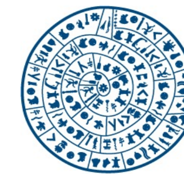
Efficient Key-Value Stores for Fast Storage Devices

Angelos Bilas (presenting the work of many others)

bilas@ics.forth.gr



FORTH

INSTITUTE OF COMPUTER SCIENCE

Department of Computer Science

University of Crete

Institute of Computer Science (ICS)

Foundation for Research and Technology – Hellas (FORTH)

Overview

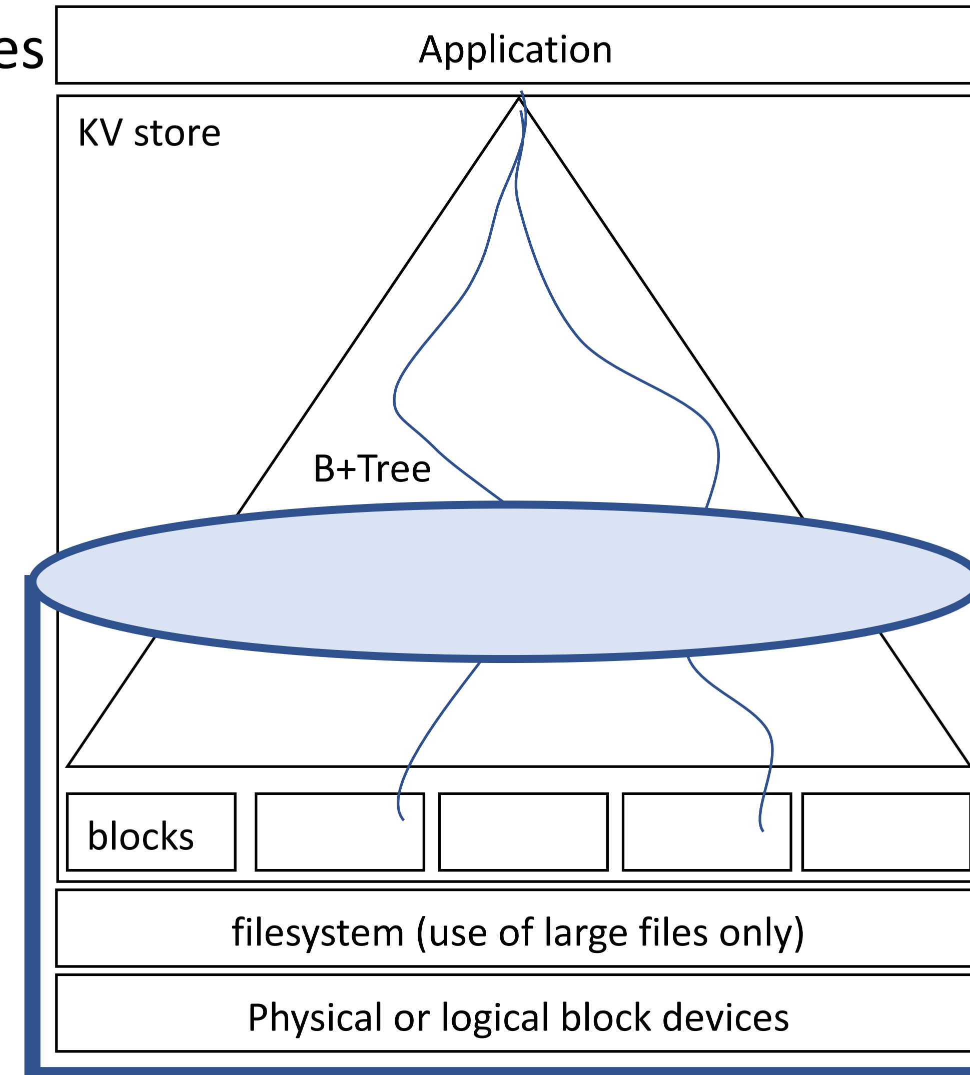
- What are key-value (KV) stores - Why?
- Our KV work at FORTH and the Univ. of Crete
  - Indexing – towards small SSTs [Tucana ATC'16, Kreon SoCC'18]
  - KV separation – differentiated treatment [Parallax SoCC'21]
  - Replication – shipping indexes [Tebis EuroSys'22]
  - Memory-mapped I/O caches (mmio) [FastMap ATC'20, Aquila EuroSys'21]
- Concluding remarks - What is the ideal KV store?

New storage trends – New problems

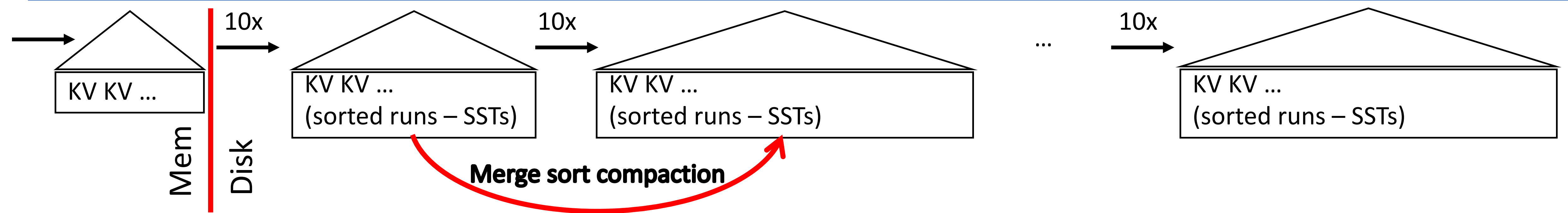
- Bottleneck in the I/O path shifts from device to host
  - With rotational disks (HDDs) CPU overhead is not significant
- Fast storage happened → NAND Flash, NVMe, NVM
  - Millions of IOPS, < 10 μs access latency, require a lot of CPU cycles
  - Form factor more convenient for certain uses (e.g. as local devices)
- Cloud happened as well → Applications evolve
  - Colocation/sharing (multiple apps), workflows, I/O sizes, data use patterns
- Today, I/O path remains a concern
- In this landscape, KV stores dominate – Why?

KV stores as storage engines

- *"New"* approach for organizing, accessing data on devices
  - Context: Rely on filesystem or KV-store for performance?
- Offer simple and versatile API to higher layers
  - Dictionary operations: put(), get(), delete(), and scan()
  - No offsets visible to applications
  - Further decouples logical/physical properties compared to FS
- Traditionally, most persistent I/O layers use a B+-tree
  - Optimal for read I/Os, very good for scans, **poor for writes**
  - **Really poor for bursts of small updates** (read-modify-write)
- *KV-stores:* write-optimized schemes to cope with ingest
  - Idea: Buffer writes + use single I/O to write (LSM-Tree, B^{ε}-Tree)
  - **Reorganize** data over time to improve reads + updates
  - Leads to **multi-stage organization** for data



Multistage stores: constant re-organization



- They organize data in N levels (L_0 in memory) – sizes grow by f (e.g. f=10)
  - Data (keys) in each level are unique and sorted (to improve reads/scans)
- Reorganizing data from level to level (**compaction**)
  - Reason: 1. Reclaim old values (garbage), 2. Keep data sorted for reads/scans
  - Various techniques: leveling, tiering, incremental, variations
- All approaches use some sort of sorted string tables (SSTs): container of data on device
- Compaction: Merge-sort operations for SSTs = read–sort–write
  - + Generates only large I/Os
  - - Increases I/O traffic compared to application I/O (amplification)
  - - Incurs CPU overhead, data traversals poor match to tech trends

Why are KV stores popular?

