Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value store

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

FORTH-ICS
Institute of Computer Science

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

Key-value Stores – Important Building Block

- Key-value store: A dictionary for **arbitrary** key-value pairs
  - Used extensively: web indexing, social networks, data analytics
  - Supports inserts, deletes, point (lookup) and range queries (scan)
- Today, key-value stores **inefficient**
  - Consume a lot of CPU cycles
  - Mostly optimized for HDDs – right decision until today

Challenges

- Overhead is related to several aspects of key-value stores
  - Indexing data structure
  - DRAM caching and I/O to devices
  - Persistence and failure atomicity
- Our goal: improve CPU efficiency of key-value stores
  - Design for fast storage devices (SSDs)
  - Bottleneck shifts from device performance to CPU overhead

Outline of this talk

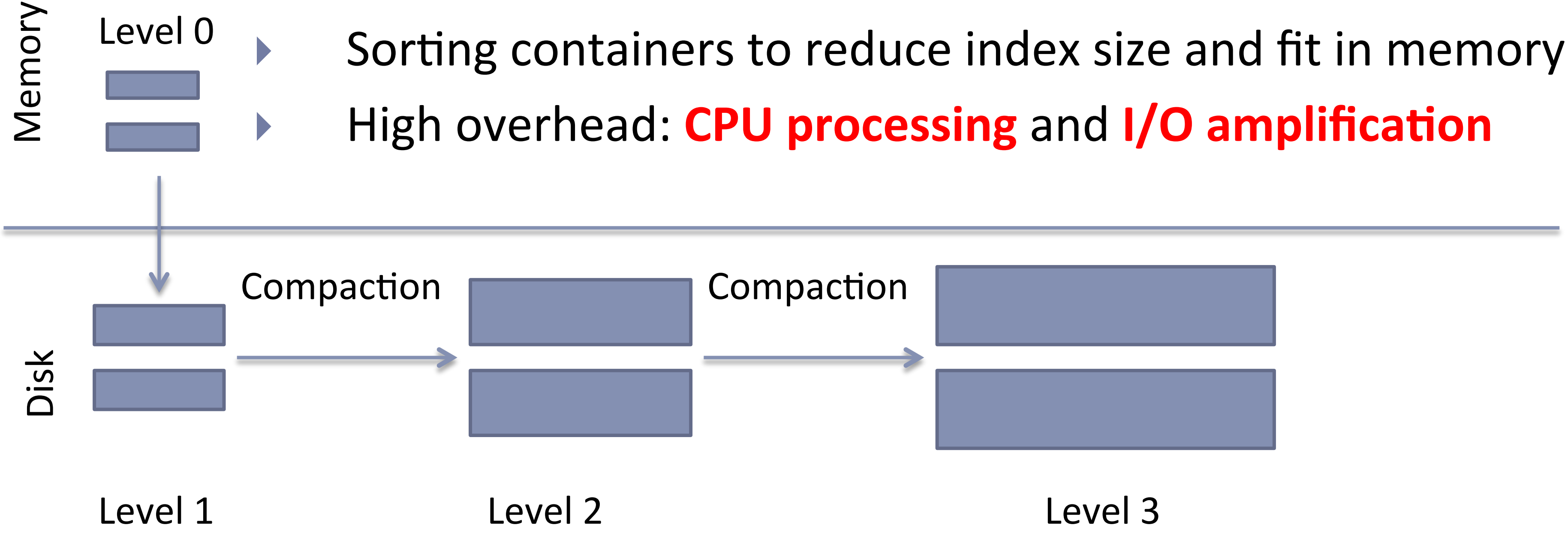
- **Discuss our design and motivate decisions**
  - Indexing data structure
  - DRAM caching and I/O to devices
  - Persistence and failure atomicity
  - H-Tucana: An HBase Integration
- Evaluation
- Conclusions

Write Optimized Data Structures (WODS)

