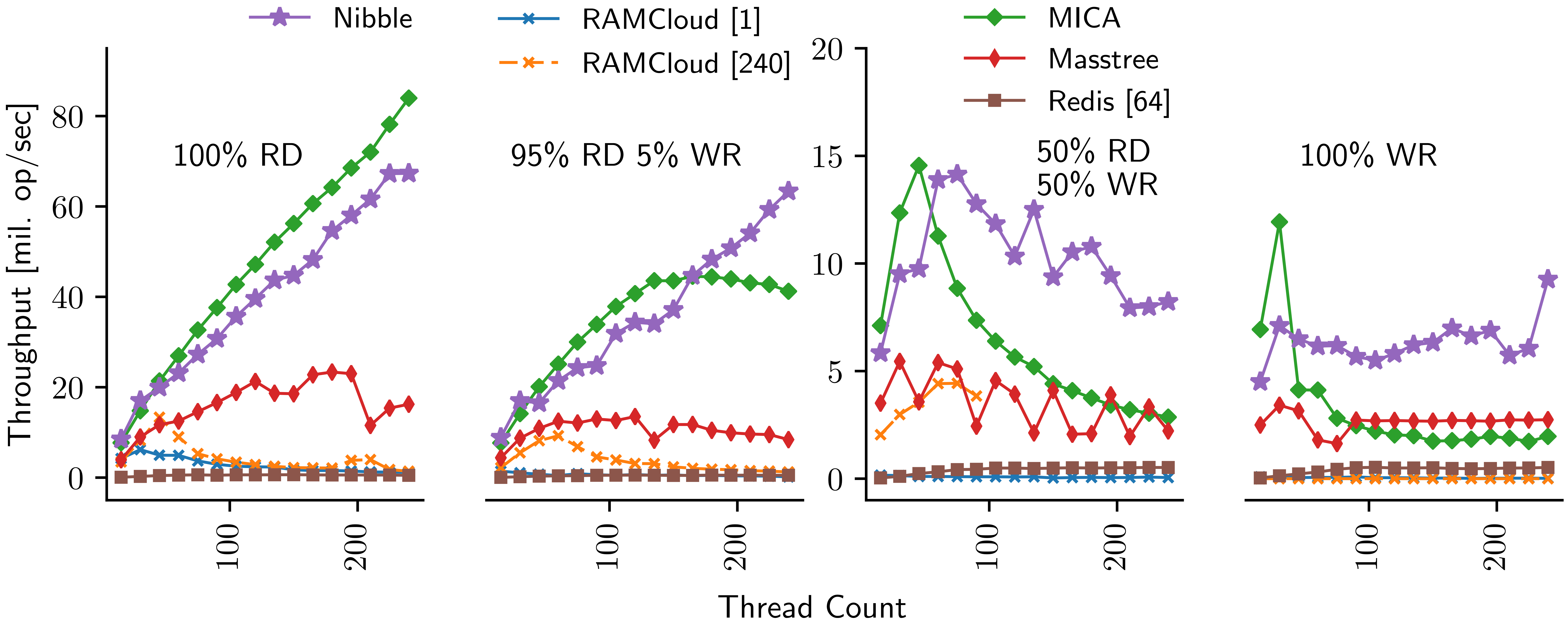
Memory bandwidth

Shoal [ATC'15]
Carrefour [ASPLOS'13]
BATMAN [GT-TR'15]