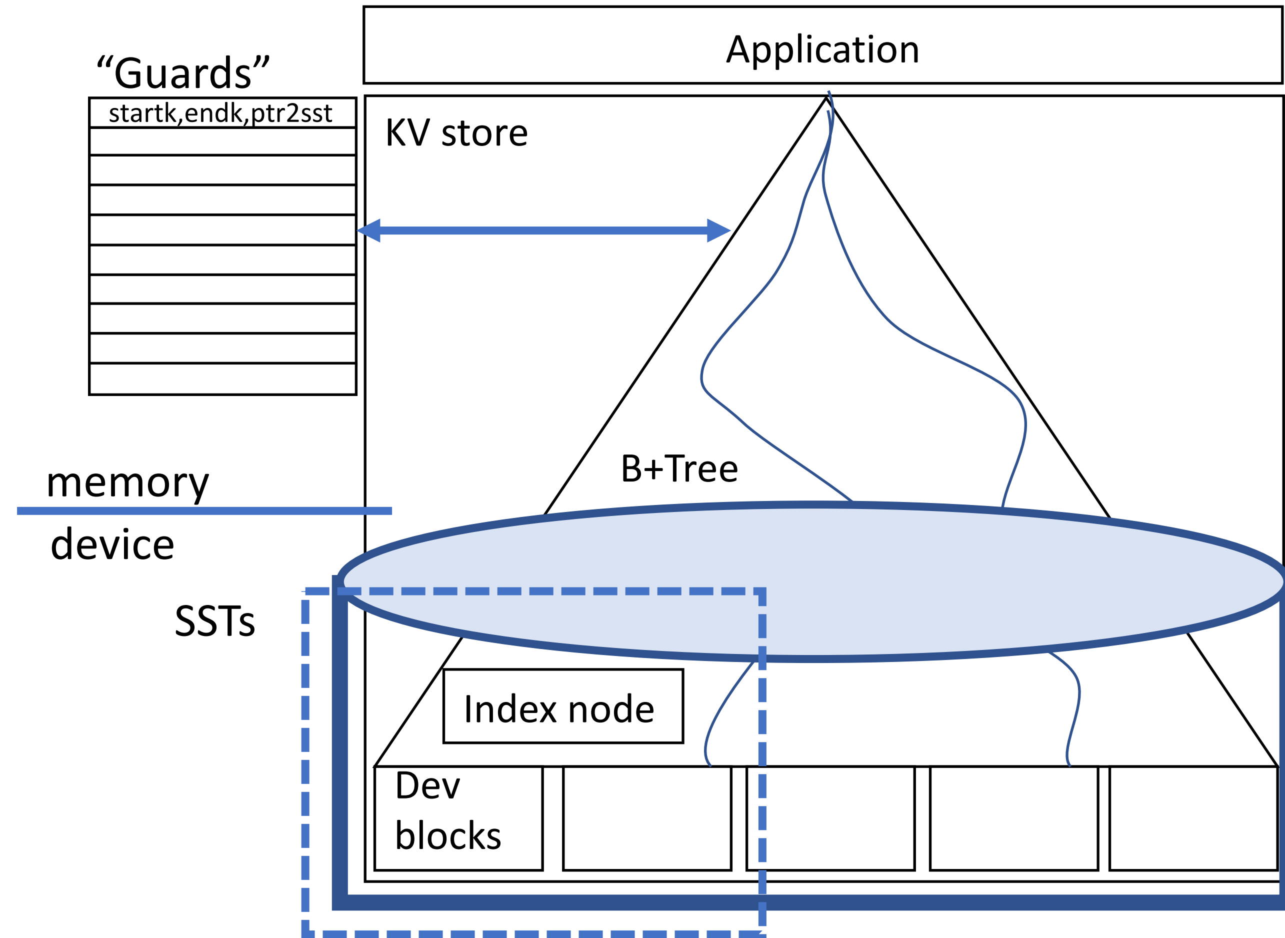
1. High device **efficiency for small writes**
   - No read-modify-write operations, only large writes at device level
   - Particularly important (cloud apps)
2. Ingest path **insensitive to mixed workloads**
   - It makes application-level optimizations irrelevant
   - Particularly important (cloud colocation/resource sharing)
3. Convenient **abstraction**
   - KV pairs vs. offsets - Further decouple data from location on device
   - Appears to be important as well

Overview

- What are key-value (KV) stores - Why?
- Our KV work at FORTH and the Univ. of Crete
  - Indexing – towards small SSTs [Tucana ATC'16, Kreon SoCC'18]
  - KV separation – differentiated treatment [Parallax SoCC'21]
  - Replication – shipping indexes [Tebis EuroSys'22]
  - Memory-mapped I/O caches (mmio) [FastMap ATC'20, Aquila EuroSys'21]
- Concluding remarks - What is the ideal KV store?

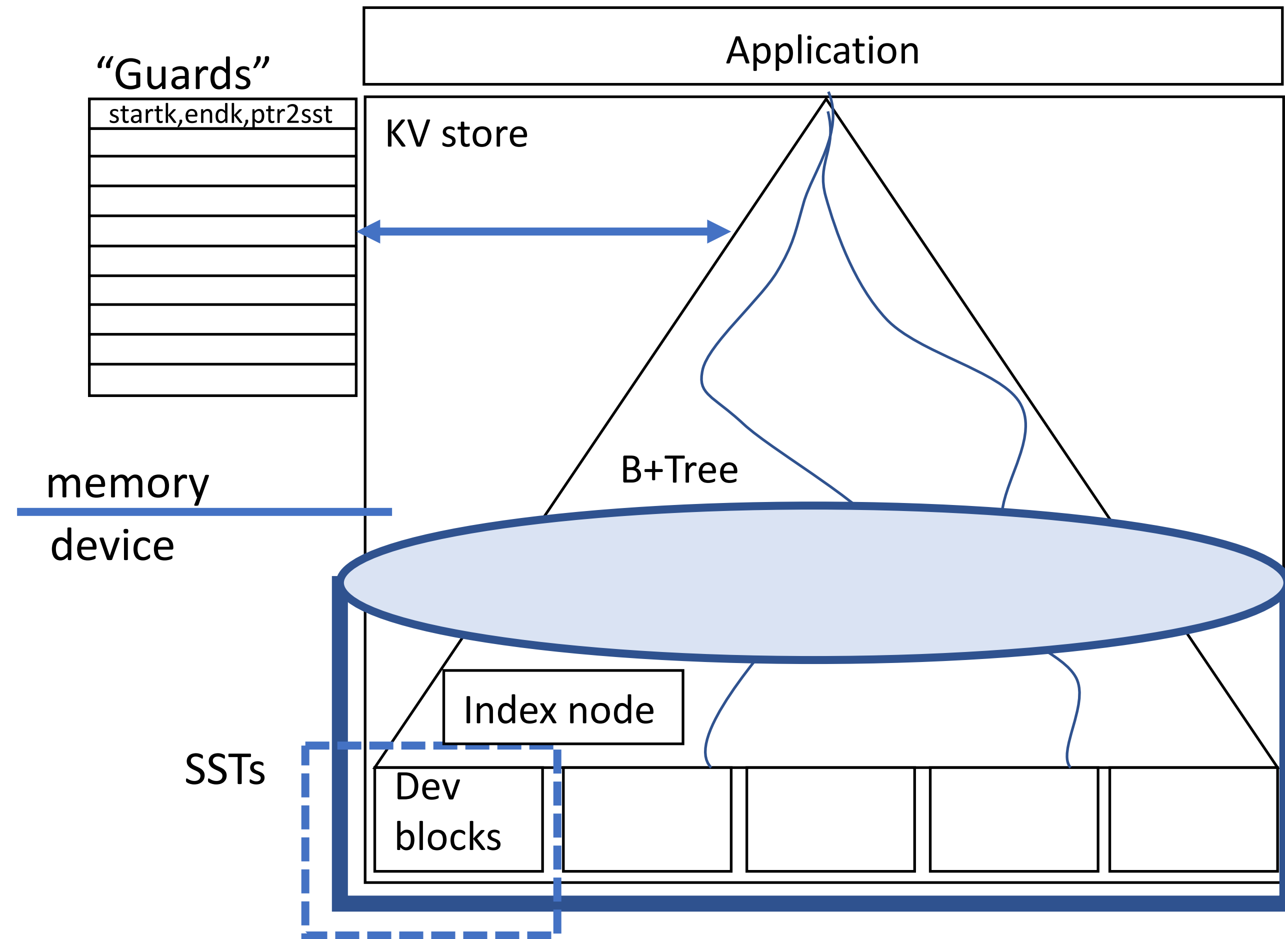
Typical KV index - large SSTs

- Index points to dev/data blocks (4K)
- Upper part of index can be a table
  - One entry per SST
  - Few SSTs even for TBs of storage
  - Always in memory
- Lower part of index packed with data blocks in each SST
- Large SSTs, e.g. 64 MBs – Large I/Os
- Compaction: heavy merge-sort ops



Towards small SSTs

- + Avoid CPU cycles for heavy merge-sort
  - Good for CPU trends
  - **Take advantage of key skew**
  - **Benefits for tail latency as well (current work)**
- - Increase I/O randomness & small I/Os
  - ok with device trends
- Requires
  - More guards → may not fit in memory
  - Guard "table" needs dynamic organization (avoid constant copies)
  - SST structure (and contents) need to change
- We explored two approaches
  - Tucana – B^{ε}-Tree index
  - Kreon – B+-Tree index

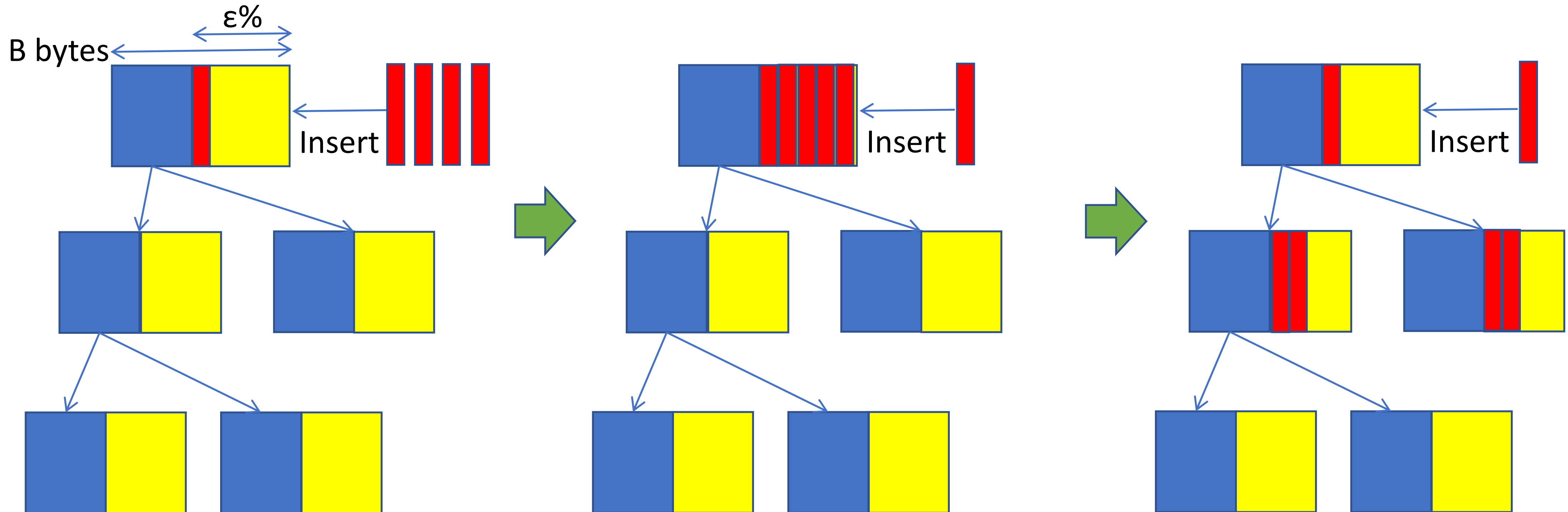


B^{ε}-Trees

- B^{ε} is a B-Tree variant that uses buffering for inserts
- Similar complexity as B-Tree for point, range queries
- No compactions – small, unsorted buffers in index nodes
  - Better CPU overhead and I/O amplification
  - Worse I/O randomness and I/O size