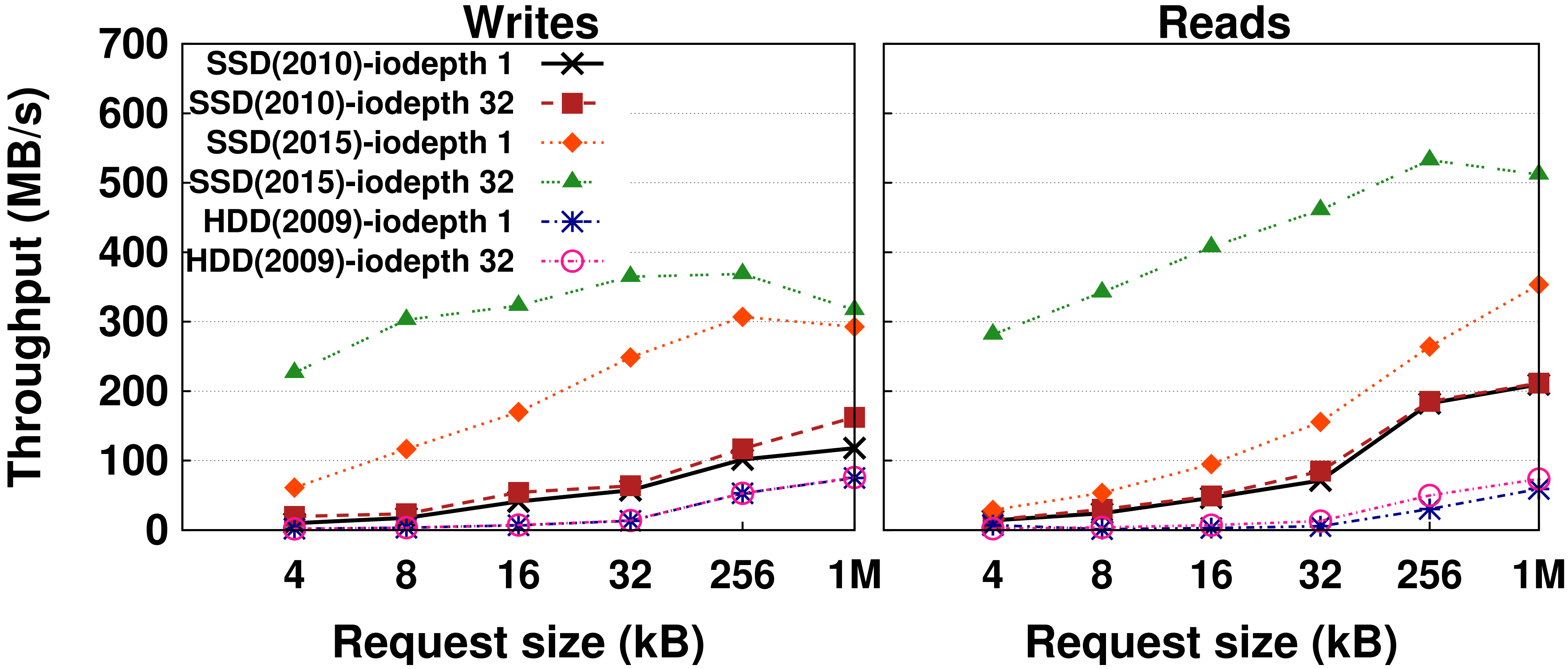
- Inserts are important for key-value stores
- Need to avoid a single I/O per insert
- Main approach: Buffer writes in some manner
  - … and use single I/O to the device for multiple inserts
  - Examples: LSM-Trees, B^{ε}-Trees, Fractal Trees
- Most popular: LSM-Trees
  - Used by most key-value stores today
  - Great for HDDs - always perform large sequential I/Os

LSM-Trees

- Data in large containers - leads to large/sequential I/O
- **Great for HDDs!** However, they require **compactions**
- Sorting containers to reduce index size and fit in memory
- High overhead: **CPU processing** and **I/O amplification**



