Faster: A Concurrent Key-Value Store with In-Place Updates

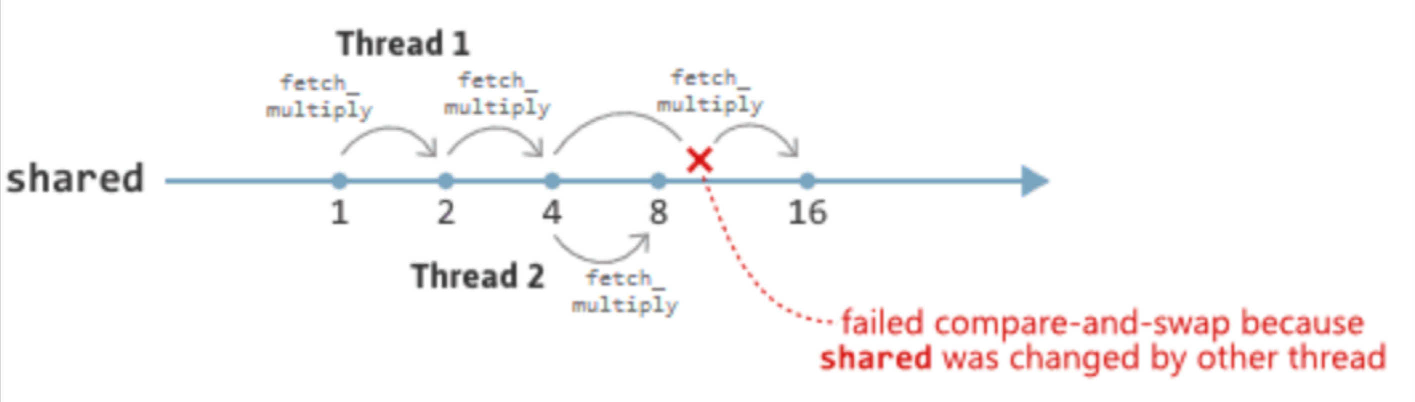
presented by Shirley Hu

High Level

- 1 Design Principle: scalability
- 2 Building Blocks: epoch protection framework + hash index
- 3 Memory Allocators: in-memory + log-structured -> HybridLog

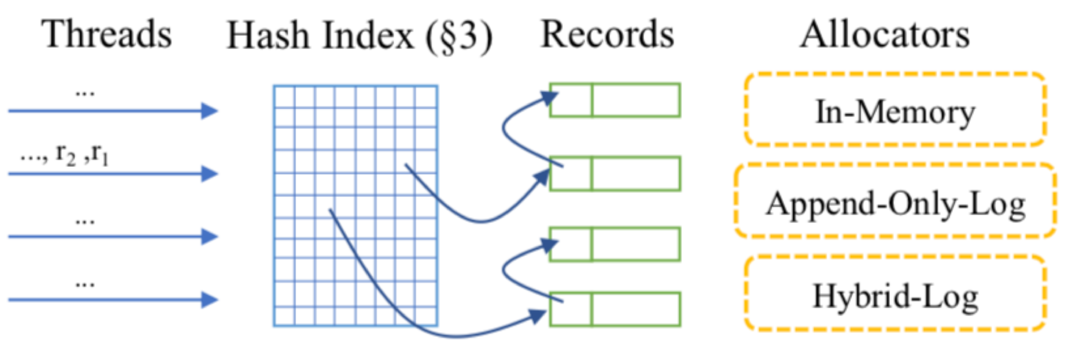
Context and Infrastructure

- Context(not orthogonal)
  - Concurrent: process but not execute all tasks at the same time
  - Atomic: all or nothing, CAS, FAA, FAI
  - Multi-thread
    - Cache-optimized, cache line aligned
      - Each address has fixed size bytes, 64-bit
      - Important to keep track of address in multi-threading
    - Difficult to access otherwise
  - Latch-free: check before apply operations to object
- Infrastructure: 64-bit machine, 2 sockets, single network



System Architecture

- Hash index: later, key not part of it
- Record: linked-list, further studies can be done for entry's history
- Allocators: later, comparison for now



	Record Allocator	**Concurrent & Latch-free**	**Larger-Than-Memory**	**In-Place Updates**
§4	In-Memory	✓		✓
§5	Append-Only-Log	✓	✓	
§6	Hybrid-Log	✓	✓	✓

Figure 1: Overall FASTER architecture.