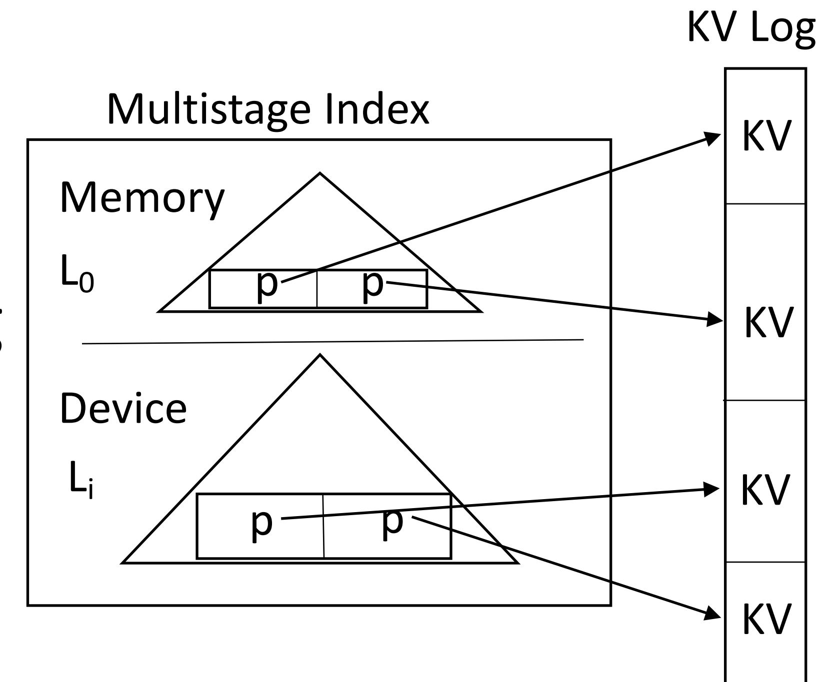
B^{ε}-Trees

- Each **internal** node has a persistent buffer
  - ε% of node used for data (ε ∈ [0,1])
- Buffers "log" multiple inserts and amortize I/O to device



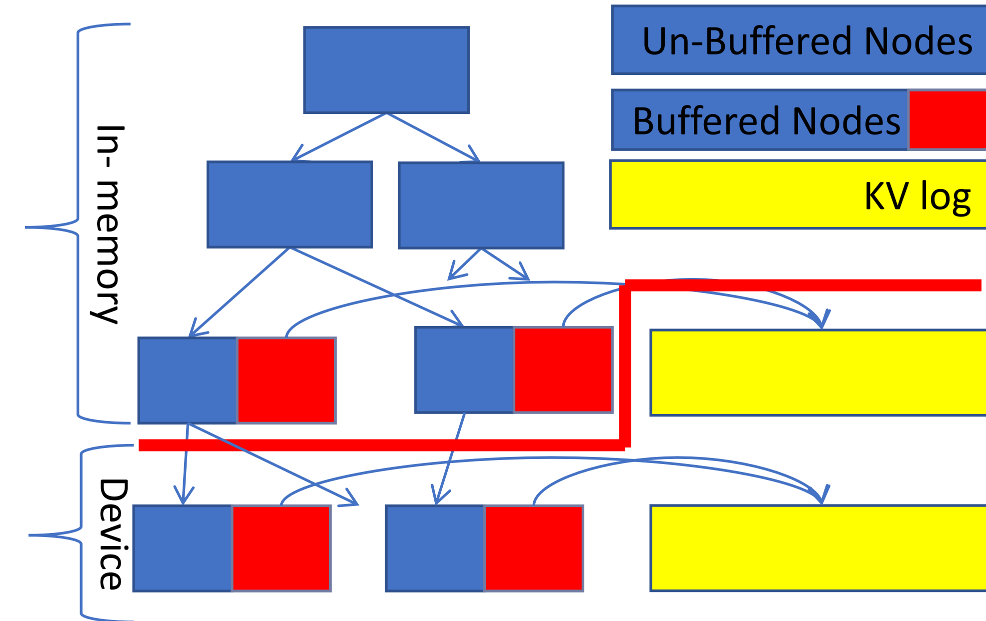
KV Separation [Atlas’15, WiscKey‘16, Tucana’16]

- Introduces randomness to reduce I/O amplification
- Places KV pairs in log (value log)
  - Contains full KV pairs (key + value)
- Maintains multistage index only for keys
  - Internal nodes contain only keys or pivots
  - Index leaves contain only keys and pointers to log
- Reduces write amplification significantly
- **But** requires cleaning the value log



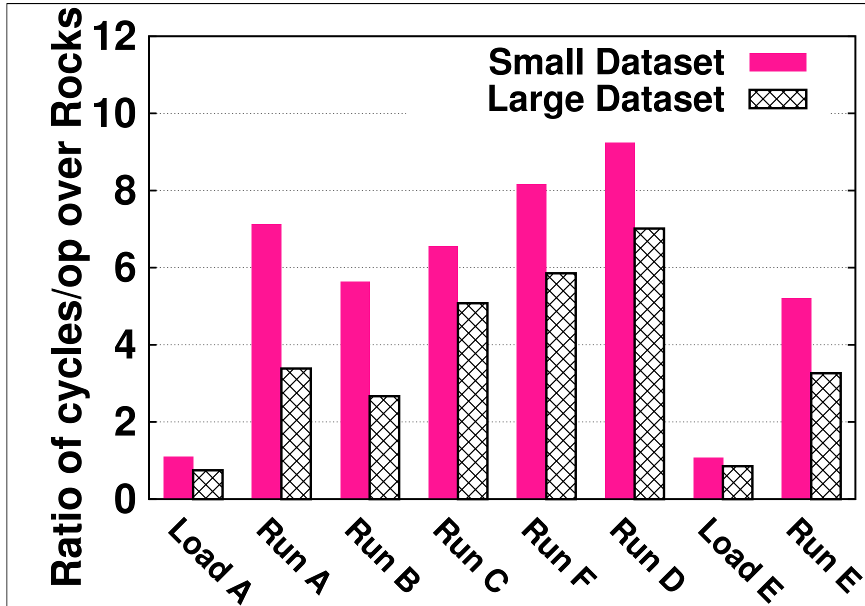
Tucana B^{ε}-Tree index

- Buffered tree + small, unsorted SSTs + KV separation
  - Full index - partly in memory
  - Buffers only in leaf nodes in memory
  - KVs only in log - index uses pointers to data
- "Spills" of data (instead of compactions)
  - Propagate updates to lower levels
  - Less reorganization of buffers - more 4K I/Os
- Searching requires key accesses on device
  - Tucana uses two optimizations for buffered nodes
  - Key prefixes (fixed size) → 65%-75% fewer I/Os for keys in all queries
  - Hashes for full keys (fixed size) → 98% fewer I/Os for keys in point queries