SSDs vs. HDDs



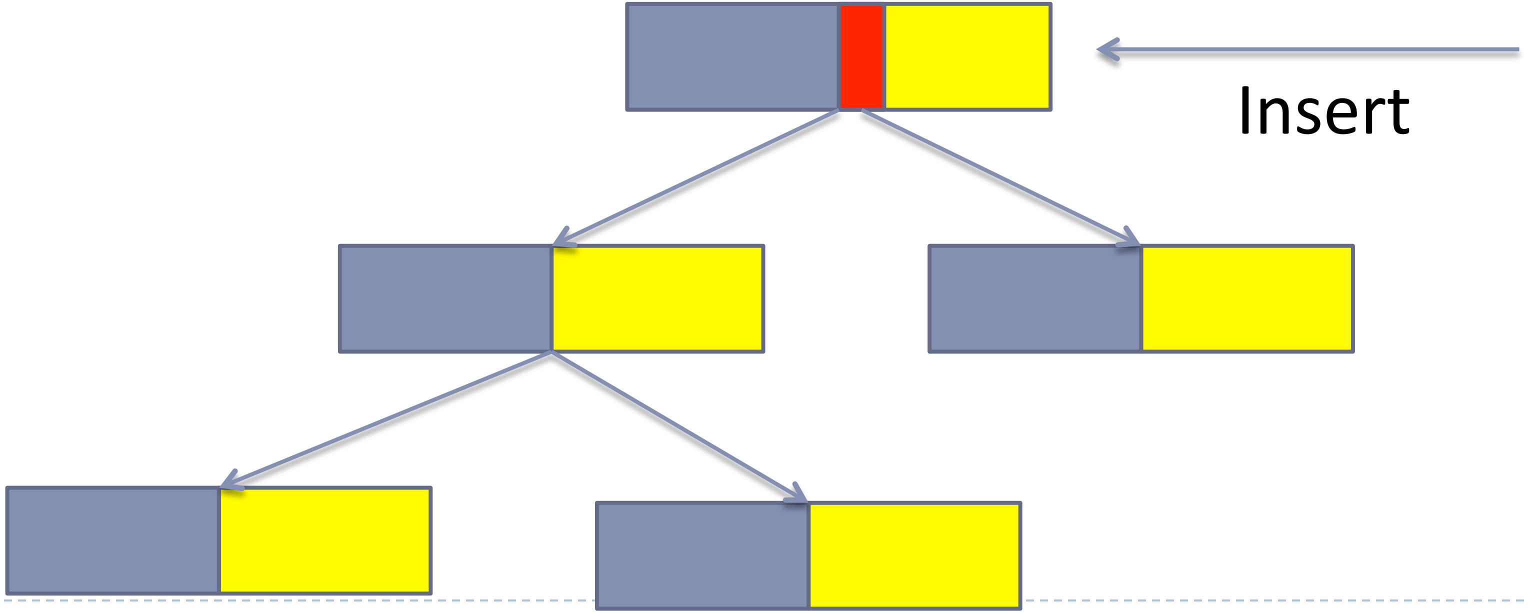
Writes
Reads
Throughput (MB/s)
700
600
500
400
300
200
100
0
SSD(2010)-iodepth 1
SSD(2010)-iodepth 32
SSD(2015)-iodepth 1
SSD(2015)-iodepth 32
HDD(2009)-iodepth 1
HDD(2009)-iodepth 32
4
8
16
32
256
1M
Request size (kB)
Request size (kB)

B^{ε}-Trees

- B-Tree variant that uses buffering to improve inserts
- Similar complexity as B-Tree for point, range queries
- No compactions – unsorted buffers, full index
- Better CPU overhead and I/O amplification
- Worse I/O randomness and size