Building Block #1: Epoch-Based Synchronization (scalable threading model)

- Life cycle: Acquire -> (->BumpEpoch ->) Refresh -> Release
  - Acquire: reserve an entry for T and set Et to E
  - BumpEpoch(action): c -> c+1, add <c, action> into drain-list
  - Refresh: update Et periodically(e.g. 256 operations) to E, Es to current max safe epoch + trigger ready actions in drain-list
  - Release: remove entry for T from epoch table
- Epoch
  - E, Es, Et: for all T, Es < Et <= E
  - Trigger actions: trigger ready actions from drain-list, a list of <epoch, action>, whenever Ec = Es, using compare-and-swap to ensure an action is executed exactly once.
- Scalability: recompute Es and scan through drain-list only when changes in current epoch

Building Block #2: Hash Index (cache-aligned array of 2^*k* hash buckets)

- Organization: <tentative bit, tag, address>
  - Address/offset: 2^k hash bucket for a key with hash value is first identified via the first k bits of h
    - find or delete: identify the hash-bucket with k hash bits and then scan to find the matching tag to operate on operate
  - Tag: increase the effective hashing resolution by reducing hash collisions; next 15 bits of address or offset
  - Tentative bit: two phase insert
    - insert: deterministically choose the first empty slot and mark tentative + rescan to either retry or reset

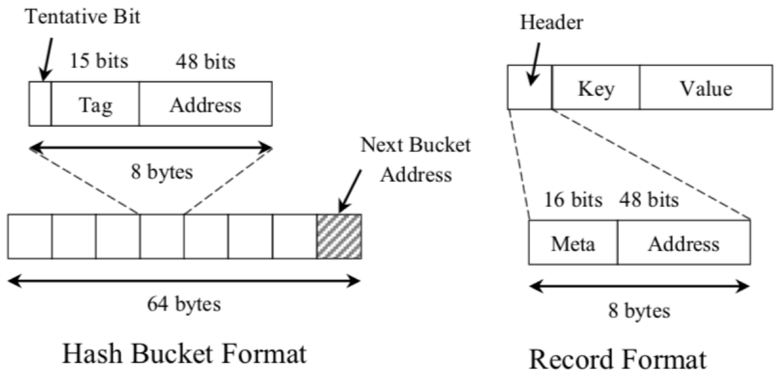


Figure 2: Detailed FASTER index and record format.

Building Block #2: Hash Index - cont. (cache-aligned array of 2^k hash buckets)

- Resize and checkpointing the index
  - On-the-fly: epoch protection(low overhead) and state machine
  - Two versions: double or half the size, and set prepare-to-resize, resizing, and stable states

Interaction with Current Memory Allocators

- In-Memory: store physical address in memory
- Append-Only Log Structured: store logical address in disk

In-Memory: store physical address in memory

- Analysis
  - Pros: Enables latch-free access and in-memory updates
  - Cons: recovery
- Operations for records
  - Reads: find a matching tag, then traverse the linked-list for that entry to find a record with the matching key
  - Updates and Inserts: Blind Update (Upserts) + Read-Modify-Write (RMW)
    - Find the hash bucket entry for the key
      - If doesn't exist: two phase insert
      - If exists: scan the LL to find a record with a matching key
        - If record exists: in-place update
          - Epoch guarantees the thread's access to the memory safety as long as it doesn't refresh its epoch
        - Otherwise: splice the new record into the tail of LL via compare and swap.
  - Deletes : splicing it out of the LL via compare-and-swap either on a record header or hash bucket entry if it's the first record
    - Set entry to 0 to make it available for future inserts
    - Epoch protection enables in-place updates because of each thread's thread-local of drain-list

Append-Only Log Structured: store logical address in disk (existing techniques + epoch protection)