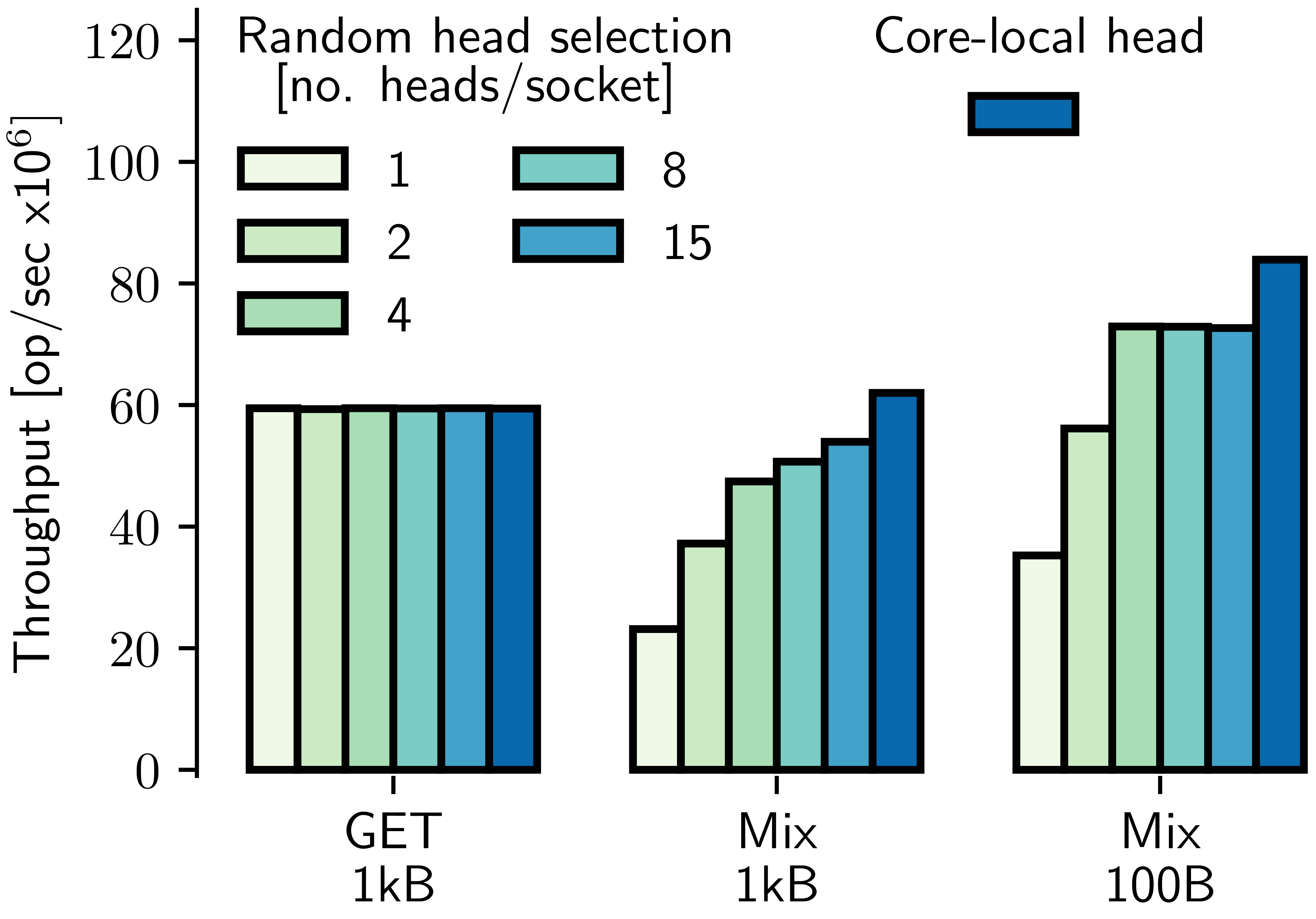
Scalability / Concurrent Programming

CST Locks [ATC'17]
ASCY [SOSP'13, ASPLOS'15]
Optik [PPoP'16]
Broadcast Trees [OSDI'16]