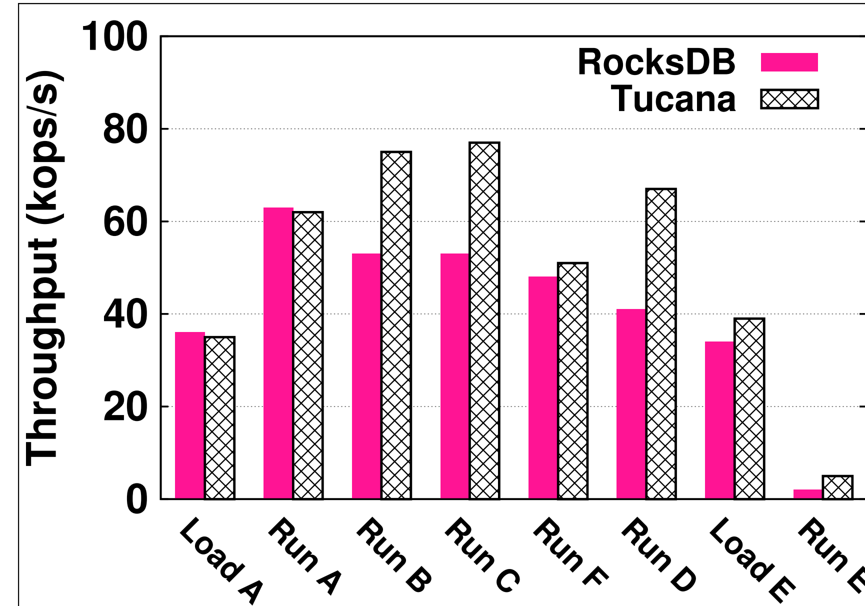
Efficiency

cycles/op



- Small dataset: 5.2x up to 9.2x
- Large Dataset: 2.6x up to 7x

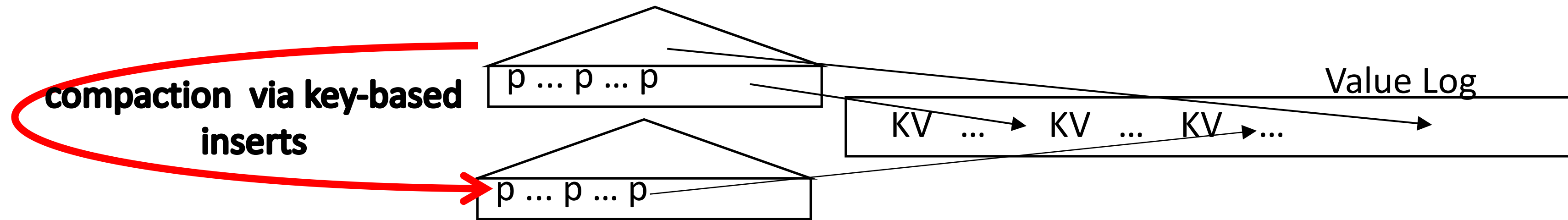
ops/s



- Large dataset: 1.1x up to 2x
- Device is the bottleneck

- Tradeoff: Amplification vs. Randomness (Writes)
  - Tucana has 3.5x less I/O traffic but 49x #I/Os (smaller, random)
  - For two SSD generations Tucana: 4.7x and 3.1x throughput over RocksDB

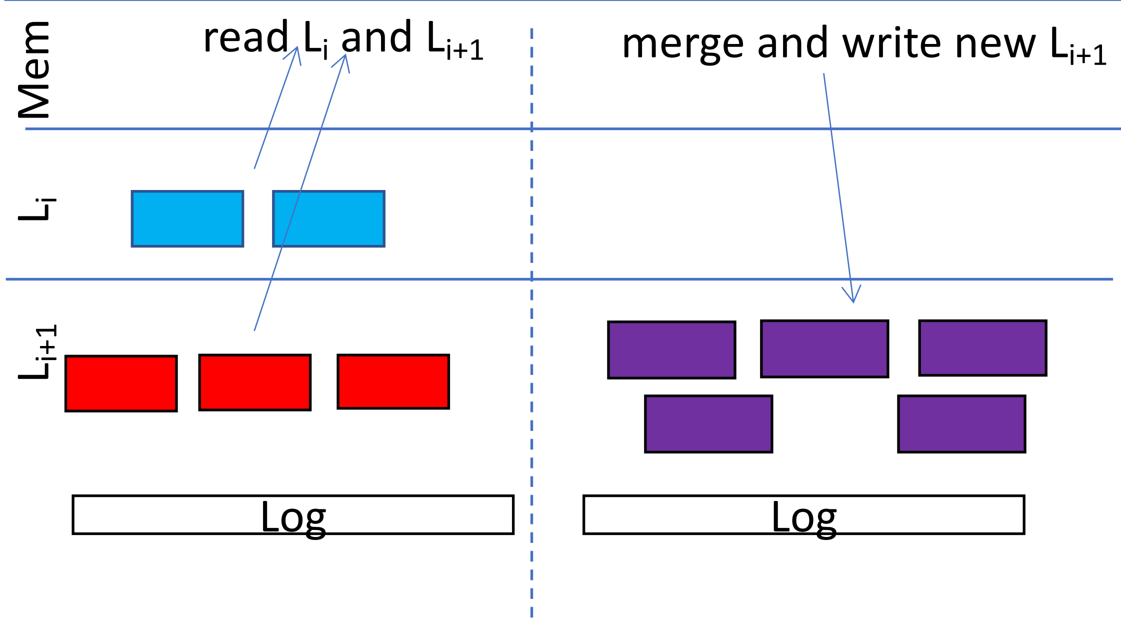
Kreon: LSM + KV log + Fine-grain indexing [ACM SoCC'18]

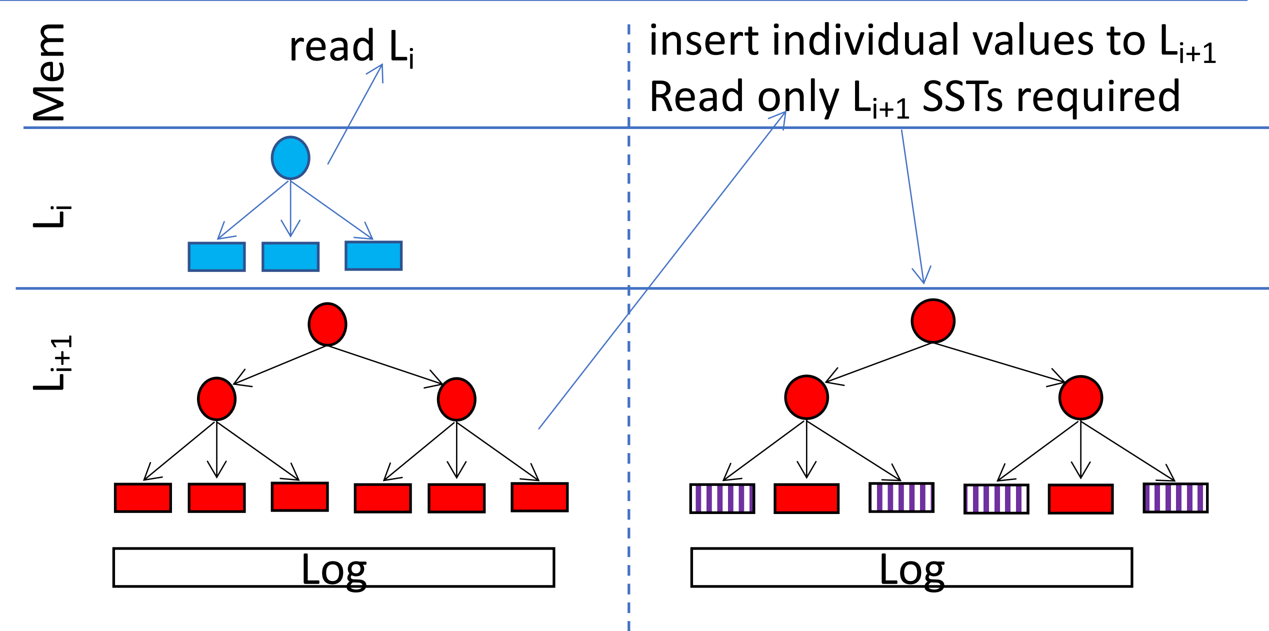


- LSM-type multilevel structure (memory-mapped)
  - Separate B+-tree index per level with small leaves/SSTs (sorted/unsorted) [Similar to bLSM'12]
  - Place KV pairs in a value log and reorganizes only pointers [Atlas'15, WiscKey'16, Tucana'16]
- Efficient merging via spills of small SSTs
  - Can take advantage of key skew to read less data from of L_{i+1} compared to large SSTs
- Kreon uses partial/adaptive data reorganization
  - In each level, reorganize data in each SST so data are sorted
  - … and a few (logically) neighboring SSTs
  - Rest of SSTs can be located anywhere on the device
  - Use average scan size to determine threshold for how much to reorganize

LSM compaction

Kreon spill





- KV separation and fine-grain indexing reduce I/O amplification (10x)
  - Depends on KV pair size – great for larger (e.g. 1 KB) KV pairs
  - Not great for scans, but rely on fast storage devices, e.g. NVMe
- Learned that
  - We touch less data but with more operations – Fine-grain index updates cost in CPU
  - Log cleaning can be very expensive for small KVs
    - 60% of KV pairs observed in production workloads (Meta/FB) are small (33-100 Bytes) [Cao et al. FAST'20]

Overview

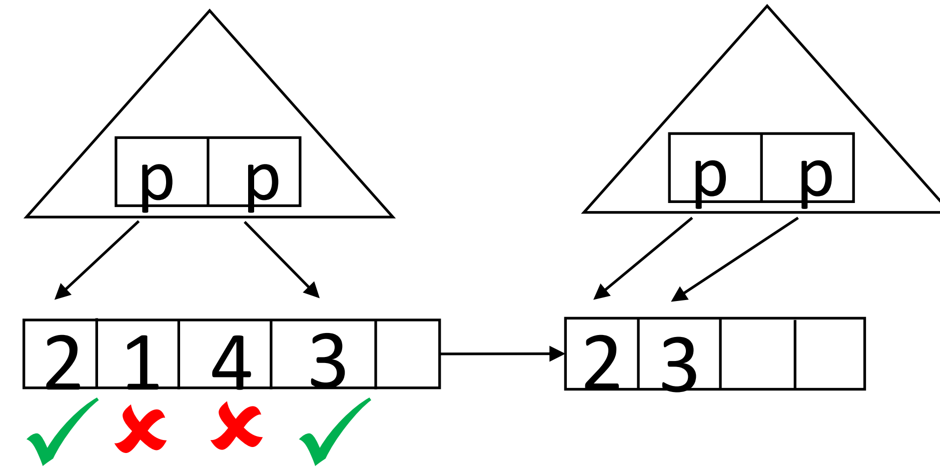
- What are key-value (KV) stores - Why?
- Our KV work at FORTH and the Univ. of Crete
  - Indexing – towards small SSTs [Tucana ATC'16, Kreon SoCC'18]
  - KV separation – differentiated treatment [Parallax SoCC'21]
  - Replication – shipping indexes [Tebis EuroSys'22]
  - Memory-mapped I/O caches (mmio) [FastMap ATC'20, Aquila EuroSys'21]
- Concluding remarks - What is the ideal KV store?

