An Efficient Memory-Mapped Key-Value Store for Flash Storage

Anastasios Papagiannis, Giorgos Saloustros,
Pilar González-Férez, and Angelos Bilas

Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)
Greece

Saving CPU Cycles In Data Access

- Data grows exponentially
  - Seagate report claims that data grow 2x every 2 years
- Need to process more data with same number of servers
  - Cannot increase number of servers - power, energy limitations
- Data access for data serving/analytics incurs high cost
- Today key-value stores used broadly for data access
  - Social networks, data analytics, IoT
  - Consume a lot of CPU cycles/operation - Optimized for HDDs
- Important to reduce CPU cycles in key value stores

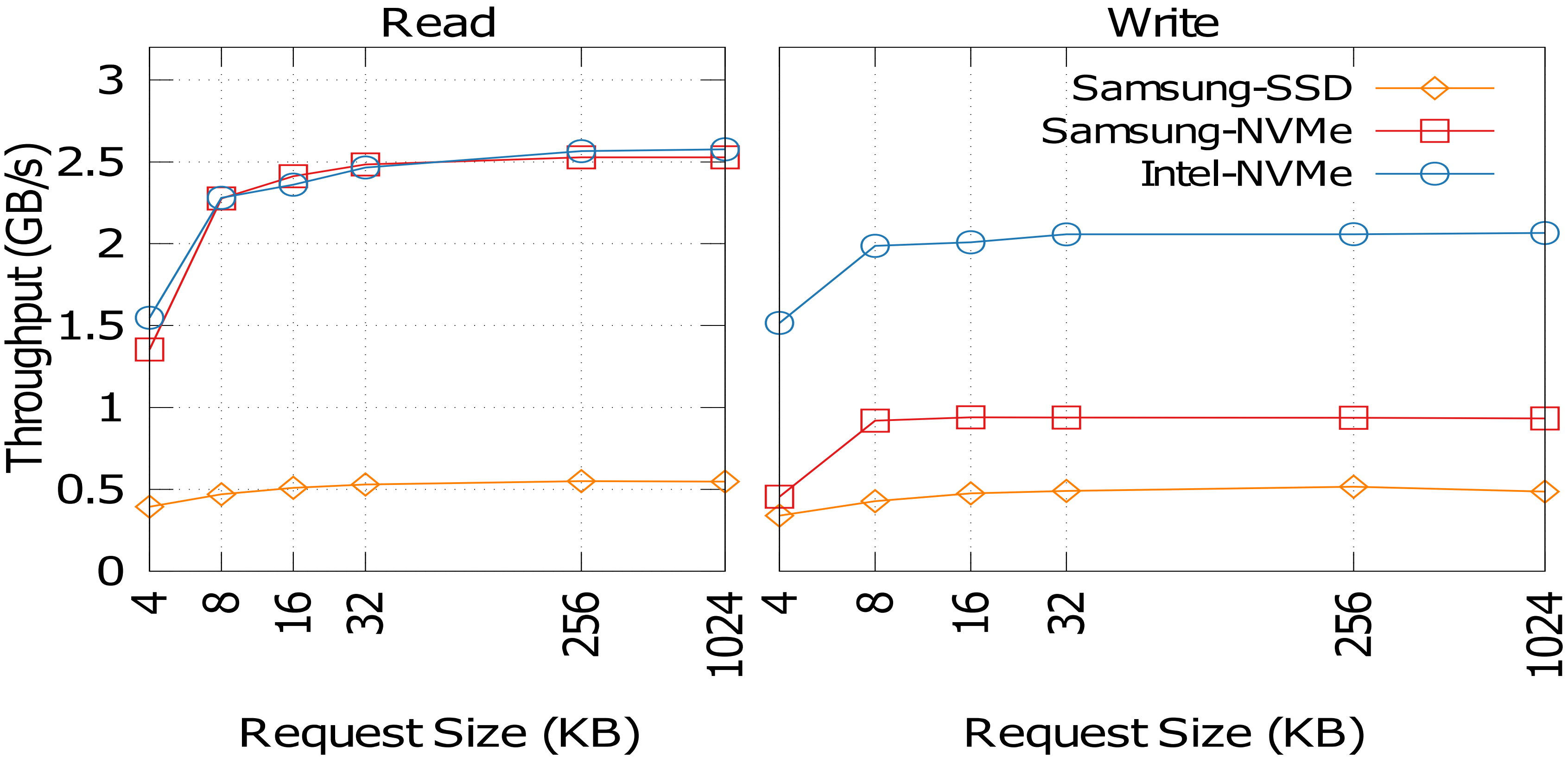
Dominant indexing methods

- Inserts are important for key-value stores
  - Reads consist the majority of operations
  - However, need to handle bursty inserts of variable size items
- B-tree optimal for reads
  - Needs a single I/O per insert as the dataset grows
- Main approach: Buffer writes in some manner
  - … and use single I/O to the device for multiple inserts
  - Examples: LSM-Tree, B^{ε}-Tree, Fractal Tree
- Most popular: LSM-Tree
  - Used by most key value stores today
  - Great for HDDs - always perform large sequential I/Os

New Opportunities: From HDDs To Flash

- In many applications fast devices (SSDs) dominate
- Take advantage of device characteristics to increase serving density in key value stores
  - Serve same amount data with less cycles
- High throughput even for random I/Os at high concurrency

SSDs Performance For Various Request Sizes



User Space Caching Overhead

- User space cache: no system calls for hits - explicit I/O for misses
- Copies from user to kernel space during I/O
- Hits incur overhead in user-space index+data in every traversal

Our Key Value Store: Kreon

- In this paper we deal with two main sources of overhead
  - Aggressive data reorganization (compaction)
  - User-space caching
- We increase I/O randomness for reducing CPU cycles
- We use memory-mapped I/O instead of a user-space cache