Static Data Sets — YCSB Zipfian

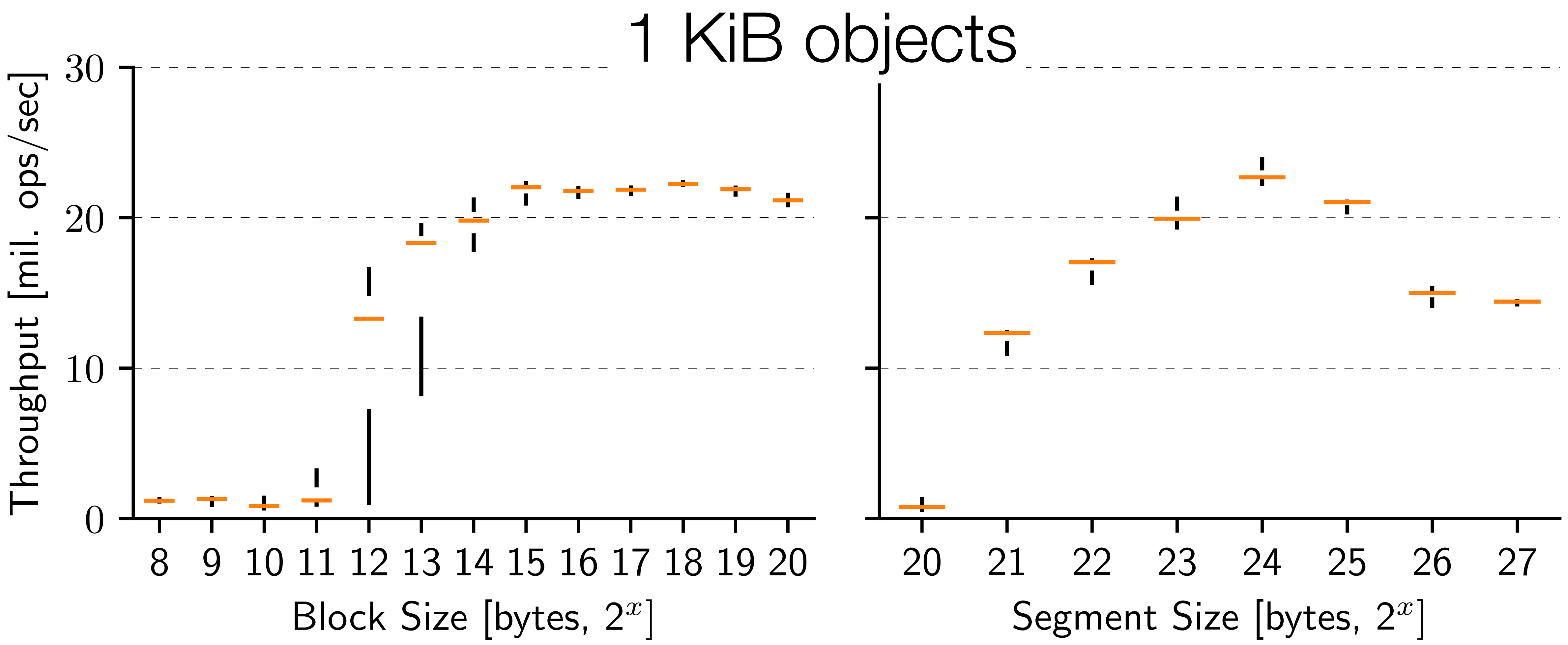


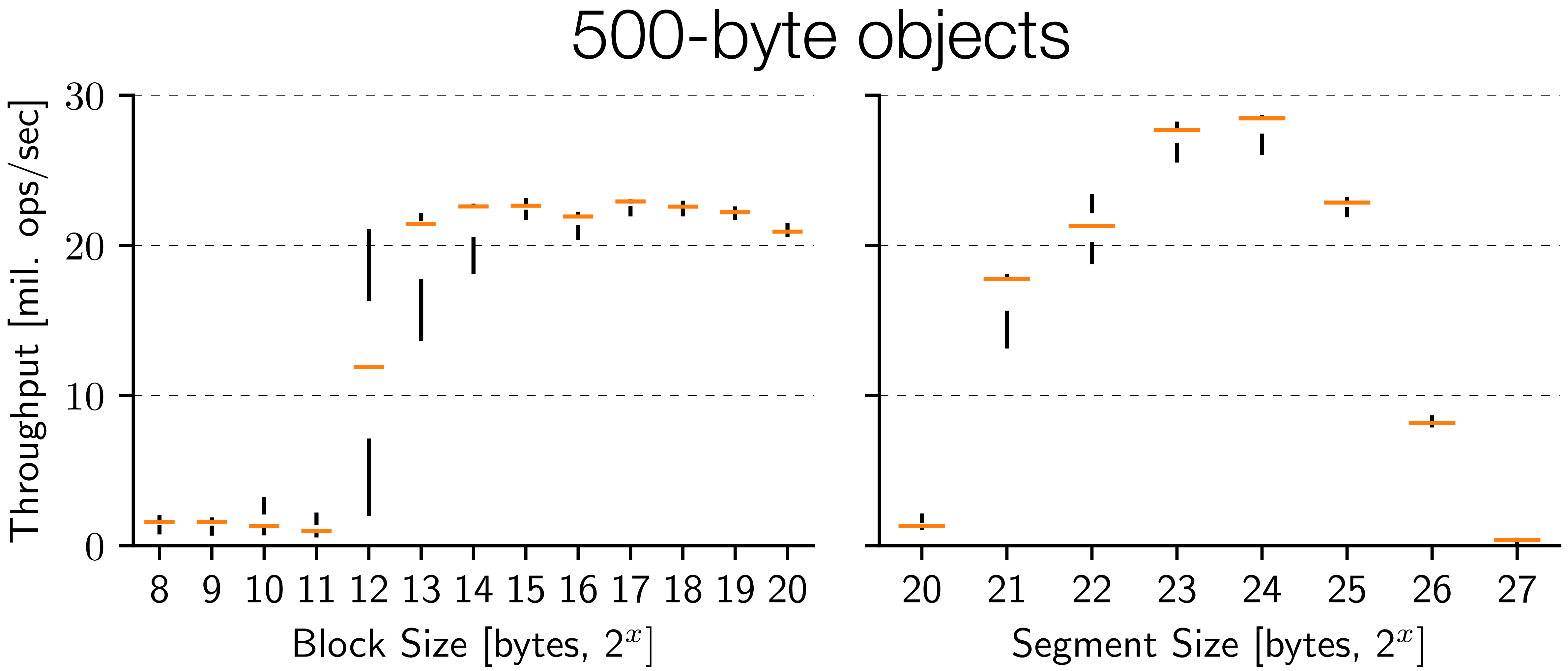
Multi-Head



- How useful to use multiple heads?
- Vary 1 to 15 heads
    - Choose head randomly, or assign head to core (choose with `rdtscp`)
- +15% with core-local head vs random