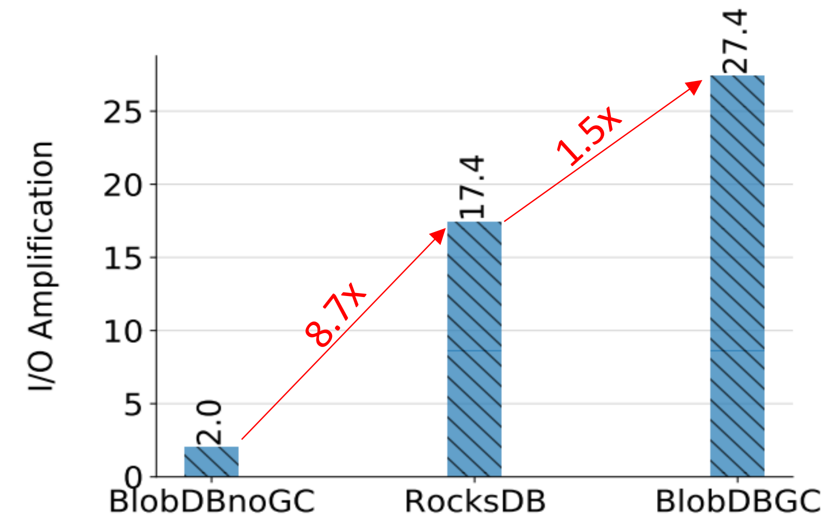
Value log garbage collection

- Deletes, updates result in stale values
  - Need to garbage collect
  - Otherwise run out of space quickly
- Main overheads for GC to free a log segment

1. Identify valid KV pairs → requires lookup in the index for each key in the segment
2. Re-append valid KV pairs in log → requires updating the index

- GC impact on I/O amplification
  - GC cost large for small (~1 to 100 bytes) KV pairs

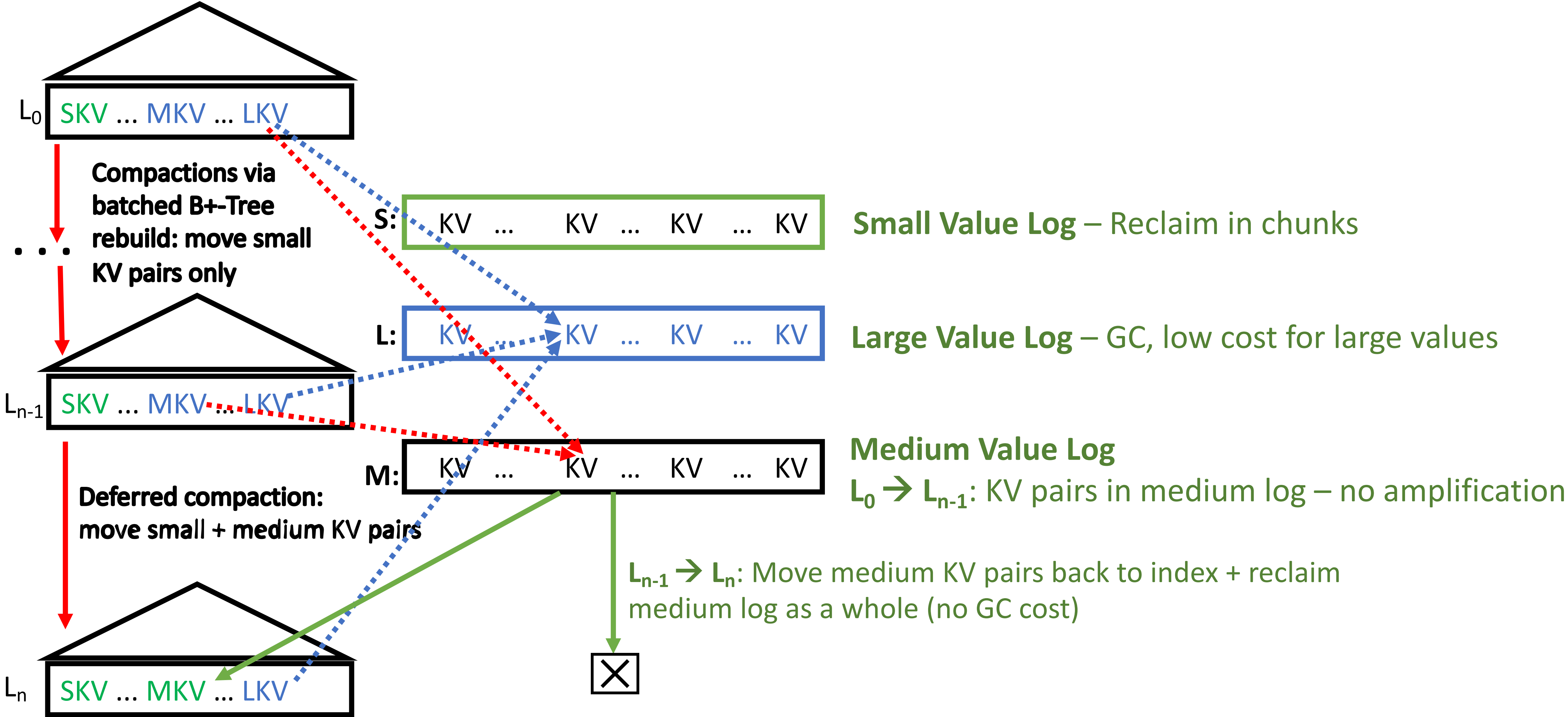




Parallax: Hybrid KV Placement [ACM SoCC'21]

- Idea: Do not treat all KV pairs in the same manner
- Categorize KV pairs based on size
- Small pairs (1–100B): Always in-place
  - No benefits from KV separation - Avoid garbage collection completely
- Large KV pairs (>1000B): Always in a separate (large) log
  - GC not as expensive - Use garbage collection to reclaim space
  - Up to 10x less amplification compared to compactions
- Medium KV pairs (100-1000B): Deferred Compaction
  - Store a pointer in the LSM index for first N-1 levels
  - Defer placement in medium log during L_0 -> L_1 compaction
  - Keep medium KV pairs in **sorted** runs for efficient merge (in-place) at L_i

Parallax: Hybrid KV Placement [ACM SoCC'21]

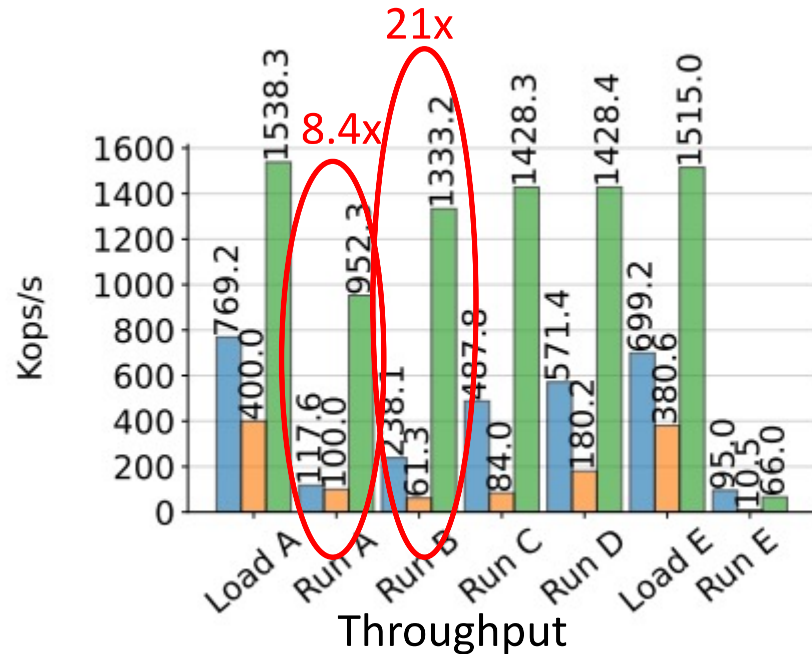


Recovery in Parallax

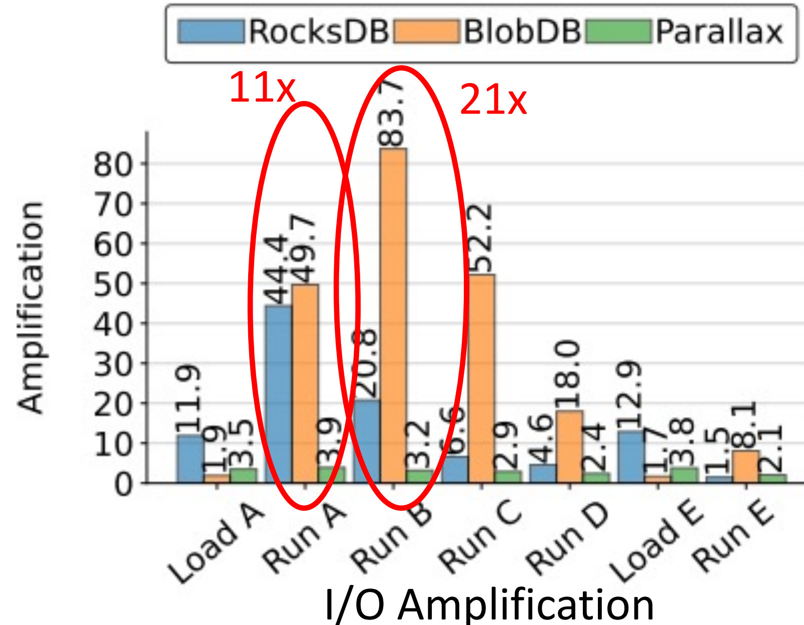
- Using multiple logs breaks ordering of operations
  - Use sequence numbers across logs
  - Increasing monotonic counter stored with each KV pair
- Recovery logs
  - Use large log for recovery directly for large KV pairs
  - Use L_0 recovery log for small and medium KV pairs
  - Medium KVs are moved to medium log during $L_0 \rightarrow L_1$ compaction
- KV pairs may change size and category over time due to updates