Outline of this talk

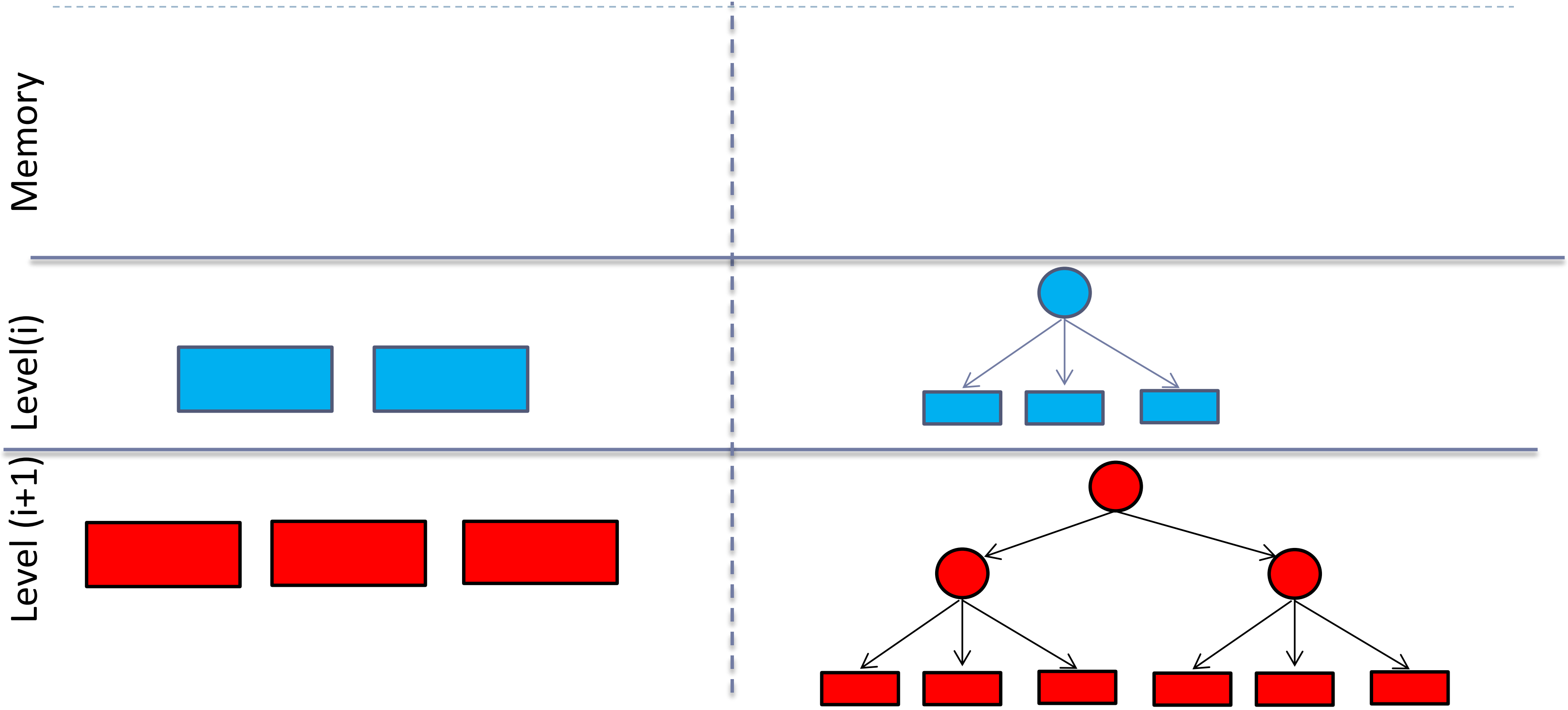
- Motivation
- **Discuss Kreon design and motivate decisions**
  - Indexing data structure
  - DRAM caching and I/O to devices
- Evaluation
  - Overall Efficiency – Throughput
  - I/O amplification
  - Efficiency breakdown
  - Tail latency