- Log-Structured Allocator Structure
  - Tail offset: points to the next free address
    - Where new record allocation happens via fetch-and-add (reset or retry)
    - Updates epoch when cross page boundaries
      - Flush, and bump epoch current epoch to set flush-status
  - Head offset: tracks lowest logical address
    - Evict pages: increment head offset and bump current epoch with trigger action to set closed-status , once safely offloaded
  - Circular Buffer: fixed-size page frames with a LA each, sector-aligned, to avoid additional memory copies for unbuffered reads and writes

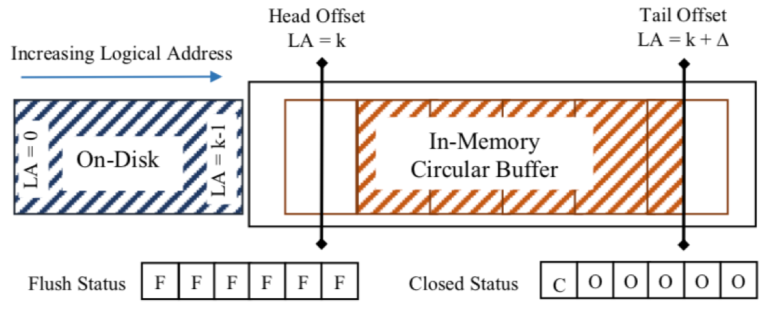


Figure 4: Tail Portion of the Log-Structured Allocator

Append-Only Log Structured: store logical address in disk - cont.
(existing techniques + epoch protection)

- Operations
  - Update and Inserts: same as above, except for:
    - set invalid in header bit and to retry when fails
    - Insert updates to the tail of the log and link to previous record
  - Delete: same as LSM tree, tombstone using a header bit and require log garbage collection
  - Read: check if address is more than current head
    - If yes: like before
    - Otherwise: issue async to request to retrieve the record
      - each user operation is associated with context
      - each thread-local has a pending queue of contexts of completed async requests that refresh periodically

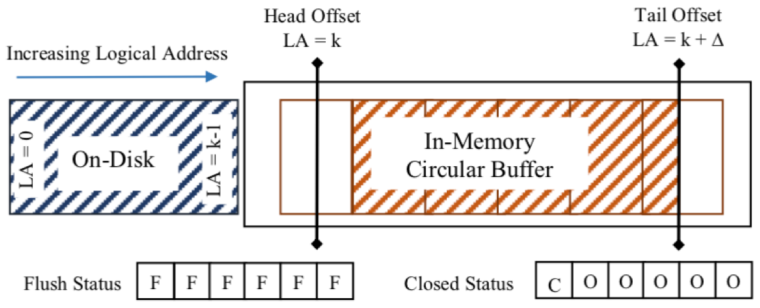


Figure 4: Tail Portion of the Log-Structured Allocator

HybridLog

- Advantages
  - Higher level of cache for more frequently accessed records
  - I: Access path for keys of different hash buckets don't collide
  - L: Updating parts of a larger value is efficient
  - I & L: Most updates don't need to modify the FASTER hash index
- Structure
  - Read-only offset
    - Update: similar as log, except for now we employ Read-Copy-Update
  - Safe read-only offset
    - Problem: both followed epoch correctly yet the RO offset changed, so incorrect result -> two copies of L now
    - Tracks the read-only offset seen by all the threads. The values is between minimum value of read-only offset seen by any active FASTER thread and maximum read-only offset
      - Only one could succeed
- Fuzzy region: region between safe read-only and read-only offset
  - Different updates
  - CRDT: conflict-free replicated data types
    - Each computed as independent partial values that can later be merged
- Recovery and Consistency
  - states: none, only r1, or r1 and r2

HybridLog

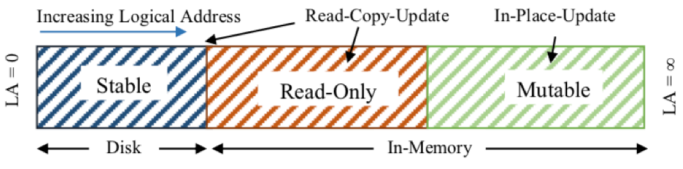


Figure 5: Logical Address Space in HybridLog

Logical Address	Read-Modify-Write	CRDT Update	Blind Update
`Invalid`	Create a new record at tail-end	Create a new record at tail-end	Create a new record at tail-end
< `HeadAddress`	Issue Async IO Request	Create a delta record at tail-end	
< `SafeReadOnlyAddress`	Add to pending list		
< `ReadOnlyAddress`	Create an updated record at tail-end		
< ∞	Update in-place concurrently	Update in-place concurrently	Update in-place concurrently