Parallax Results

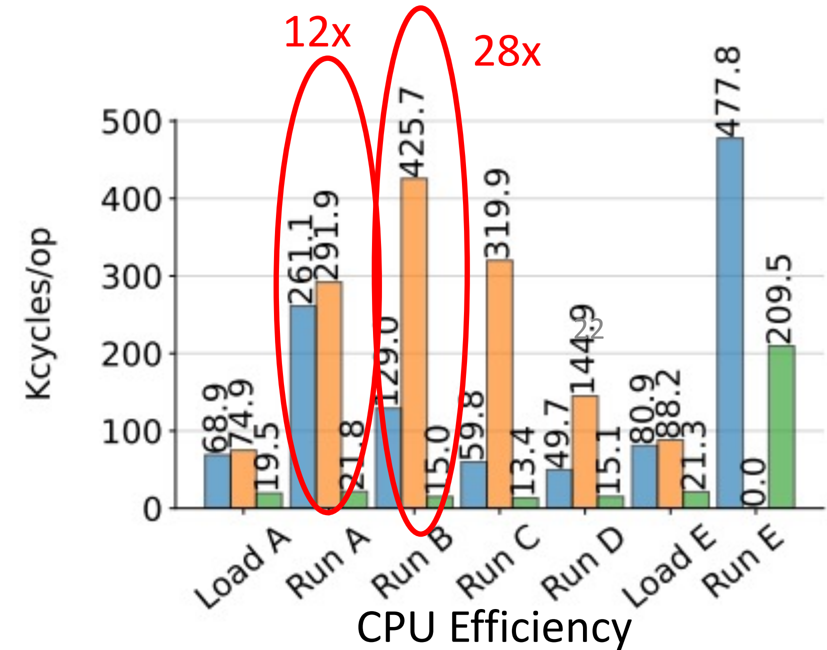
- Important to examine workloads with mixed KV sizes
  - Workloads from Facebook distribution [FAST'20]
- Examine I/O amplification, I/O throughput, CPU efficiency (cycles/operation)
- Significant improvements + Parallax makes KV separation more practical



Throughput



I/O Amplification



CPU Efficiency

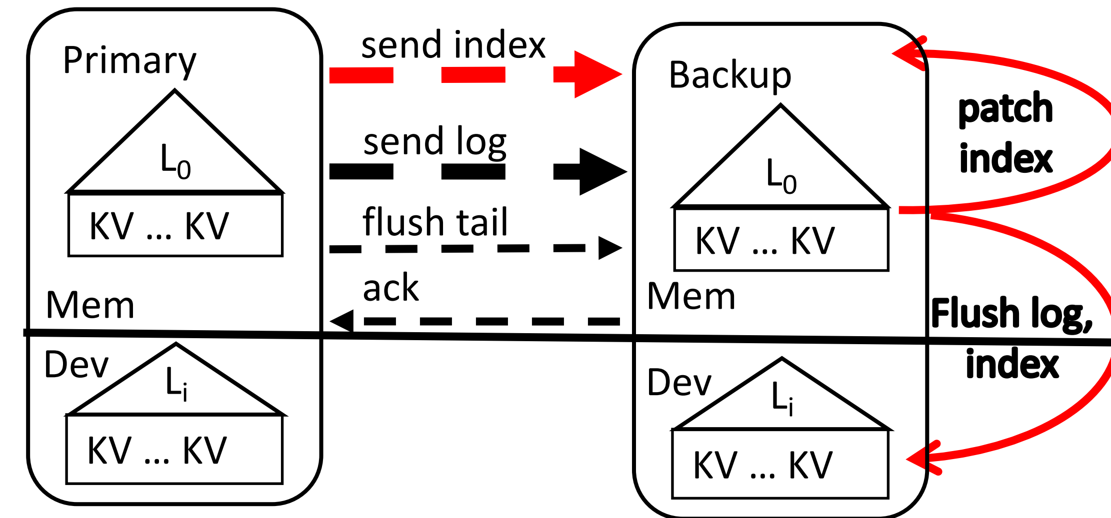
- **Current prototype: LSM + B+-Tree index + Small SSTs & batched rebuild + Hybrid KV placement**

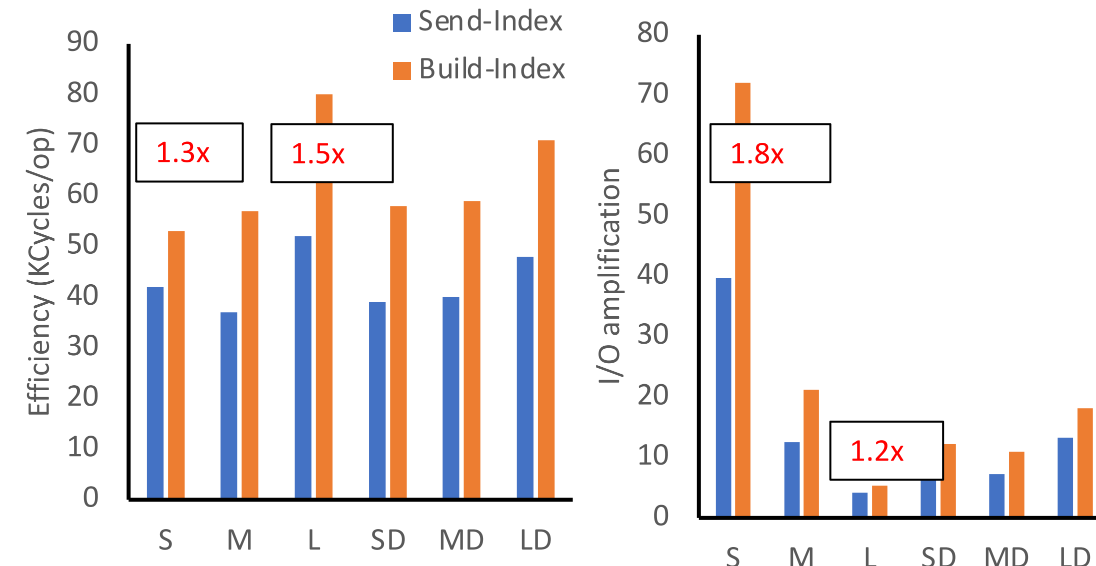
Overview

- What are key-value (KV) stores - Why?
- Our KV work at FORTH and the Univ. of Crete
  - Indexing – towards small SSTs [Tucana ATC'16, Kreon SoCC'18]
  - KV separation – differentiated treatment [Parallax SoCC'21]
  - Replication – shipping indexes [Tebis EuroSys'22]
  - Memory-mapped I/O caches (mmio) [FastMap ATC'20, Aquila EuroSys'21]
- Concluding remarks - What is the ideal KV store?

Tebis: Replication via Index Shipping [EuroSys'22]

- KV stores replicate data and index
  - Typically: Send data to all backups
  - Requires compaction in each backup
  - Reduces network traffic
- Tebis: Send pre-built index to replicas
  - Build index once in primary & send to all
  - Requires translating device pointers in index
  - Saves (in backup nodes)
    - CPU → no merge sort
    - Device I/O → no read for L_i (trade for network)
    - Memory → no L_0 in backups
  - Better for both large and small KV pairs





Overview

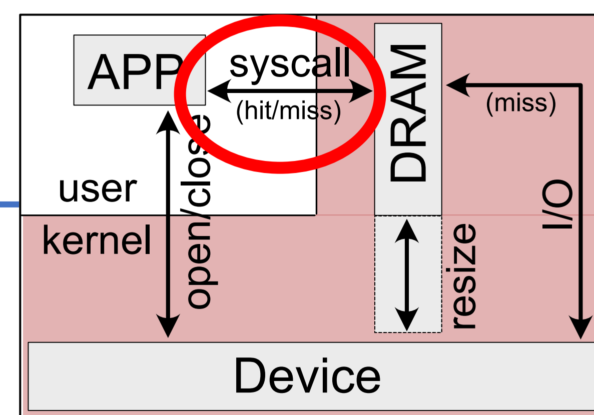
- What are key-value (KV) stores - Why?
- Our KV work at FORTH and the Univ. of Crete
  - Indexing – towards small SSTs [Tucana ATC’16, Kreon SoCC’18]
  - KV separation – differentiated treatment [Parallax SoCC’21]
  - Replication – shipping indexes [Tebis EuroSys’22]
  - Memory-mapped I/O caches (mmio) [FastMap ATC’20, Aquila EuroSys’21]
- Concluding remarks - What is the ideal KV store?