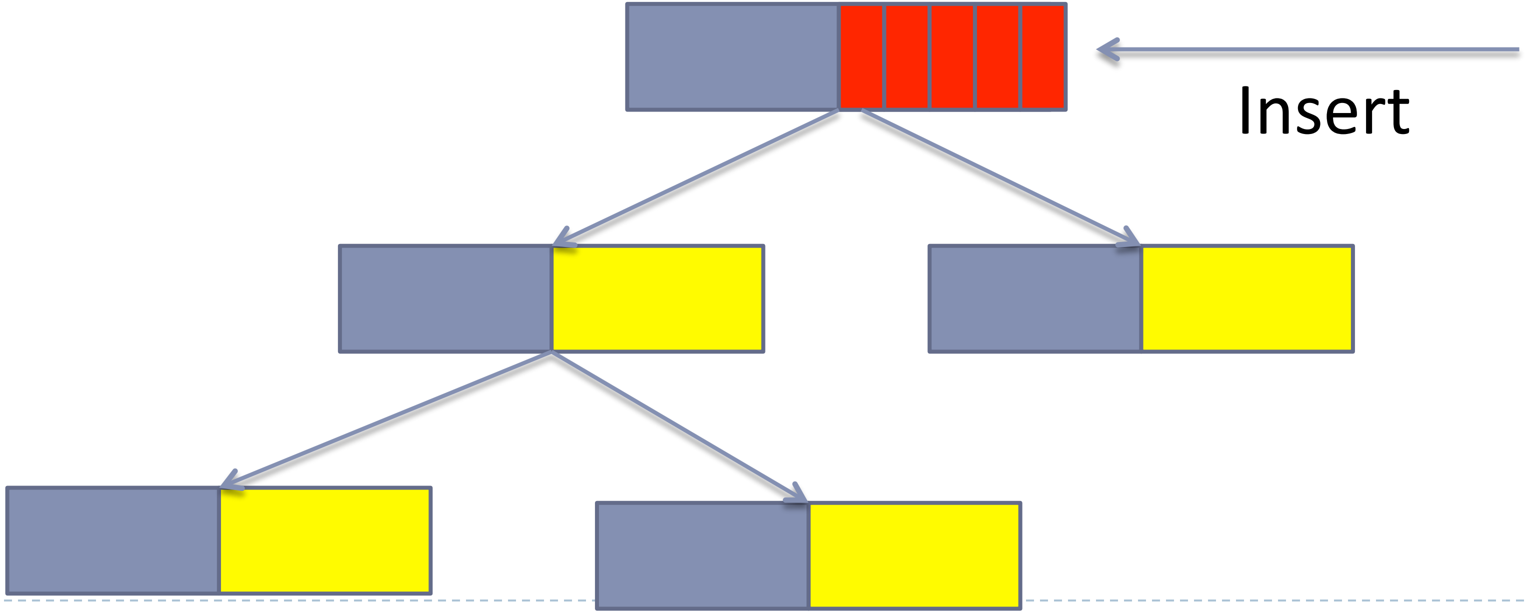
B$^{\varepsilon}$-Trees

- Each internal node has a persistent buffer
- Buffers "log" multiple inserts and use one I/O to device



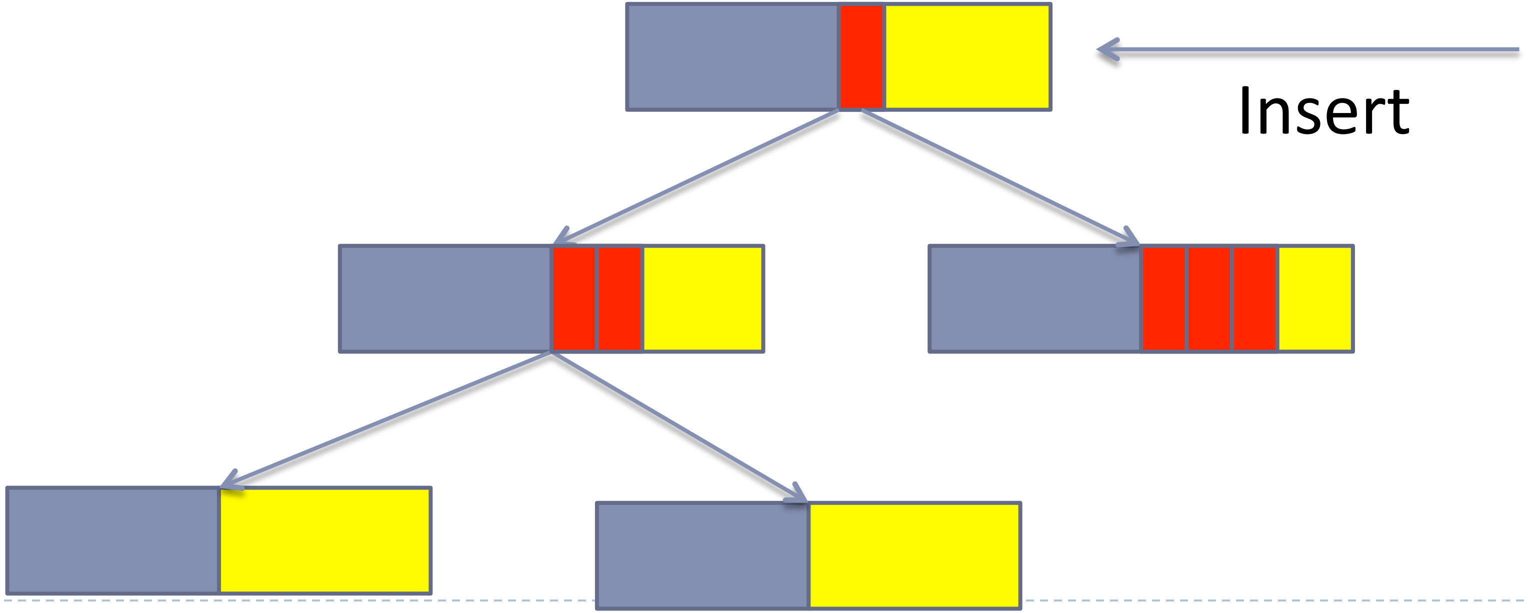
B$^{\varepsilon}$-Trees

- Each internal node has a persistent buffer
- Buffers "log" multiple inserts and use one I/O to device