Table 2: Update scheme for different types of updates

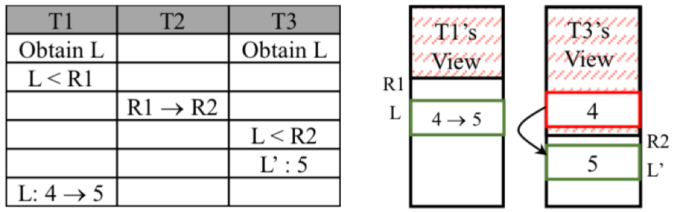


Figure 6: Lost Update Anomaly

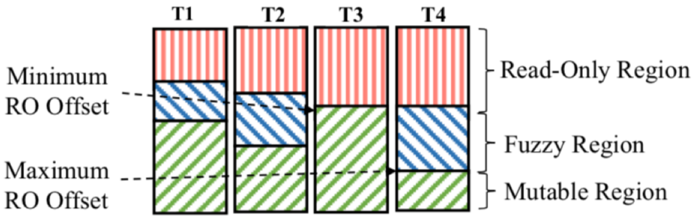


Figure 7: Thread Local View of Hybrid Log Regions

Experiments

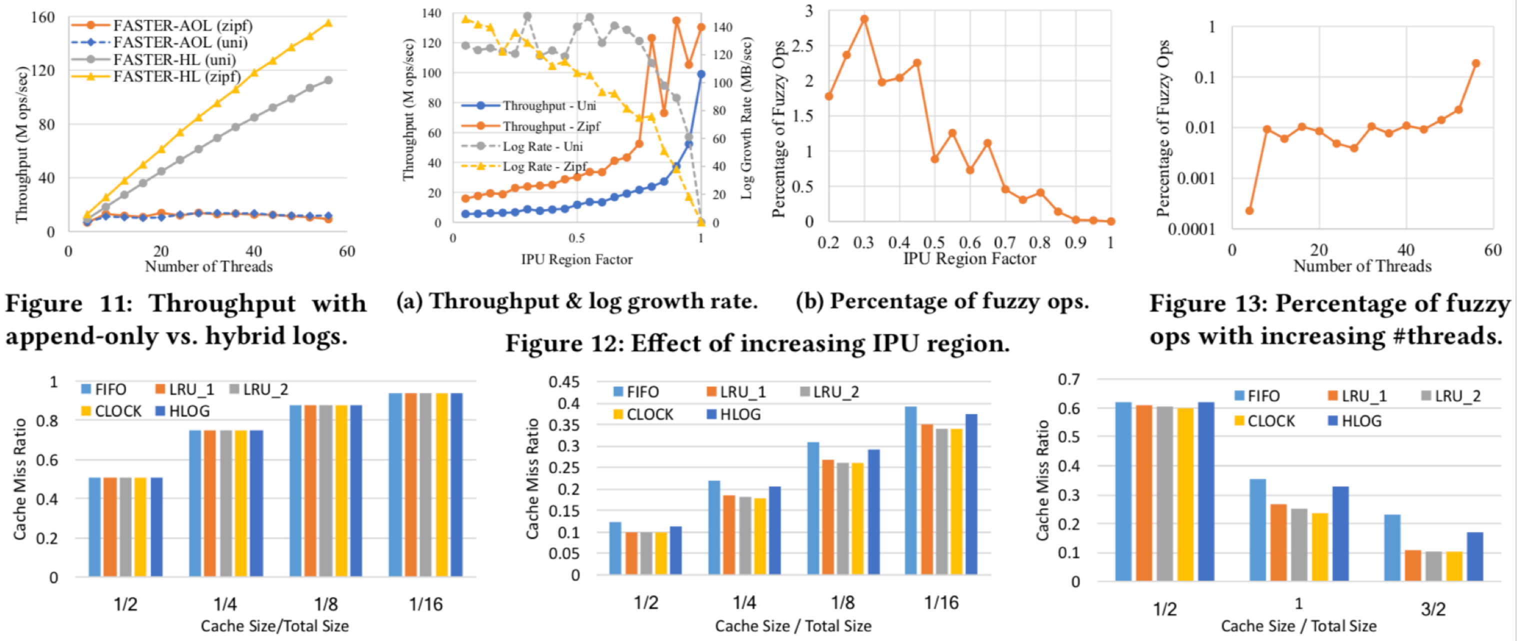


Figure 11: Throughput with append-only vs. hybrid logs.