Device I/O: syscalls vs. mmio

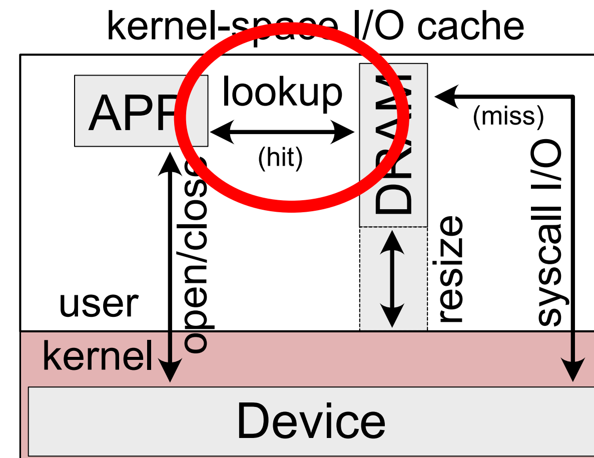
- Explicit I/O = read/write system calls
  - Can be buffered or unbuffered (direct I/O)
  - SPDK avoids system calls but dedicates devices to a single application
- Memory-mapped I/O → mmap in Linux
  - A file mapped to the virtual address space
  - Use load/store instructions to access data
  - Kernel fetches/evicts pages on-demand
  - Always buffered - common buffers between application and page cache
- You can mix explicit and memory-mapped I/O
  - Semantics may not be trivial

I/O Caching

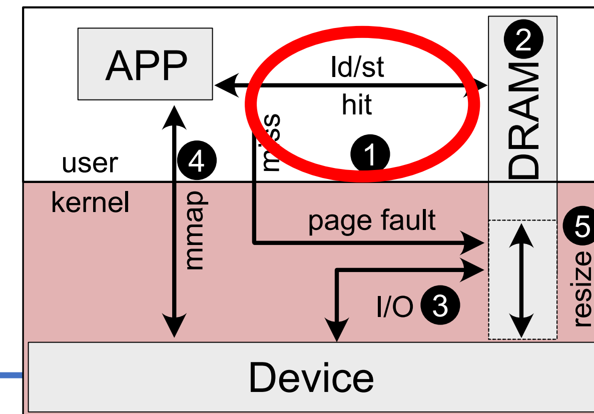
- Caching essential in I/O path - Reduce accesses to the device
  - All KV stores use a cache (typically read+block cache)
- Norm is explicit I/O
  - (1) Kernel-space cache: Requires system calls also for hits
  - (2) User-space cache: Function call for hits + system calls for misses
  - Even a user-space cache incurs high CPU overhead
  - Also, memory use is less flexible (no sharing with others)
- Memory-Mapped I/O
  - Eliminates cost for lookups: virtual memory mappings (MMU+TLB)
  - Requires no copy between kernel – user space
  - Eliminates serialization/deserialization of data



kernel-space I/O cache



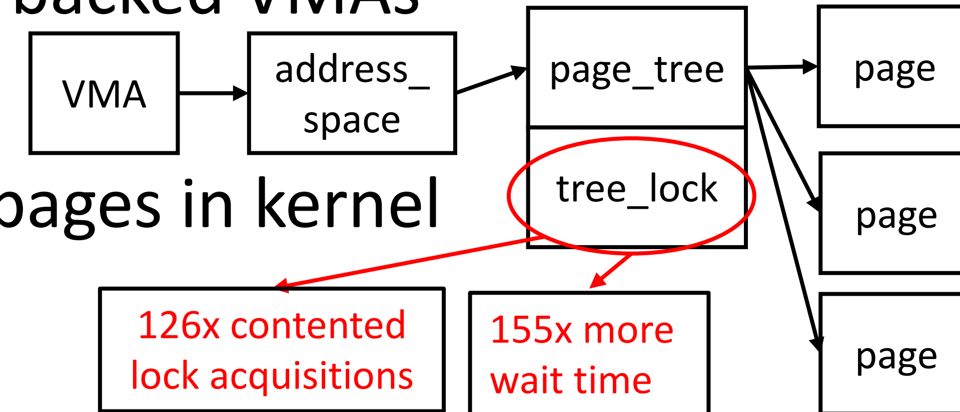
user-space I/O cache



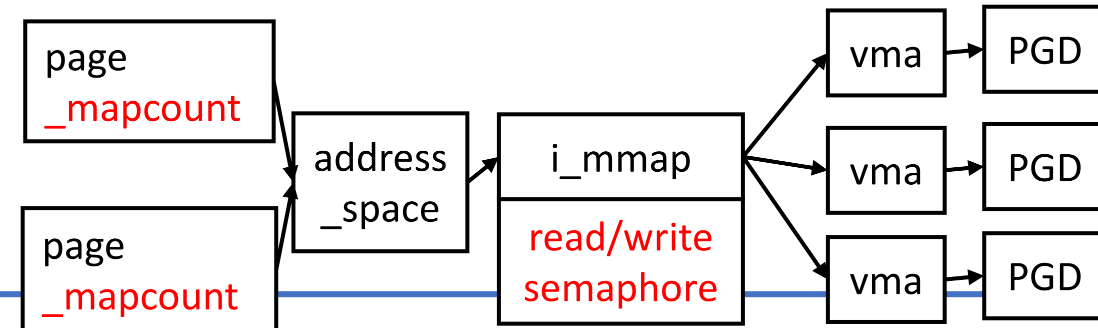
Linux mmap I/O cache

FastMap: mmio scaling [Usenix ATC'20]

- FastMap is a custom mmap path in Linux for file-backed VMAs
- Addresses mmap scalability issues
- (1) Fastmap separates metadata for clean, dirty pages in kernel
  - Linux clean + dirty page structure uses tree_lock
  - Fastmap avoids all centralization points

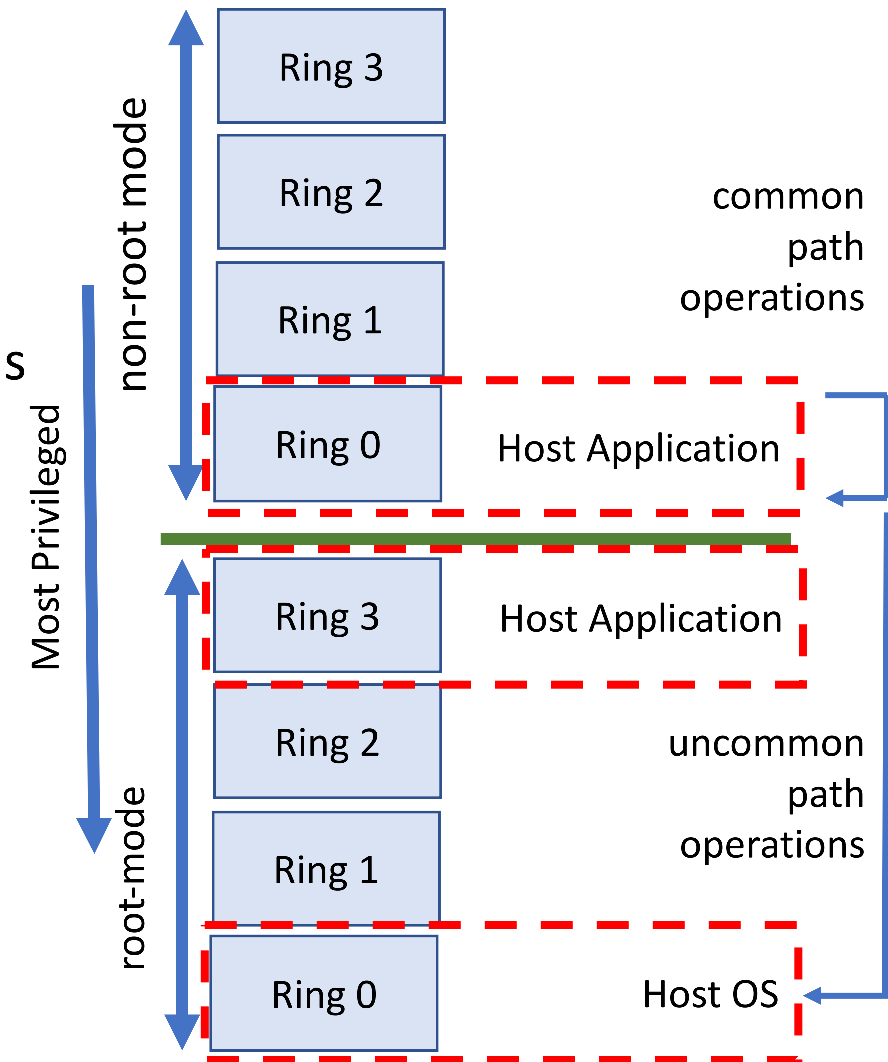


- (2) Fastmap optimizes (replaces) Linux reverse mappings
  - Linux uses object-based reverse mappings for sharing pages among many processes
  - Results in high cross NUMA-node traffic & CPU cycles
  - Fastmap uses full-reverse mappings to reduce CPU cost
- (3) Fastmap uses a custom I/O cache
  - Reduce interference and latency variability



Aquila: mmio miss overhead [EuroSys'21]