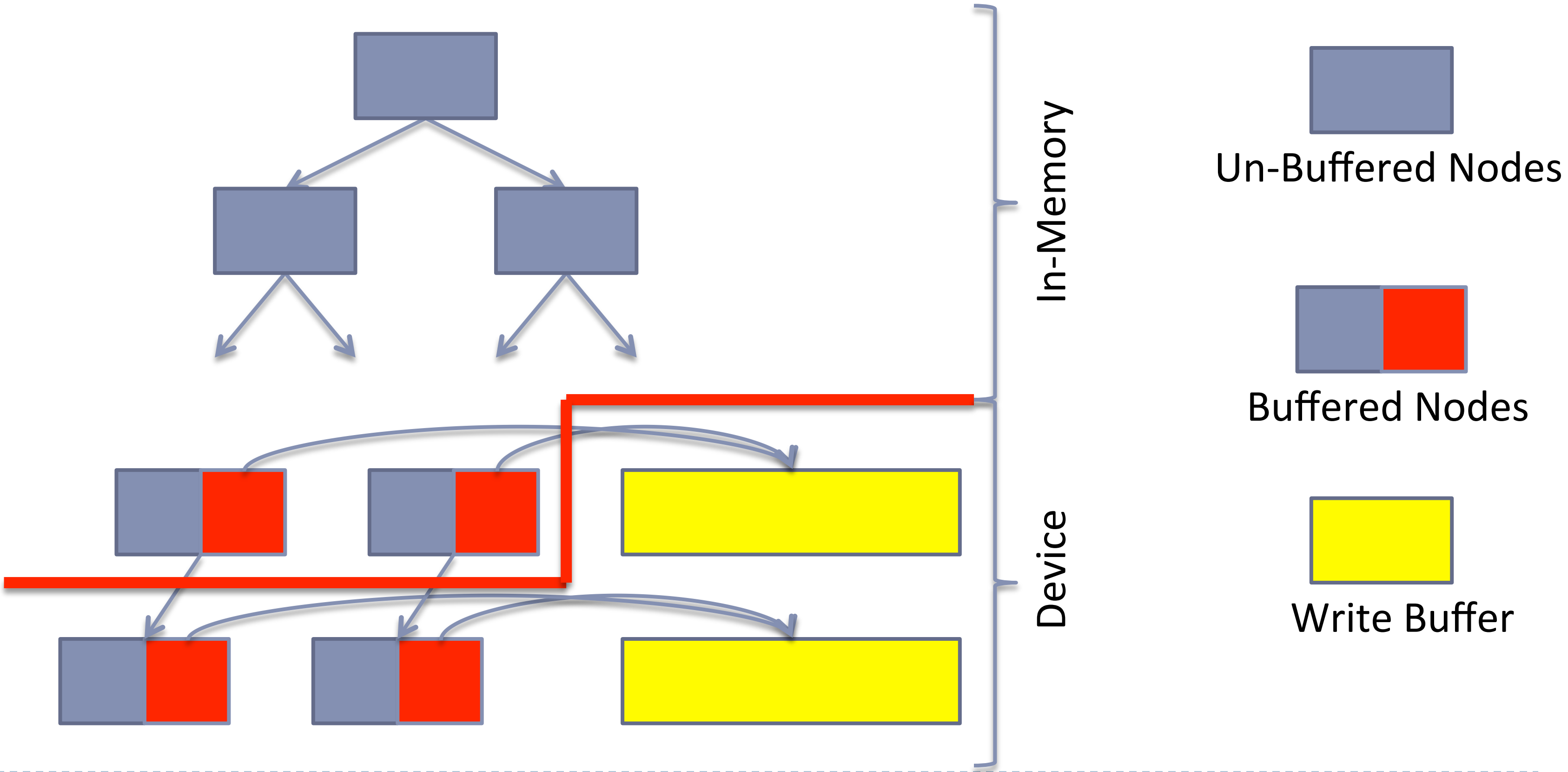


B^{ε}-Trees

- Each internal node has a persistent buffer
- Buffers "log" multiple inserts and use one I/O to device