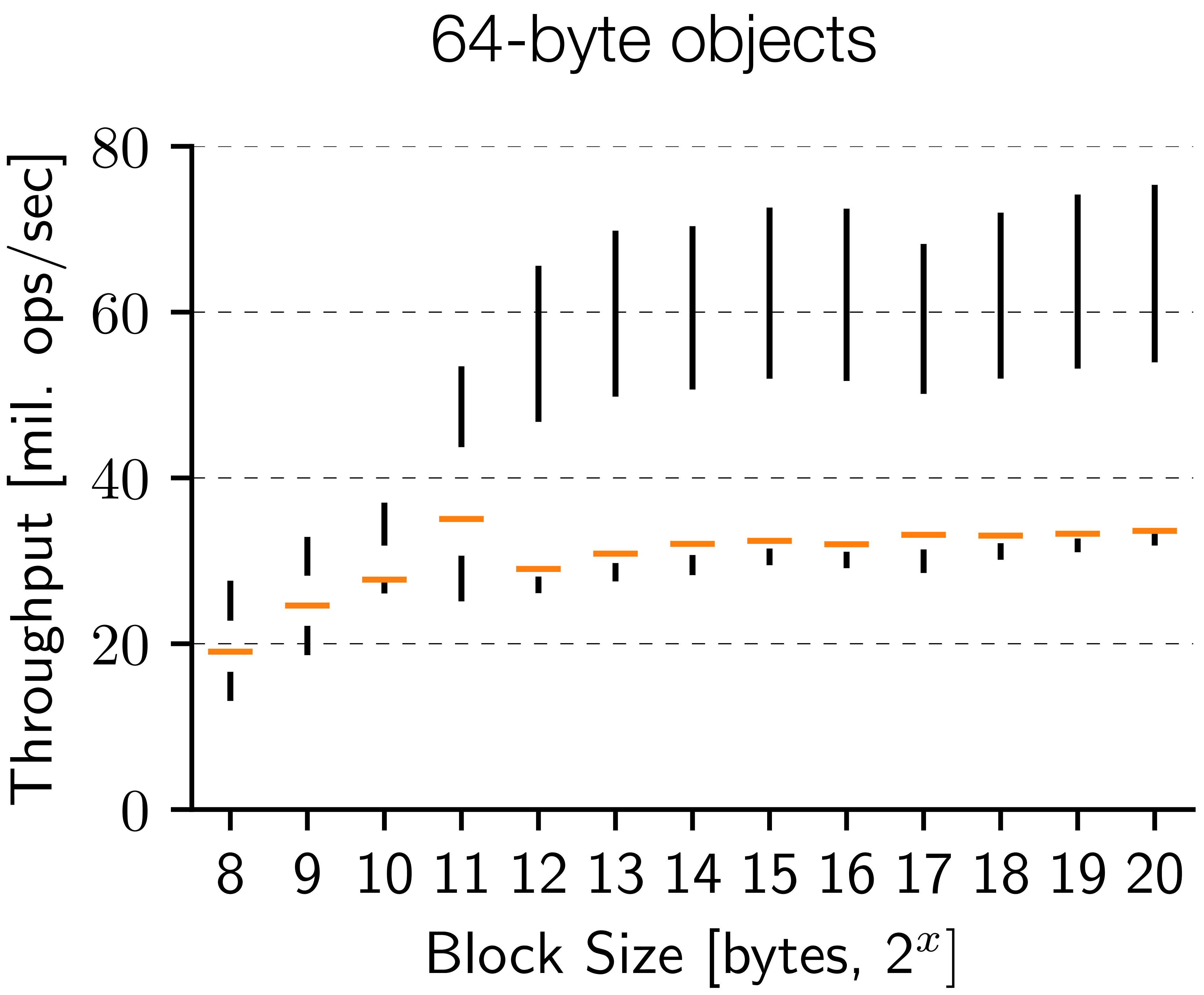
Effects of Block and Segment Sizes


1 KiB objects


500-byte objects

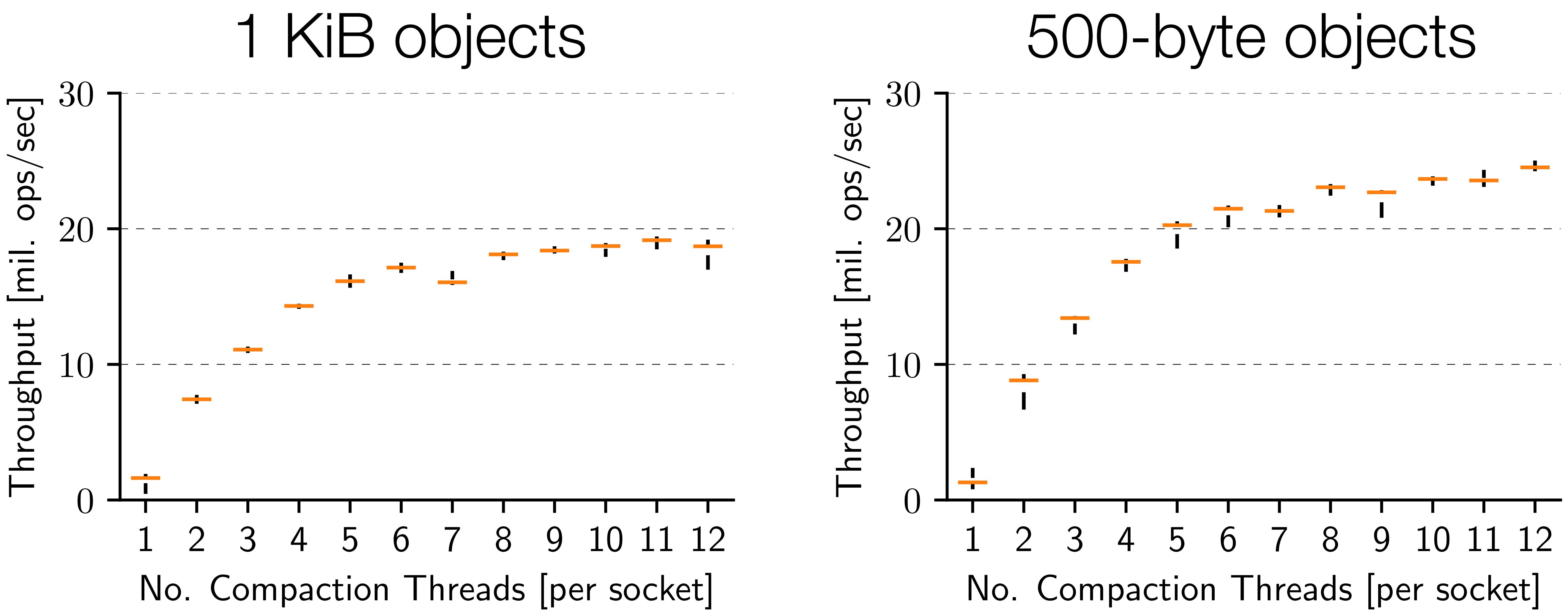
- Small blocks are very costly
  - Destructing and constructing segments
  - Objects split into many pieces
- Segment sizes have interesting “sweet spot”
  - Too small — frequent recycling
  - Too big — greater latency to compact