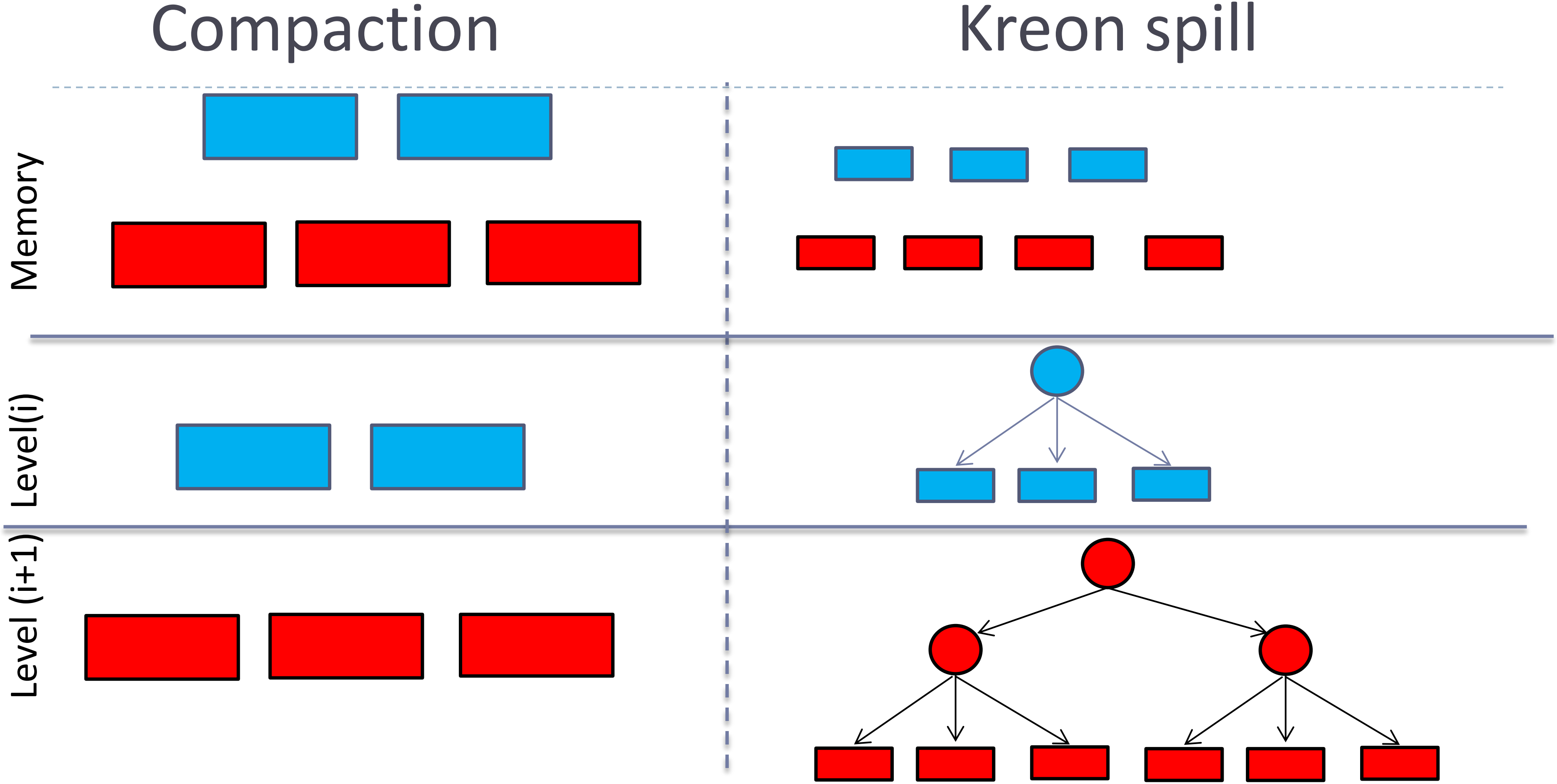
Kreon Persistent Index

- Kreon introduces partial reorganization
- Allows to eliminate sorting [bLSM’12]
  - Key value pairs stored in a log [Atlas’15, WiscKey ‘16, Tucana’16]
  - Index organized in unsorted levels /B-tree index per level
- Efficient merging – Spill
  - Reads less data from of L_{i+1} compared to LSM
  - Inserts take place in buffered mode as in LSM

Compaction

Kreon spill




Compaction
Kreon spill
Memory
Level(i)
Level (i+1)

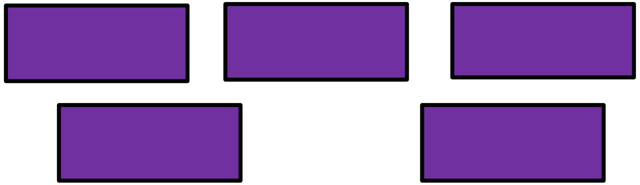
Compaction

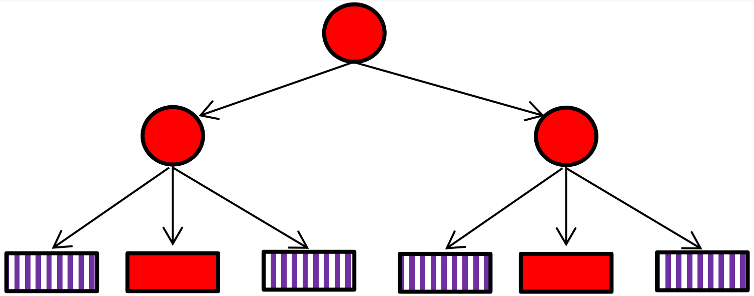
Kreon spill

Memory

Level(i)

Level (i+1)





Kreon Performs Adaptive Reorganization

- With partial reorganization repeated scans are expensive
  - With repeated scans, it is worth to fully organize data
- Kreon reorganizes data during scans
  - Based on policy (current threshold based)

Reduce caching overheads with memory mapped I/O

- Avoid overhead of user-kernel data copies
- Lower overhead for hits by using virtual memory mappings
  - Either served from TLB or page table traversal
- Eliminates serialization with common layout in memory and storage
- Using memory mapped I/O has two implications
  - Requires common allocator for memory and device
  - Linux kernel mmap introduces challenges

Challenges of Common Data Layout

- Small random read less overhead with mmap
- Log writes large – irrelevant
- Index updates could cause 4K random writes to device
  - Kreon generates large writes by using Copy-on-Write and extent allocation on device
- Recovery with common data layout
  - Requires ordering operations in memory and on device
  - Kreon does this with CoW and sync
- Extent allocation works well with common data layout in key value stores
  - Spills generate large frees for index
  - Key value stores usually experience group deletes

mmap Challenges for Key Value Stores

- Cannot pin L_0 in memory
  - I/O amortization relies on L_0 being in memory
  - Prioritize index nodes across levels and with respect to log
- Unnecessary read-modify write operation from device
  - Writes to newly allocated pages no need to read them
- Long pauses during user requests and high tail latency
  - mmap performs lazy memory cleaning and results in bursty I/O
  - Persistence requires msync which uses coarse grain locking