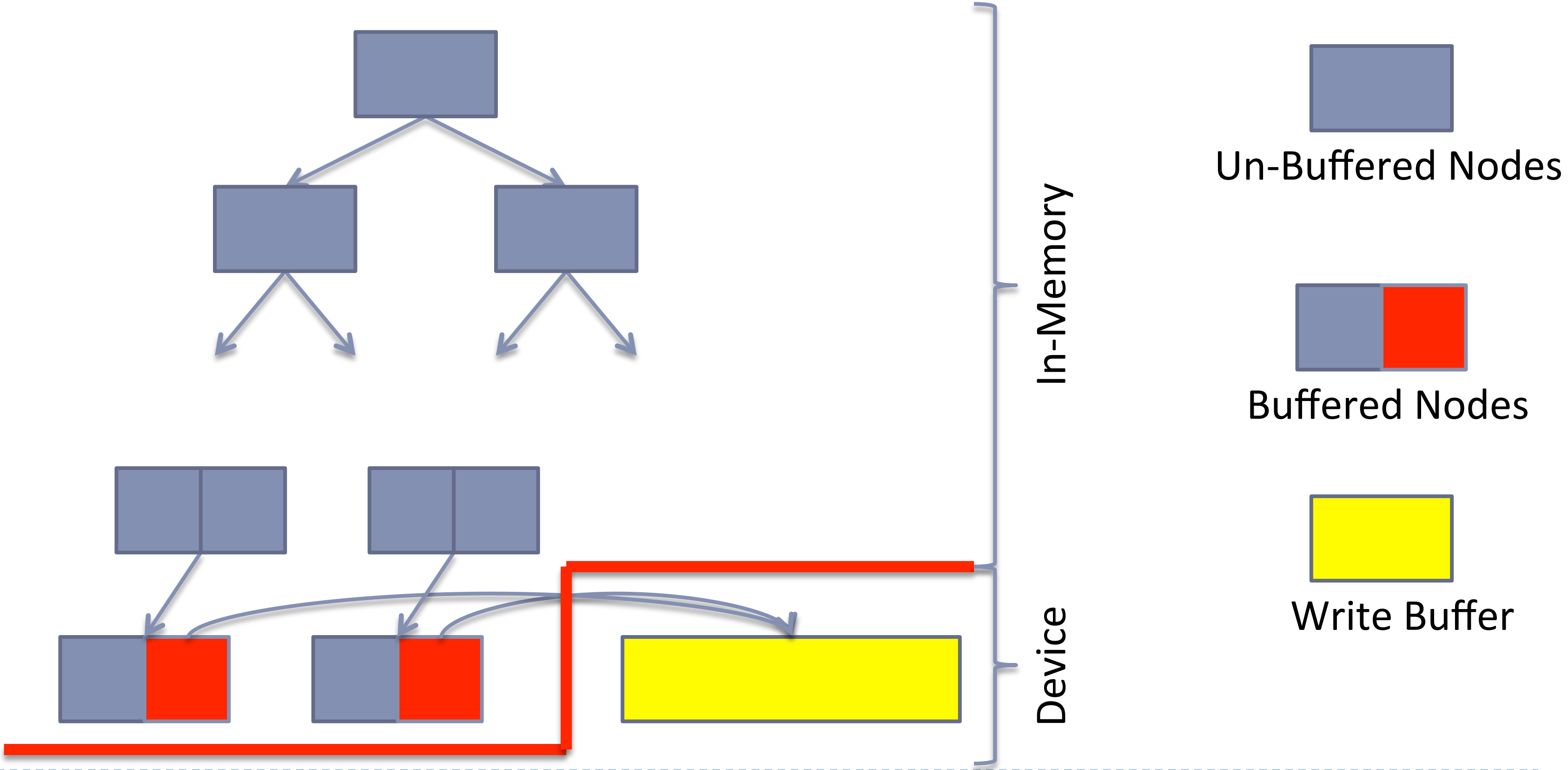


Tucana B$^{\varepsilon}$-Tree



FORTH-ICS
Institute of Computer Science

Tucana B$^{\varepsilon}$-Tree



Buffered Node Organization

- Searching buffered nodes requires accessing keys on device
- Tucana uses two optimizations for buffered nodes
- 1) Include key prefixes (fixed size)
  - Eliminates 65%-75% of I/Os for keys in all queries
- 2) Include hashes for full keys (fixed size)
  - Eliminates 98% of I/Os for keys in point queries

DRAM Caching – Device I/O

- Key-value stores use a user-space DRAM cache
  - Avoids system calls for hits - Explicit kernel I/O for misses
- However, hits incur overhead in user-space
  - **Both** index+data in **every** traversal – Not important for HDDs