- Reduces mmap page fault overheads
  - Today: Apps run in ring 3, page faults handled in ring 0
  - Requires frequent crossings/traps to kernel
- Aquila stipulates that for file-backed mappings
  - System needs protection only for (large) memory allocations
  - Page faults and device access can happen in a libOS
- Aquila re-designs operations of page fault path
  - Uncommon: placed in **root ring 0** for protection
  - Common: placed in **non-root ring 0** for performance
- As a result
  - Application incurs low cost for page faults & dev access
  - Protection still available, at low cost (uncommon path)



Memory-mapped I/O path scalability and overhead

microbenchmark
device: null_blk
dataset: 4TB
DRAM cache: 192GB

3x in reads

6x in writes

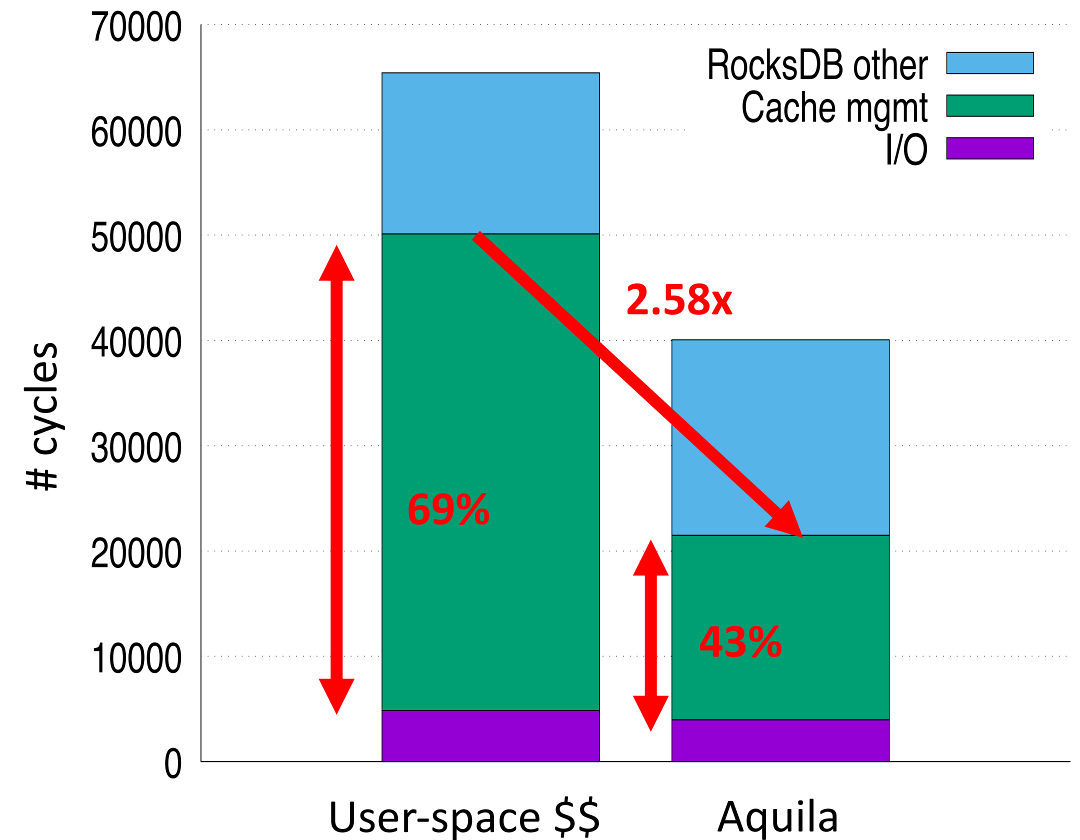
million page-faults/s (IOPS)

5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0

1 2 4 8 16 32

Linux-Read Linux-Write FastMap-Read FastMap-Write

#threads



xmap – Current Work

- Introduce huge pages for file-backed mmap
  - Linux THP only for anonymous mappings
- Supports both transparent and guided use of regular/huge pages
  - Dynamic and flexible use of both page types
- **Asynchronous** page promotions and demotions
- High degree of (explicit and implicit) control
  - Multiple page pools, VMA directives, policies

Limited use for mmio today

- Traditionally used for mapping executables and shared libraries (and some hand-tuned systems)
- Scalability: Pages faults with #threads
  - **We thought:** The kernel usually behaves well with a large number of threads → Not really
- Misses & I/O granularity: 4KB pages result in small and random I/Os
  - **We thought:** Page faults for I/Os do not cost much with modern devices/CPUs → Not really
  - +Synchronous behavior (I/O + PTE + TLB setup)
- Lack of control in I/O cache: Cache behavior & pollution
  - **We thought:** LRU (and variants) work → Not really, direct I/O is a useful tool
- Complexity in recovery: Persistent data selection & ordering
  - **We thought:** You can avoid the 2x writes of journaling without cost → Not really
  - mmio mixes persistent & non-persistent data (expensive, loss of ordering across threads)

Overview

- What are key-value (KV) stores - Why?
- Our KV work at FORTH and the Univ. of Crete
  - Indexing – towards small SSTs [Tucana ATC’16, Kreon SoCC’18]
  - KV separation – differentiated treatment [Parallax SoCC’21]
  - Replication – shipping indexes [Tebis EuroSys’22]
  - Memory-mapped I/O caches (mmio) [FastMap ATC’20, Aquila EuroSys’21]
- Concluding remarks - What is the ideal KV store?

Concluding Remark: Towards a “perfect” KV store?

- Application writes → **At device speed**
  - Match available device throughput
  - Amortize, compaction I/O amplification ~1x
  - **Hybrid KV placement, small SSTs (key skew)**
- App reads (and small scans) → **1x rnd I/O**
  - (for existing keys, no I/Os for reads of non-existing keys)
  - Match available device IOPS
  - **Place all metadata in memory**
- Tail latency → **From secs to ~100 μs**
  - 1-2 IOs/level **x** #levels
  - **Small SSTs and bounded overlap across levels**
- Small working sets → **no IO**
  - Match mem xput – role of the cache
  - **Caching in KVs: Non-trivial in several respects**
    - Variable vs. fixed size items, data vs. metadata, hot/cold behavior, reads vs. writes
- Scale-out → **Load-balancing, parallelism**
  - Better use of available resources for scaling
  - **Horizontal disaggregation of KV functions**
- Device heterogen. → **DRAM–NVM–FLASH**
  - **(Unclear !)** So far, NVM used only for special purpose uses, e.g. metadata

Quite a bit of work remains to be done !

Work By

- Code repository: https://github.com/CARV-ICS-FORTH
- People
  - Georgios Saloustros (KV lead)
  - Pilar Gonzalez-Ferez (KV)
  - Manolis Marazakis (mmio)
  - Panagiota Fatourou (KV VAT model)
  - Anastasios Papagiannis, PhD (FastMap, Aquila, Tucana, Kreon)
  - Georgios Xanthakis, PhD (Parallax)
  - Nikolaos Batsaras, MSc (VAT model, Parallax)
  - Michail Vardoulakis, MSc (Tebis)
  - Ioannis Malliotakis, MSc (xmap, heap extension)
  - Georgios Stylianakis, MSc (Parallax, Tebis)
  - …and discussions with many others in FORTH

- Main funding support
  - European Commission H2020 Programme: EVOLVE (GA 825061), ExaNeSt (GA 671553)
  - European High-Performance Computing Joint Undertaking (JU): EUPEX (GA 101033975)
  - Meta through Graduate Fellowships

Further details

- Tebis: index shipping for efficient replication in LSM key-value stores. Michalis Vardoulakis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. In Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys'22). Association for Computing Machinery, New York, NY, USA, 85–98. https://doi.org/10.1145/3492321.3519572
- Parallax: Hybrid Key-Value Placement in LSM-based Key-Value Stores. Georgios Xanthakis, Georgios Saloustros, Nikolaos Batsaras, Anastasios Papagiannis, and Angelos Bilas. In proceedings of ACM Symposium on Cloud Computing 2021 (ACM SoCC'2021). Nov 1-3, 2021 Seattle, Washington, USA (hybrid event). https://doi.org/10.1145/3472883.3487012
- (Aquila) Memory-mapped I/O on steroids. Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. In Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys'21). Association for Computing Machinery, New York, NY, USA, 277–293. https://doi.org/10.1145/3447786.3456242
- (FastMap) Optimizing memory-mapped I/O for fast storage devices. Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. In Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20). USENIX Association, USA, Article 56, 813–827. https://www.usenix.org/system/files/atc20-papagiannis.pdf
- VAT: Asymptotic Cost Analysis for Multi-Level Key-Value Stores. Nikos Batsaras, Giorgos Saloustros, Anastasios Papagiannis, Panagiota Fatourou, and Angelos Bilas. February 2020. arXIV 2003.00103. arxiv.org/abs/2003.00103v1
- (Kreon) An Efficient Memory-Mapped Key-Value Store for Flash Storage. In Proceedings of the ACM Symposium on Cloud Computing (SoCC '18). Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Association for Computing Machinery, New York, NY, USA, 490–502. https://doi.org/10.1145/3267809.3267824
- Tucana: design and implementation of a fast and efficient scale-up key-value store. Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. In Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16). USENIX Association, USA, 537–550. https://www.usenix.org/system/files/conference/atc16/atc16_paper-papagiannis.pdf

Thank you!
Questions?

Angelos Bilas (presenting the work of many others)

bilas@ics.forth.gr

FORTH

INSTITUTE OF COMPUTER SCIENCE

Department of Computer Science

University of Crete

Institute of Computer Science (ICS)

Foundation for Research and Technology – Hellas (FORTH)