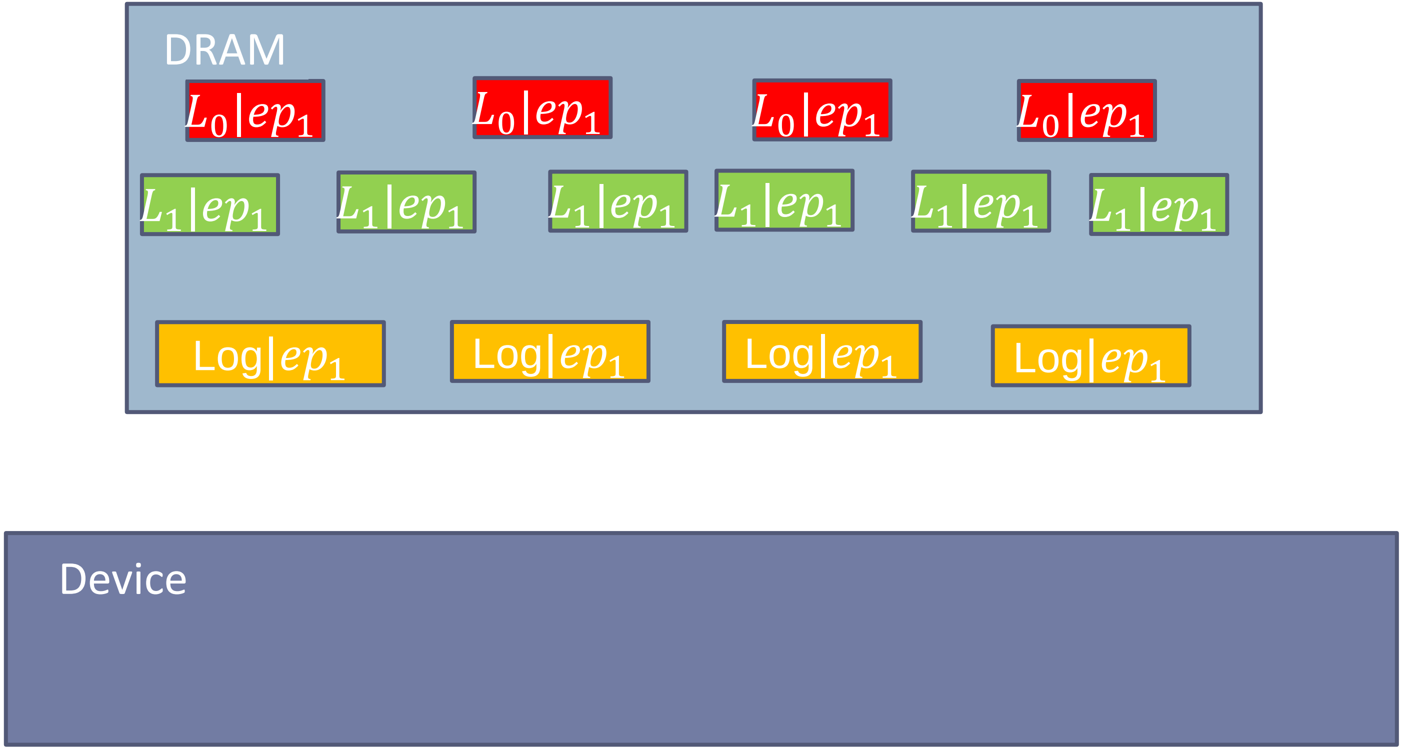
Kreon Implements a custom mmap path

- Introduces per page priorities
  - Separate LRUs per priority
  - L_0 most significant priority, index, log
- Detects accesses to new pages and eliminates device fetch
  - Keeps a non persistent bitmap with page status (free/allocated)
  - Bitmap updated by Kreon’s allocator
- Improved tail latency
  - kmmap adds bounds in memory used
  - Eager eviction policy
  - Higher concurrency in msync

Kreon increases concurrency during msync

- msync orders writing and persisting pages by blocking
- Opportunity in Kreon
  - Due to CoW the same page is never written/persisted concurrently
- Kreon orders by using epochs
- msync evicts all pages of previous epoch
- Newly modified pages belong to new epoch
- Epochs are possible in Kreon due to CoW

kmmap Operation


DRAM

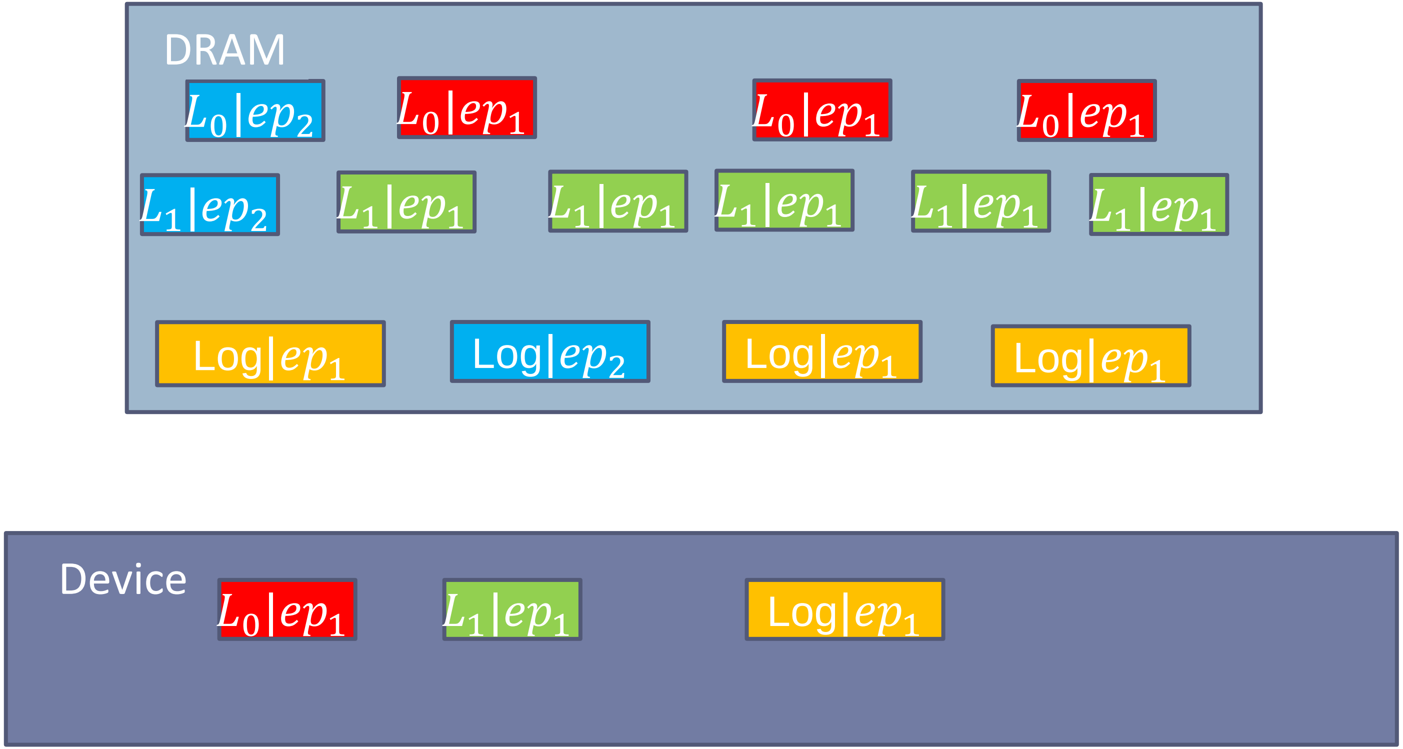
$L_0|ep_1$ $L_0|ep_1$ $L_0|ep_1$ $L_0|ep_1$

