Optimizing Linearizable Bulk Operations on Data Structures

Authors:
Matthew Rodriguez,
Michael Spear

Introduction

LEHIGH UNIVERSITY

Background

- Concurrent data structures are important, but hard to design
- Bulk operations pose significant scaling challenge
- Simplest solution is two-phase locking (2PL)

LEHIGH UNIVERSITY

Prior Work

- Atomic snapshot
  - Time and memory cost
  - Staleness
  - Read-only
- Multi-versioning
  - Memory cost
  - Staleness
  - Read-only

LEHIGH UNIVERSITY

Our Contribution

- Propose three algorithms for maps
- Support linearizable bulk operations
  - *Entire* bulk operation, end-to-end, is linearizable
- Linearizability
  - Strong correctness condition
  - Must exist an equivalent sequential ordering
  - Must obey real-time order
- Meet several additional design constraints

LEHIGH UNIVERSITY

Design Constraints

- Focus on system software
- Strict memory bounds
- No leaks, no garbage collector
- Support mutating bulk operations
- Support bulk ops with loop-carried data dependencies
- Generic algorithms applicable to multiple data structures

LEHIGH UNIVERSITY

Motivation

- Primary underlying question: delay when?
- Answer this question using *metadata*
- Differ in location and granularity of metadata

LEHIGH UNIVERSITY

Metadata Granularity

- Low granularity
  - Lack of information
- High granularity
  - More overhead

LEHIGH UNIVERSITY

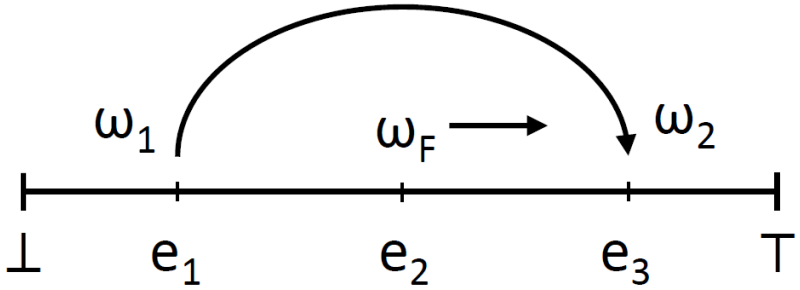
Metadata Location

- Global vs local metadata
- Global
  - Greater risk of contention
- Local
  - More overhead
  - Each thread has access to less metadata
  - Metadata granularity controlled by number of partitions

LEHIGH UNIVERSITY

Cycle Example

- Thread E performs elementals ω_1, ω_2
- Thread F performs foreach operation ω_F
- Due to order of accesses to e_1, $\omega_F \rightarrow \omega_1$
- Due to order of accesses to e_3, $\omega_2 \rightarrow \omega_F$
- Due to temporal order by same thread, $\omega_1 \rightarrow \omega_2$
- $\omega_1 \rightarrow \omega_2 \rightarrow \omega_F \rightarrow \omega_1$ ... A cycle.
- At least one op must delay to prevent this



LEHIGH UNIVERSITY

Insight

- The cycle occurred due to Thread E going from *left* to *right*
- Going from *right* to *left* does not cause a cycle
- ω_F linearizes when Thread E goes from right to left
- Track bulk ops' linearization points

Algorithm 1: Aggressive Ordering

LEHIGH UNIVERSITY

Overview

- Uses *coarse-grained, global* metadata
- Simple algorithm with low overhead
- Assigns total global order to all bulk ops
- Bulk ops ordered by ID
- Bulk op IDs are assigned by creation time
- Bulk ops that start later always ordered later

LEHIGH UNIVERSITY

Metadata

```
Type AOGlobalMetadata
    lastID        : Atomic<Integer64>
    linearizedID  : Atomic<Integer64>
Type AOPartitionLock extends Atomic<Integer64>
    lastVisitorID : Bit[63]
    lockBit       : Bit
```

- `lastID` used to generate bulk op IDs
- `linearizedID` used to track which ops have linearized
- Each partition tracks `lastVisitorID`

LEHIGH UNIVERSITY

Bulk Operation

- Obtain a unique ID, `myID`
- Iterate over each partition:
  - Wait until `lastVisitorID` == `myID` – 1
  - Lock the partition, do the work, unlock
  - Set `lastVisitorID = myID`
- Atomically increase `linearizedID` to `myID`

LEHIGH UNIVERSITY

Elemental Operation

- Get ID of last linearized bulk op, `lastLinID`
  - `lastLinID` represents earliest lin. point, not necessarily the actual lin. Point
- Find partition p containing sought key
- Wait until `p.lastVisitorID >= lastLinID`
- Lock, do the work, unlock
- Atomically increase `linearizedID`
  - Set it to value `p.lastVisitorID` had while locked

LEHIGH UNIVERSITY

Drawbacks

- Unnecessary ordering
  - Disjoint range operations, read-only bulk operations
- Range ops must touch every partition (`lastVisitorID`)
- Ordering by start time not always best

Algorithm 2: Dynamic Ordering

LEHIGH UNIVERSITY

Overview

- Uses *fine-grained, global* metadata
- Seeks to address shortcomings in AO with more metadata