Alternative: DRAM caching via mmap

- Use multiple regions/containers per device
- Each region contains allocator + multiple indexes
- mmap each region directly to memory
  - Same layout of metadata + data on device and in memory
- Hits via mapped virtual addresses do not incur overhead
- Misses do not require serialize/deserialize of index
- mmap introduces new challenges

mmap: Misses Cause Page Faults, Fetches, Evictions

- (1) We can improve inserts
- Inserts require a read-before-write I/O
- We insert only on newly allocated pages
- We detect and eliminate fetches to newly allocated pages
  - Requires a kernel module with access to allocator metadata
- (2) Still, no control over size, timing of I/Os + evictions
  - We use mmap hints to disable prefetching
  - Should examine these in detail in future work

Persistence And Recovery

- Typical for HDDs: Write-Ahead-Logging (WAL)
  - Sequential I/O and ability to batch I/Os – both good
  - However, double writes – first to log, then in-place
  - Incurs overhead for log management during recovery
- Alternative: Copy-On-Write (CoW)
  - Instantaneous recovery and amenable to versioning
  - Write-anywhere approach and modify pointers atomically
  - Single write, however, more random I/O

H-Tucana: An Hbase Integration

- Use Tucana to replace HBase’s LSM-based storage engine
- We keep HBase for
  - Metadata architecture
  - Fault tolerance
  - Data distribution
  - Load balancing

Outline of this talk

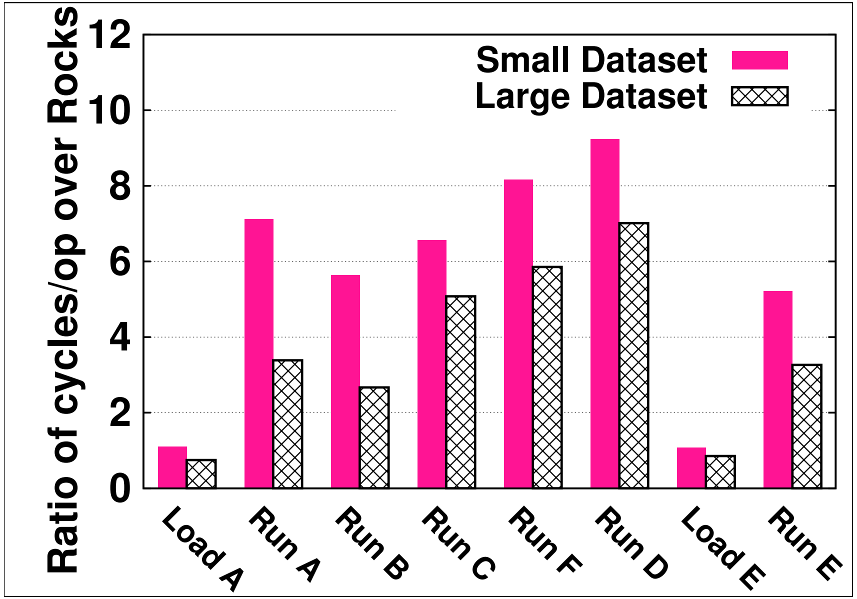
- Discuss our design and motivate decisions
- **Evaluation**
- Conclusions

Experimental Setup

- Compare Tucana with RocksDB
  - H-Tucana with HBase and Cassandra
- Platform
  - 2 * Intel Xeon E5520 with 48GB DRAM in total
  - 4 * Intel X25-E SSDs (32GB) in RAID0
- YCSB – synthetic workloads
  - Insert only, read only, and various mixes
- Two datasets
  - Small dataset fits in memory
  - Large dataset is twice the size of memory
- We examine
  - Efficiency - cycles/op
  - Throughput - ops/s
  - I/O amplification

Efficiency

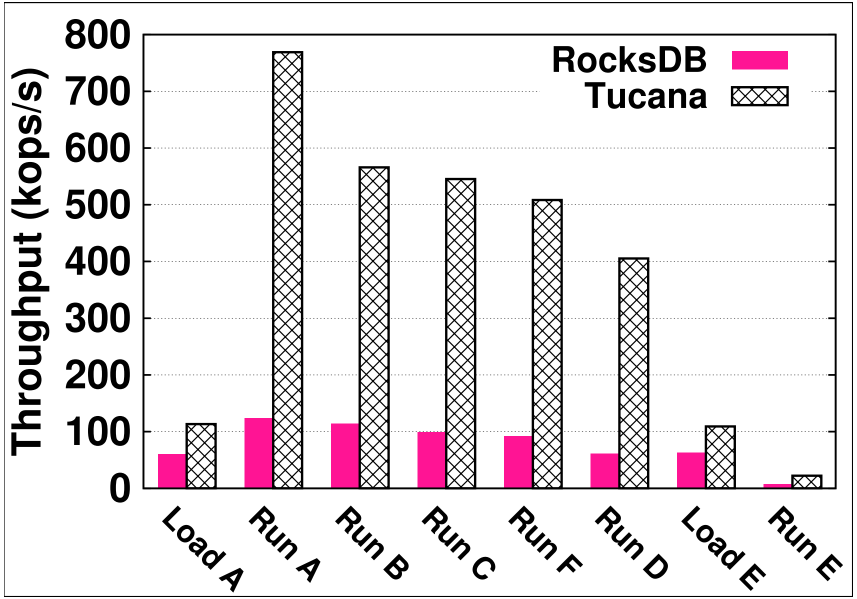
- Improvement over RocksDB in terms of cycles/op