$L_1|ep_1$ $L_1|ep_1$ $L_1|ep_1$ $L_1|ep_1$ $L_1|ep_1$ $L_1|ep_1$

Log$|ep_1$ Log$|ep_1$ Log$|ep_1$ Log$|ep_1$

Device

kmmap Operation



DRAM

$L_0|ep_2$ $L_0|ep_1$ $L_0|ep_1$ $L_0|ep_1$

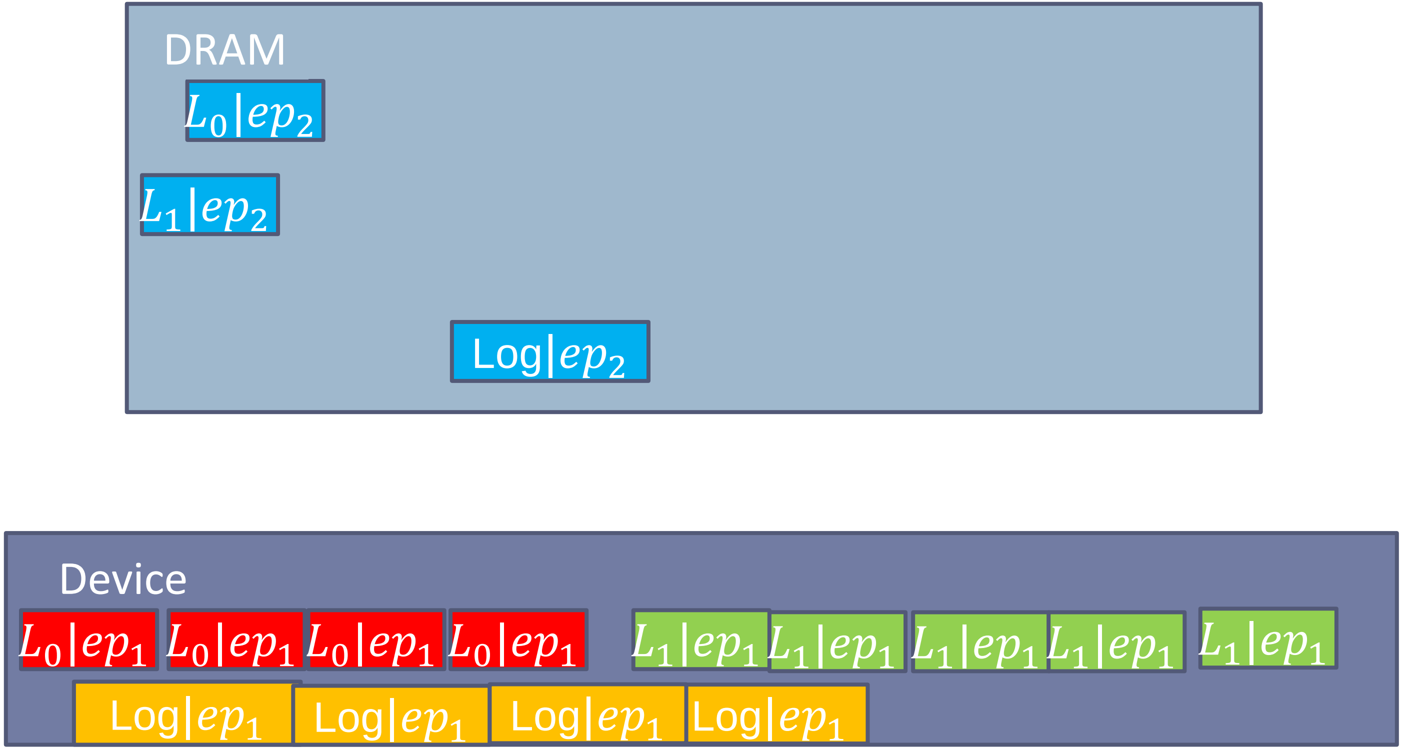
$L_1|ep_2$ $L_1|ep_1$ $L_1|ep_1$ $L_1|ep_1$ $L_1|ep_1$ $L_1|ep_1$

Log$|ep_1$ Log$|ep_2$ Log$|ep_1$ Log$|ep_1$

Device

$L_0|ep_1$ $L_1|ep_1$ Log$|ep_1$

kmmap Operation



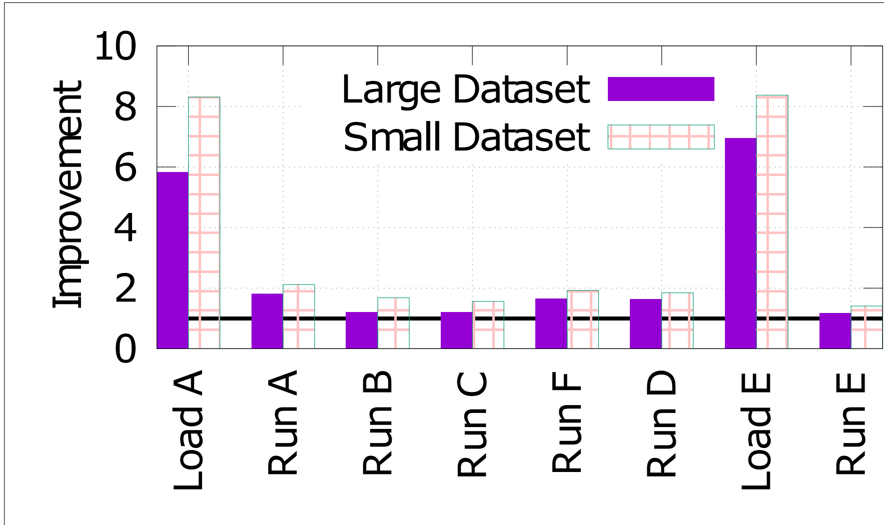
Outline of this talk

- Motivation
- Discuss Kreon design and motivate decisions
  - Indexing data structure
  - DRAM caching and I/O to devices
  - Persistence and failure atomicity
- **Evaluation**
  - Overall efficiency – throughput
  - I/O amplification
  - Tail latency
  - Efficiency breakdown