Effects of Block and Segment Sizes



- Small blocks are very costly
  - Destructing and constructing segments
  - Objects split into many pieces
- Segment sizes have interesting “sweet spot”
  - Too small — frequent recycling
  - Too big — greater latency to compact

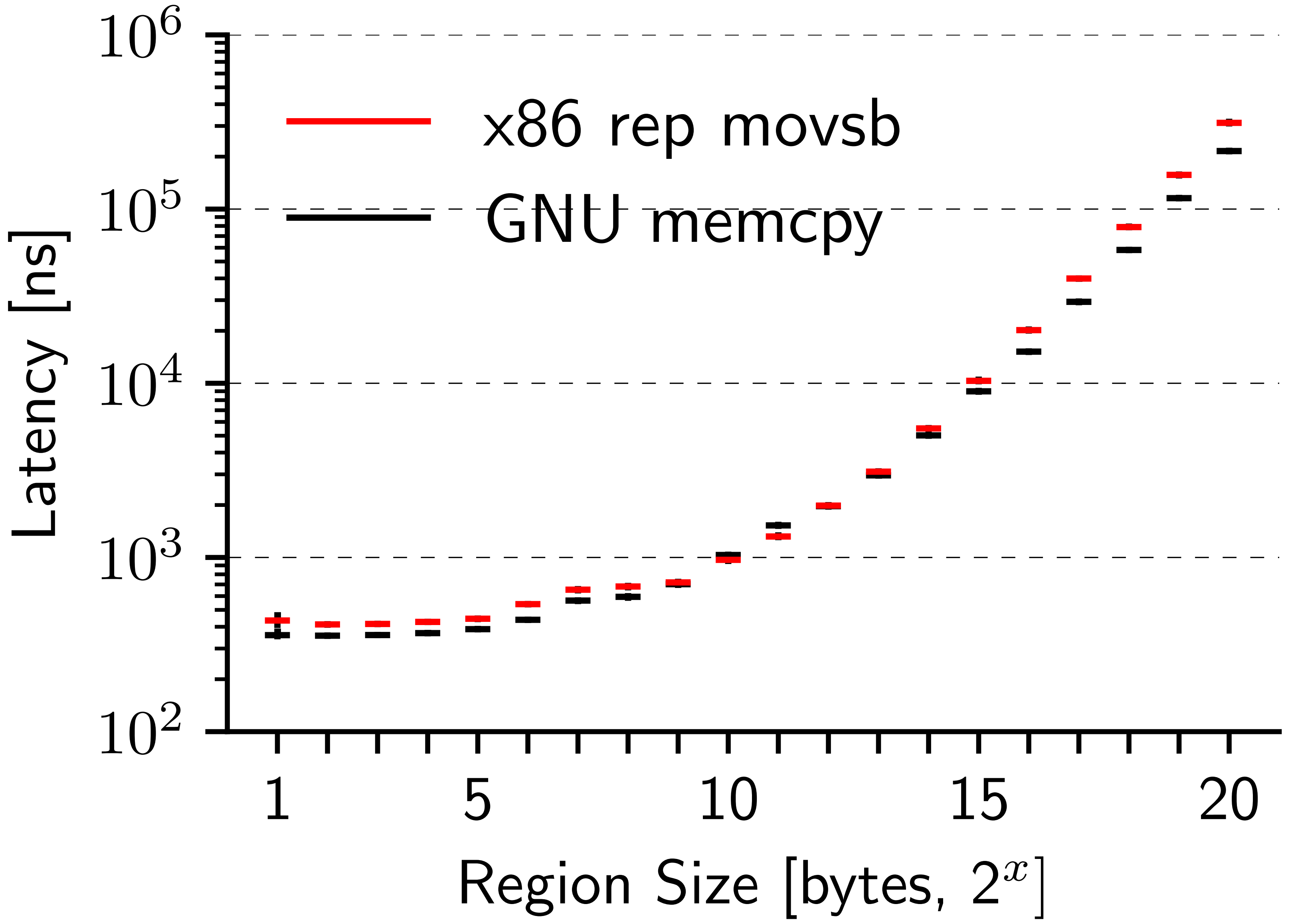
Effect of Compaction Parallelism

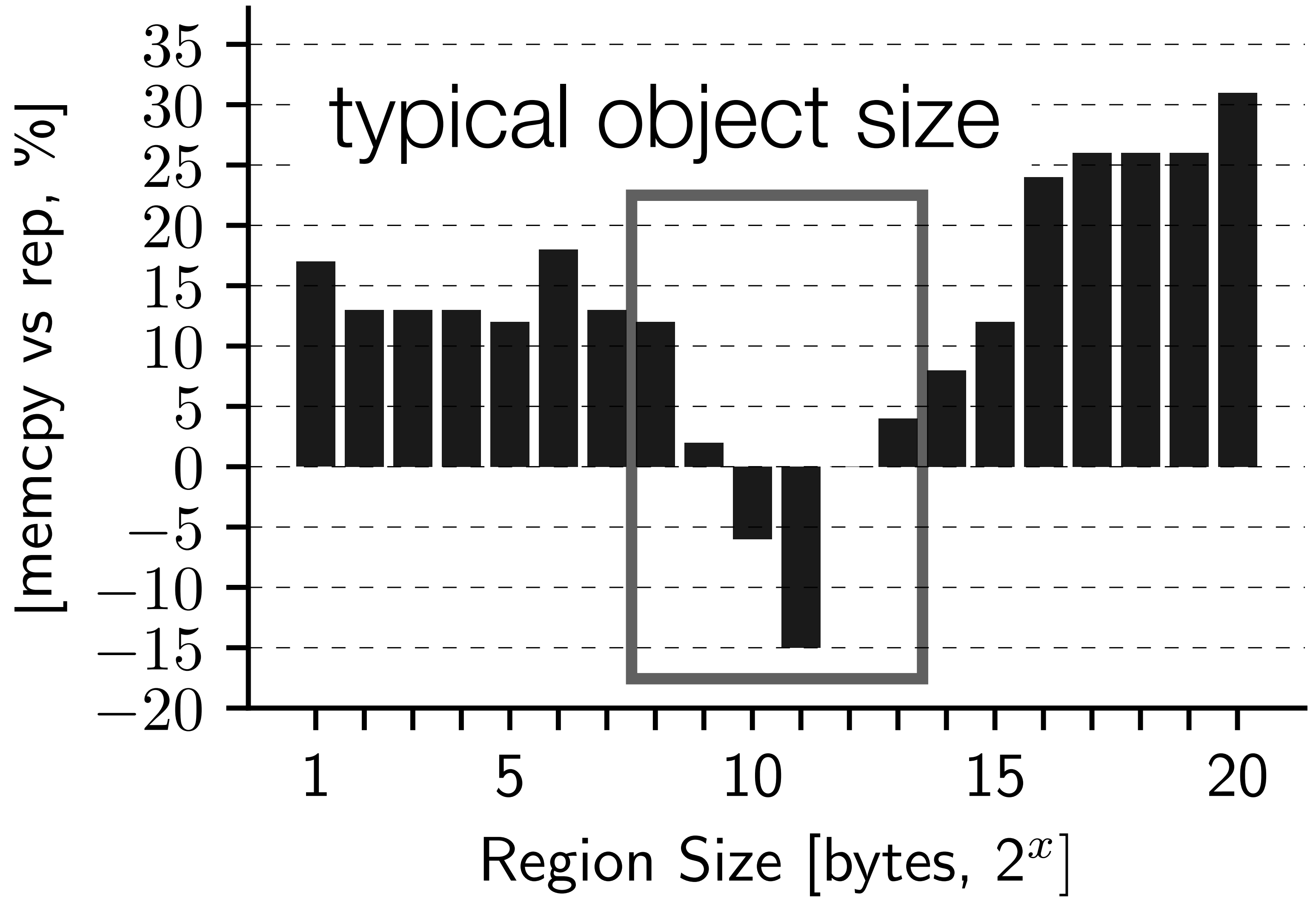


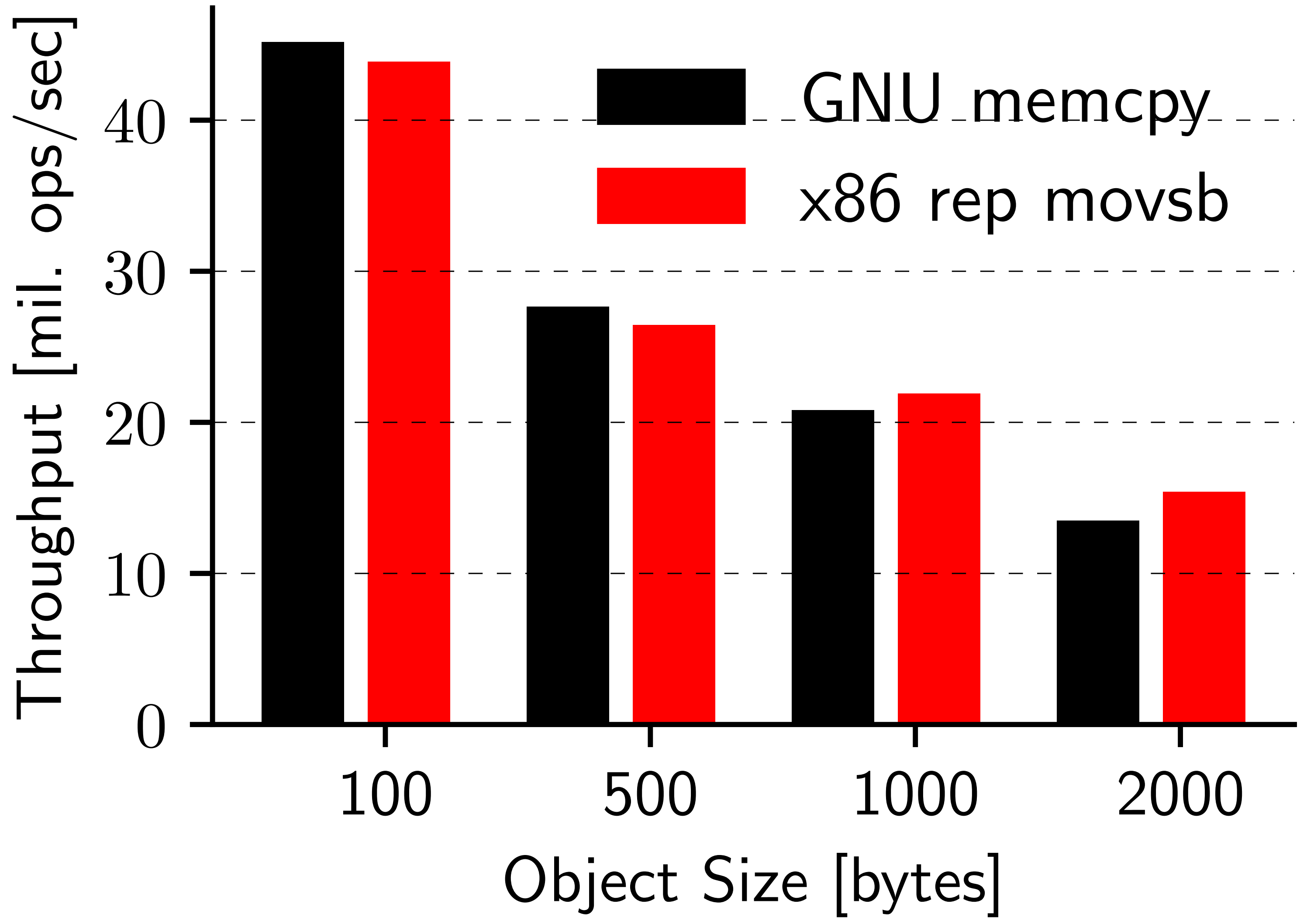
smaug3. Each CPU has 18 cores,
14 used by client threads.