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

Throughput

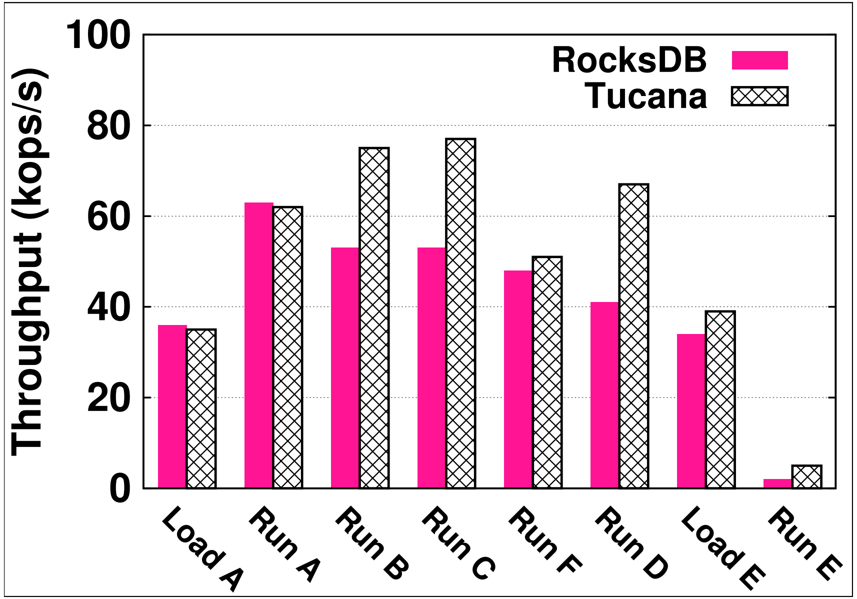
- Comparison with RocksDB in terms of ops/sec



- Small dataset
  - 2x up to 7x
  - 4.5x on average

Throughput

- Comparison with RocksDB in terms of ops/sec



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

Tradeoff: Amplification vs. Randomness (Writes)

- FIO model for I/O pattern of Tucana and RocksDB
- Based on measurements: Tucana has 3.5x less I/O traffic but 49x smaller random I/Os
- For two SSD generations Tucana's approach wins: 4.7x and 3.1x over RocksDB

			SSD (2010)	SSD (2015)
	Write (GB)	Avg. rq_size	time (sec)	time (sec)
Tucana	123	18K	133	32
RocksDB	435	884K	623	100
Ratio	**3.5x**	**49x**	**4.7x**	**3.1x**

Related Work

- Reducing I/O amplification in LSM-Trees
  - WiscKey[FAST'16]: compaction only for keys
  - LSM-trie[ATC'15]: trie of LSM, hash-based structure
  - VT-Tree[FAST'13]: less I/O via container merging
  - bLSM[SIGMOD'12]: bloom filters, compaction scheduling
- BetrFS[FAST'15]: B^{ε}-Trees for file system

Conclusions

- Tucana: An efficient key-value store in terms of cycles/op
  - Target fast storage devices
  - LSM → B^{ε}: overhead of I/O amplification & compactions
  - Explicit I/O → mmap: overhead of DRAM caching
  - WAL → CoW: overhead of recovery
- Tucana: Up to 9.2x/7x better efficiency/xput vs. RocksDB
- H-Tucana: Up to 8x/22x better efficiency vs. HBase/Cassandra

Questions ?

Anastasios Papagiannis

Institute of Computer Science, FORTH – Heraklion, Greece

E-mail: apapag@ics.forth.gr

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

Supported by European Commission under FP7 CoherentPaaS (FP7-ICT-611068), LeanBigData (FP7-ICT-619606), and NESUS COST Action IC1305