Experimental Setup

- Compare Kreon with RocksDB version 5.6.1
- Platform
  - Two Intel Xeon E5-2630 with 256GB DRAM in total
  - Six Samsung 850 PRO (256GB) in RAID-0 configuration
- YCSB
  - Insert only, read only, and various mixes
- We examine two cases
  - Dataset contains 100M records resulting in a 120 GB dataset
  - Two configurations: small uses 192 GB of DRAM large uses 16 GB

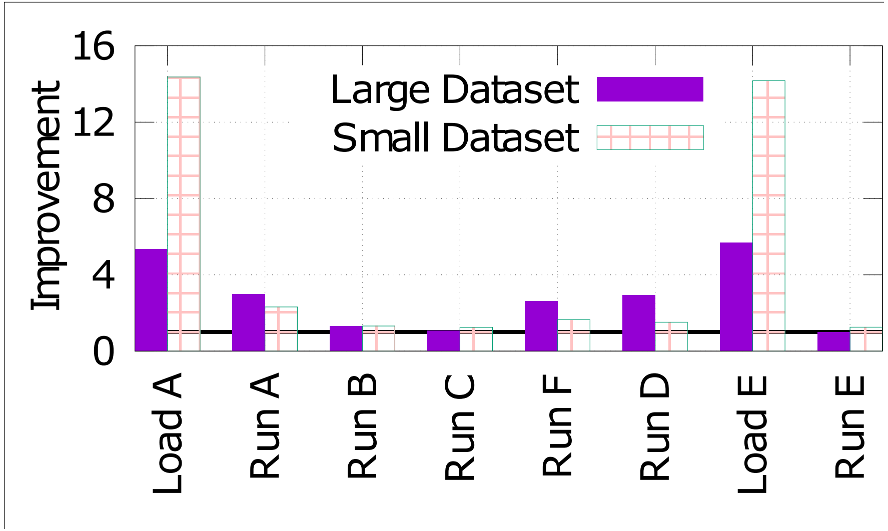
Overall Improvement over RocksDB



(a) Efficiency (cycles/op)

Small up to 6x - average 2.7x,

Large up to 8.3x - average 3.4x

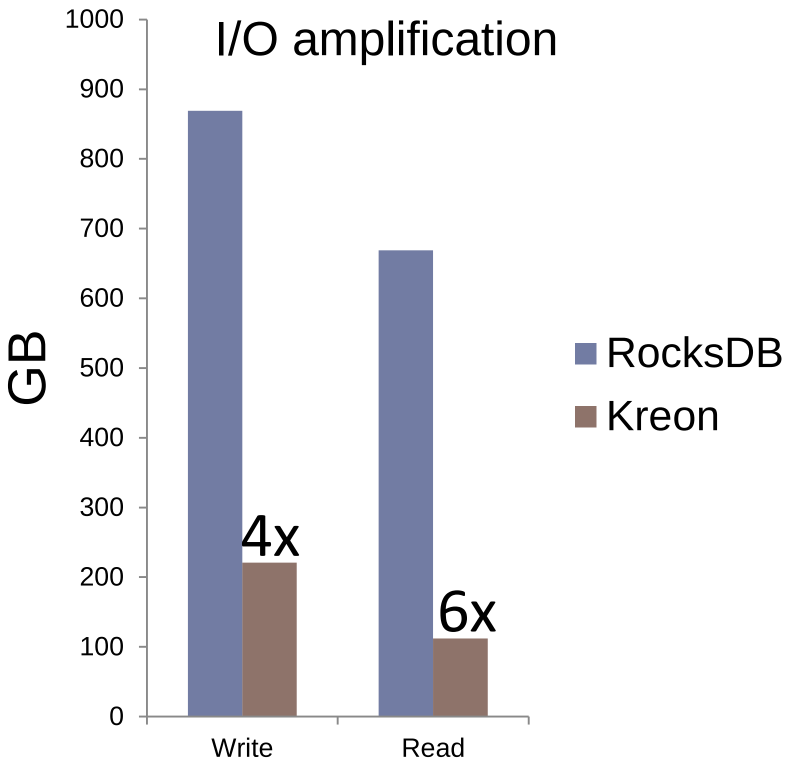


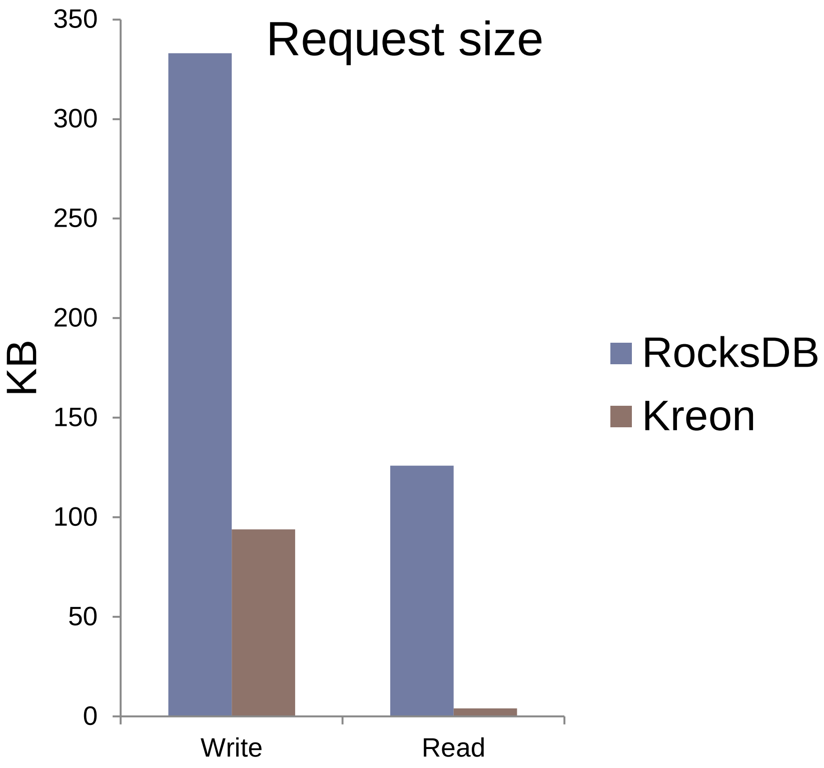
(b) Throughput (ops/s)