(a) Throughput & log growth rate. (b) Percentage of fuzzy ops.

Figure 12: Effect of increasing IPU region.

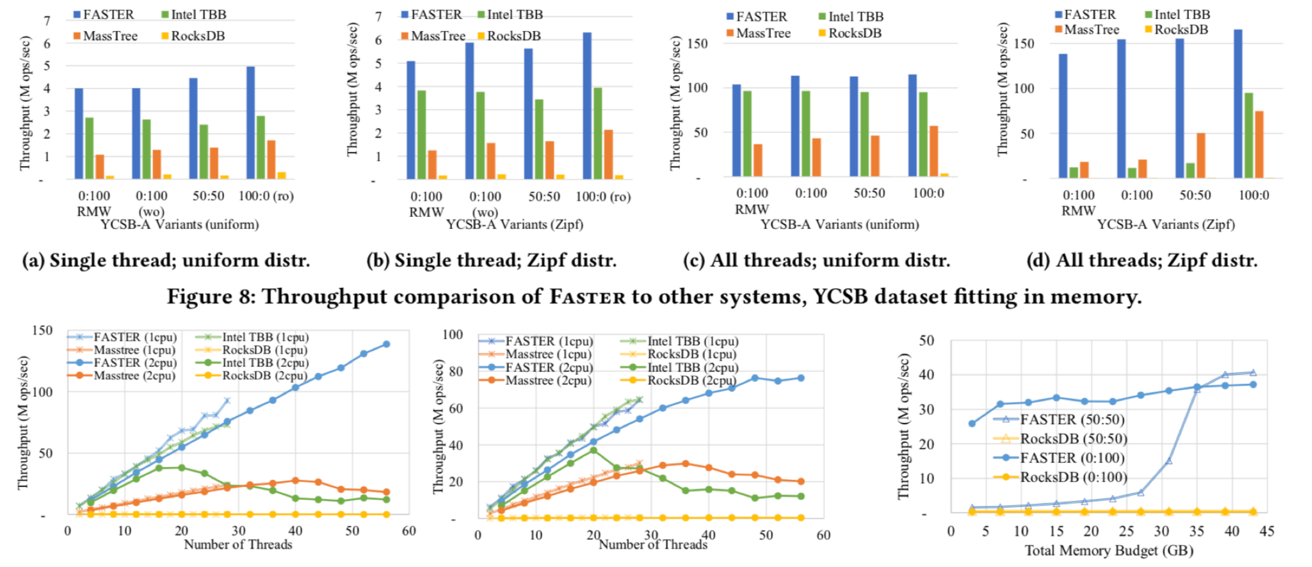
Figure 13: Percentage of fuzzy ops with increasing #threads.

Figure 14: Cache miss ratio (Uniform).

Figure 15: Cache miss ratio (Zipf).

Figure 16: Cache miss ratio (Hot Set).

Experiments



(a) Single thread; uniform distr. (b) Single thread; Zipf distr. (c) All threads; uniform distr. (d) All threads; Zipf distr.

Figure 8: Throughput comparison of FASTER to other systems, YCSB dataset fitting in memory.

(a) RMW updates; 8-byte payloads. (b) Blind updates; 100-byte payloads.

Figure 9: Scalability with increasing #threads, YCSB dataset fitting in memory.

Figure 10: Throughput with increasing memory budget, for 27GB dataset.

Reviews

- Appreciate:
  - connect the research with code implementation
  - analyzed state-of-the-art approaches thoroughly
- Would appreciate:
  - what happens if the queries have high percentage on read-only or other variants?
  - what happens if we could apply more sockets?