- Parallelism in compaction helps significantly
  - **20x compared to a single thread**
- Diminishing returns
- In Nibble, segments waiting for reclamation are assigned uniformly among threads, to balance load

Can Memory Copying Mechanisms Help?

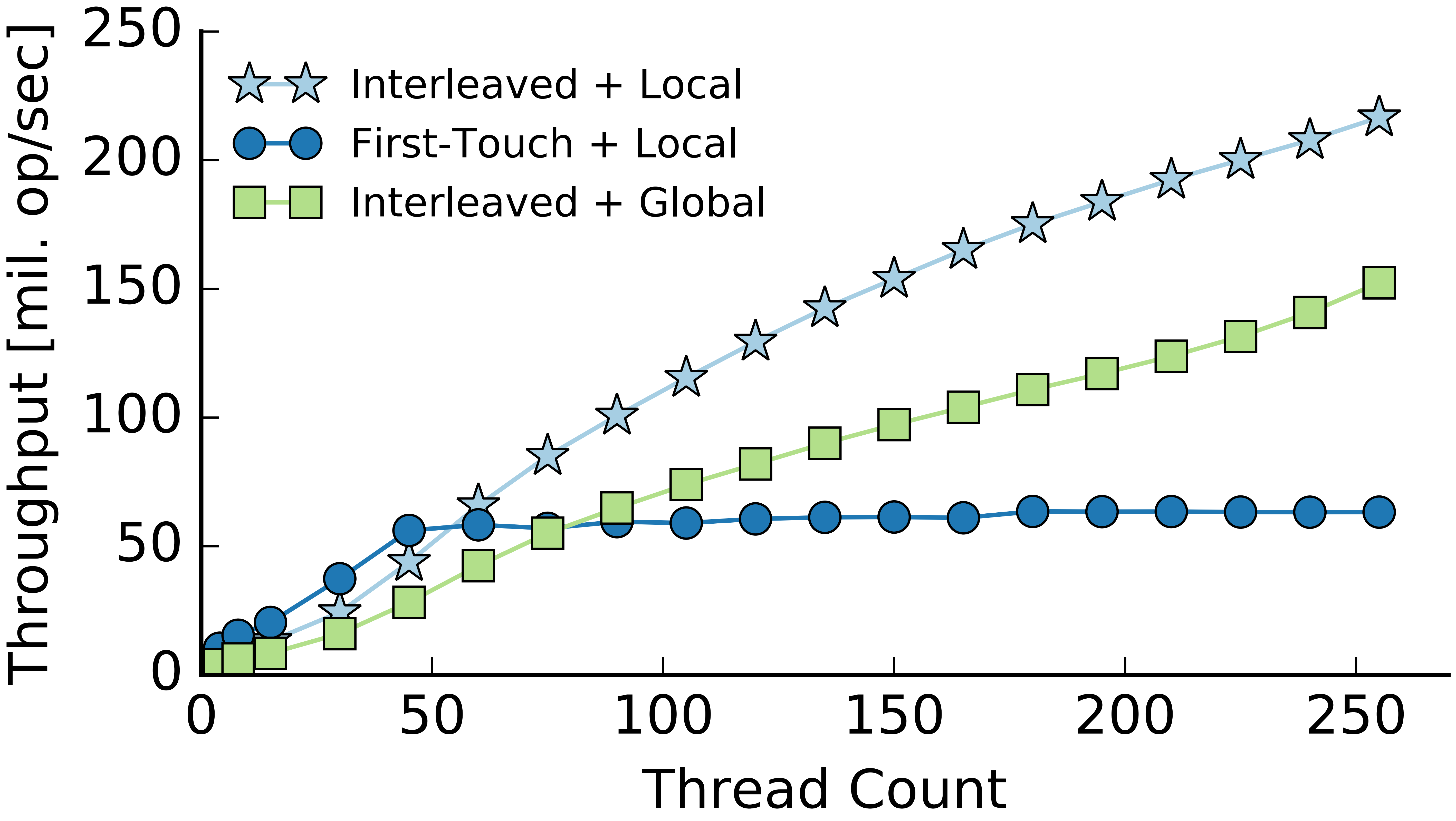




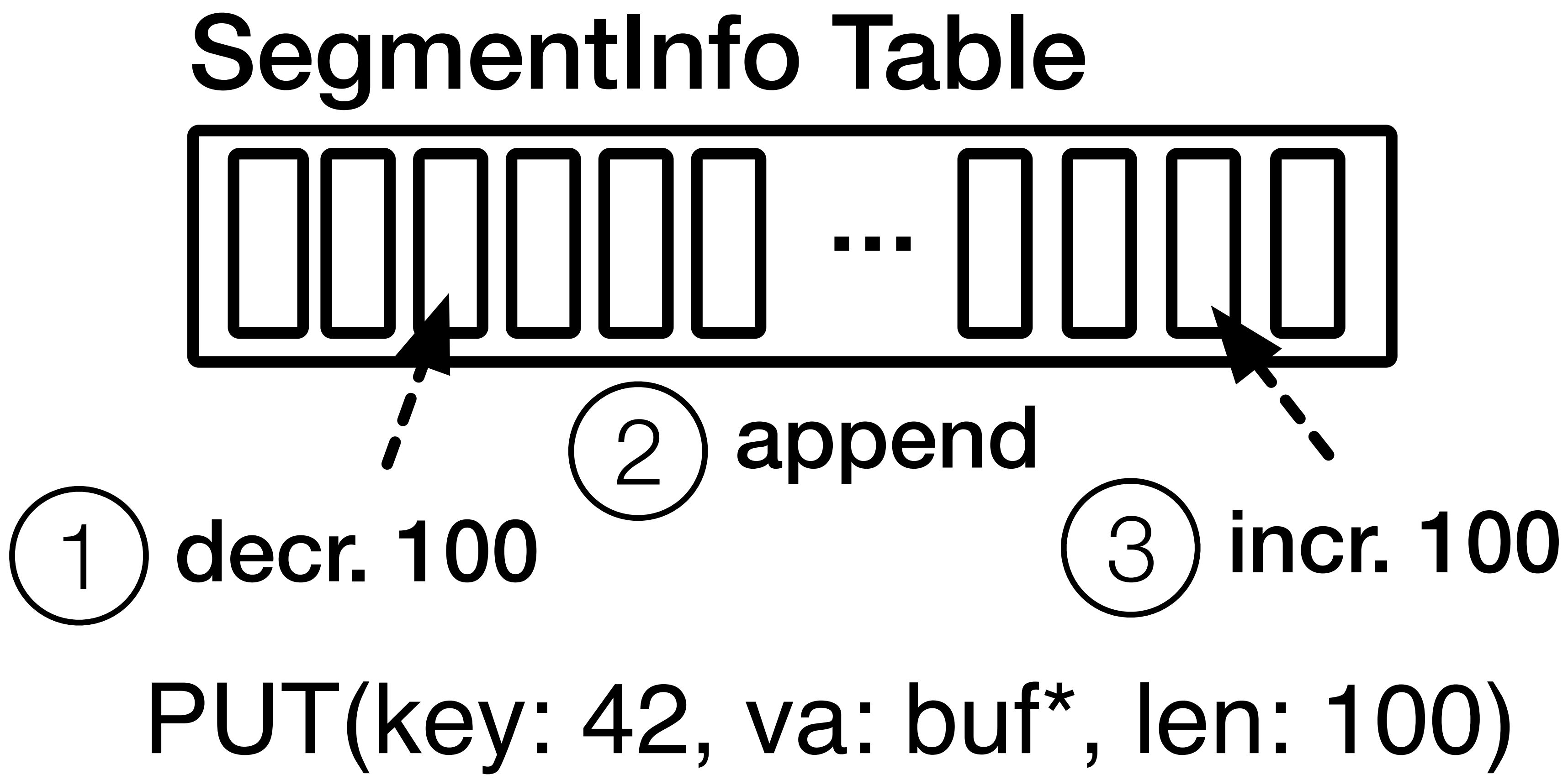


- We compare GNU libc memcpy with x86 instruction `rep movsb`
- Left two figures measure unloaded latency as region size varies
  - Allocate large region and randomly select source and destination offsets on local memory — 7 iterations
- Right figure shows end-to-end impact on use

Bandwidth Saturation — Index



Tracking Segment Sizes



SegmentInfo Table

① **decr. 100**

② **append**

③ **incr. 100**

PUT(key: 42, va: buf*, len: 100)

- Each log has table of Segment metadata
- `size` attribute managed atomically
  - Segment size monotonically decreases after closed

Candidate Segment Selection

$$benefit_i = \frac{(1 - util_i) \cdot age_i}{1 + util_i} \text{ where } util_i = \frac{live_i}{length_i} \text{ for segment } i$$

- Recalculate when segment size changes
  - Obtained from SegmentInfo table
- Compaction selects segment based on: age, current capacit
- Each socket has multiple threads
  - Closed segments assigned round-robin
  - Equal load, and trivially parallelize candidate selection