Small up to 5x - average 2.8x,

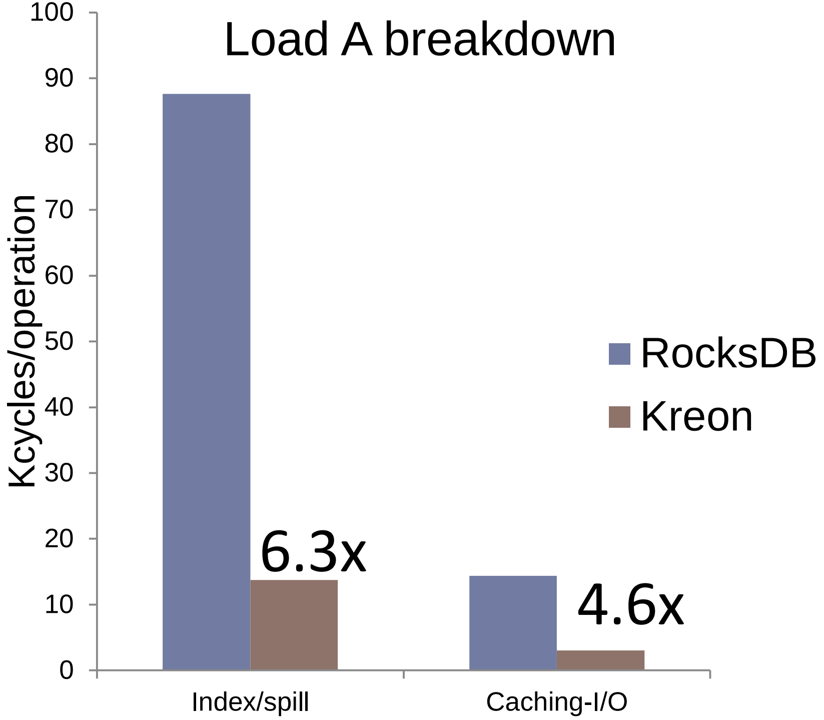
Large up to 14x - average 4.7x

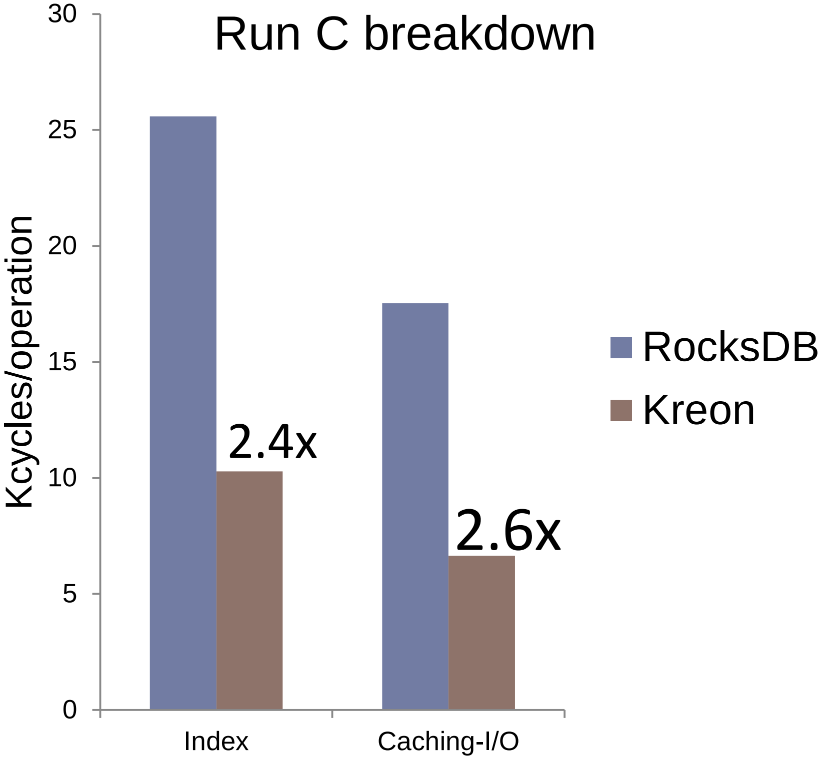
I/O amplification to devices





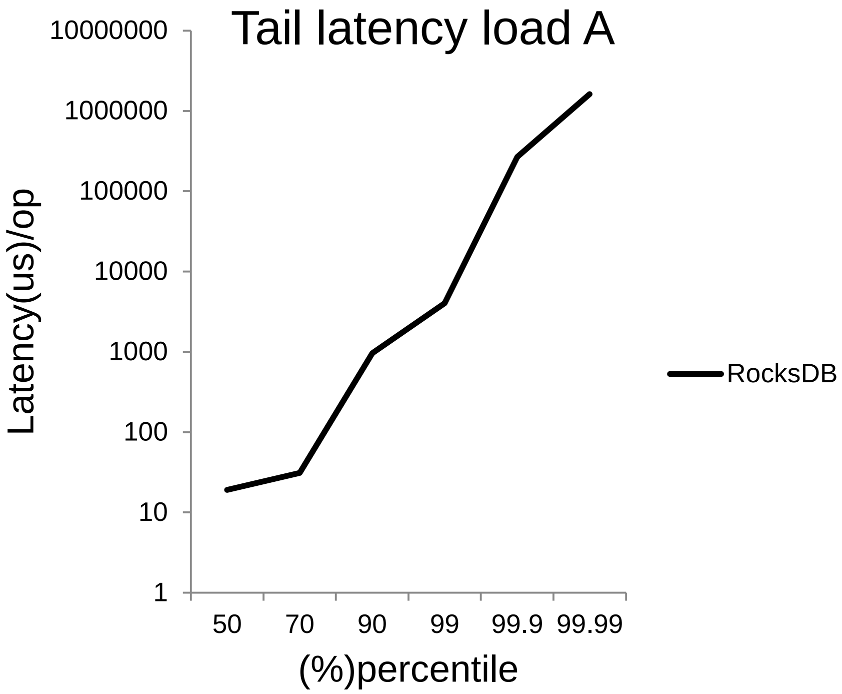
Contribution of individual techniques



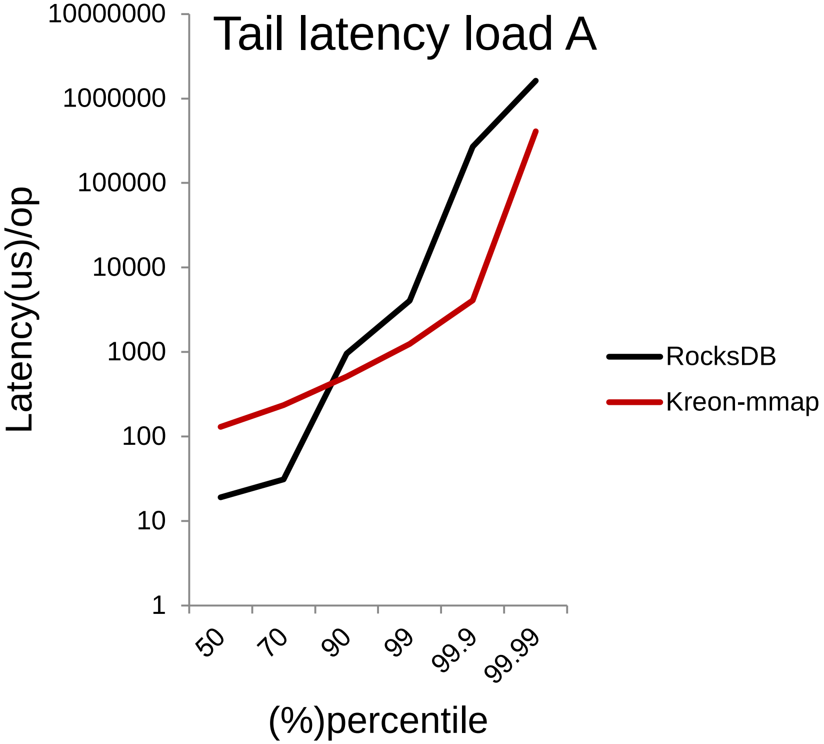


FORTH-ICS
Institute of Computer Science

kmmap impact on tail latency

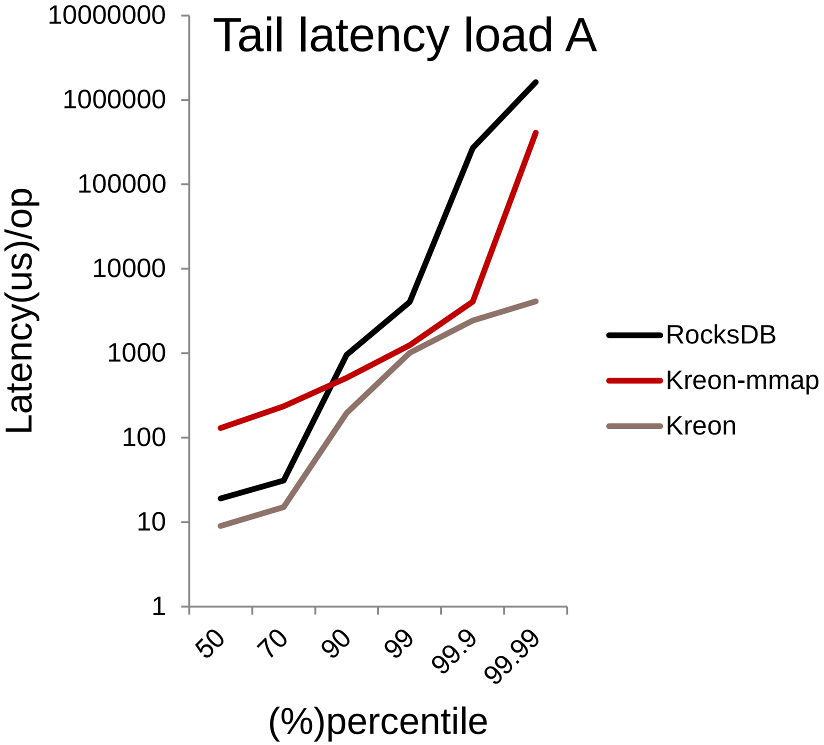


kmmap impact on tail latency



kmmap impact on tail latency

- 393x lower 99.99% tail latency than RocksDB
- 99x lower 99.99% tail latency than Kreon-mmap



Conclusions

- Kreon: An efficient key-value store in terms of cycles/op
  - Trades device randomness for CPU efficiency
  - CPU most important resource today
- Main techniques
  - LSM → Partially organized levels with full index per level
  - DRAM caching → via custom memory mapped I/O
- Up to 8.3x better efficiency compared to RocksDB
  - Both index and DRAM caching important

Questions ?

Giorgos Saloustros

Institute of Computer Science, FORTH – Heraklion, Greece

E-mail: gesalous@ics.forth.gr

Web: http://www.ics.forth.gr/carv

Supported by EC under Horizon 2020 Vineyard (GA 687628), ExaNest (GA 671553)