LEHIGH UNIVERSITY

Metadata

- The global metadata consists of a list of sets of `BulkOpMetadata` objects
- `BulkOpMetadata` stores metadata on a single bulk op
- Bulk ops in the same set are unordered wrt each other
- Bulk ops in different sets are ordered by position in list
  - Closer to head = ordered earlier, vice versa

```
Type DOGlobalMetadata
    bulkOps : List<Set<BulkOpMetadata>>
Type BulkOpMetadata
    const startKey : Key
    const endKey   : Key
    const readOnly : Boolean
    lastKey        : Key
    linearized     : Boolean
```

LEHIGH UNIVERSITY

Bulk Operation

- Initialize `BulkOpMetadata` object for op
- Insert `BulkOpMetadata` into global list *as early as possible*
- Iterate over each partition:
  - Atomically: search global list for any ops that *block* this op, and acquire lock on partition
  - Do work, release lock
- Recursively mark this op linearized
- Remove `BulkOpMetadata` object from global list

LEHIGH UNIVERSITY

Elemental Operation

- Atomically calculate a list of all preceding bulk ops
- Determine which partition p contains the sought key
- Wait until all bulk ops on list are done with k
- Acquire partition lock, perform elemental operation
- Linearize any bulk ops which have accessed k
- Release partition lock

Pros & Cons

- Avoids much of the false waiting incurred by Aggressive Ordering
- Increased overhead
- Contention on global metadata
- Accesses to global metadata are frequent and complex

Algorithm 3: Localized Ordering

LEHIGH UNIVERSITY

Overview

- Uses *fine-grained, local* metadata
- Maintains advantages of 2PL (range operations) and builds from there
- 2PL achieves linearizability by delaying operations from *starting*
- However, it is sufficient to delay them from *returning*

LEHIGH UNIVERSITY

Metadata

```
Type OLPartitionLock
    started   : AtomicQueue<ThreadID>
    completed : AtomicQueue<ThreadID>
```

- No global metadata
- Per-partition metadata consists of two queues
- Any operation must reach head of `started` queue to access object
- Any operation must reach head of `completed` queue to return
- `completed` queue increases concurrency without sacrificing correctness

LEHIGH UNIVERSITY

Bulk Operation

- For each partition p:
  - Enqueue in `p.started`
  - Dequeue self from `started` for previous partition (if exists)
  - Wait until head of `p.started`
  - Operate on `p`
  - Enqueue in `p.completed`
- Dequeue from last `started` queue
- Dequeue from all `completed` queues

LEHIGH UNIVERSITY

Elemental Operation

- Determine the partition p containing the sought key
- Enqueue in `p.started` queue
- Wait until head of `p.started`
- Do work
- Enqueue in `p.finished`
- Dequeue from `p.started`
- Wait until head of `p.finished`
- Dequeue from `p.finished`

Evaluation

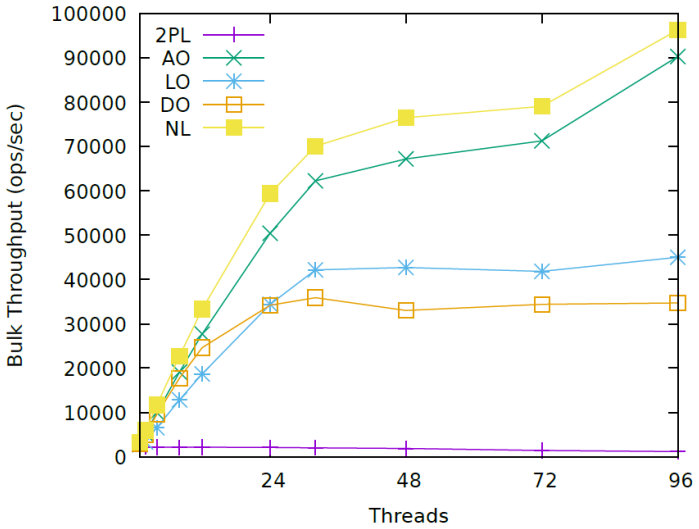
LEHIGH UNIVERSITY

Overview

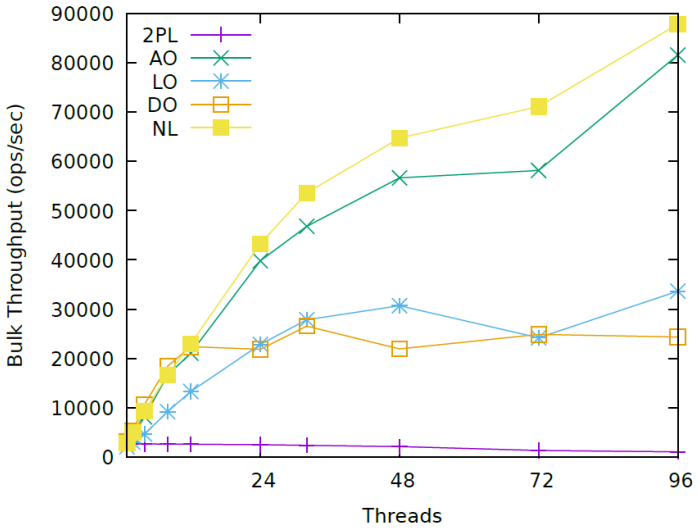
- Compare AO, DO, and LO on two data structures
- Chunked skip list of our own design
  - Ordered
- Interlocked Hash Table (IHT)
  - Unordered; therefore, no range ops
- Baseline: Two-Phase Locking
- Upper bound: an implementation where *bulk operations are not linearizable* (NL)

LEHIGH UNIVERSITY

Foreach-Only Workload

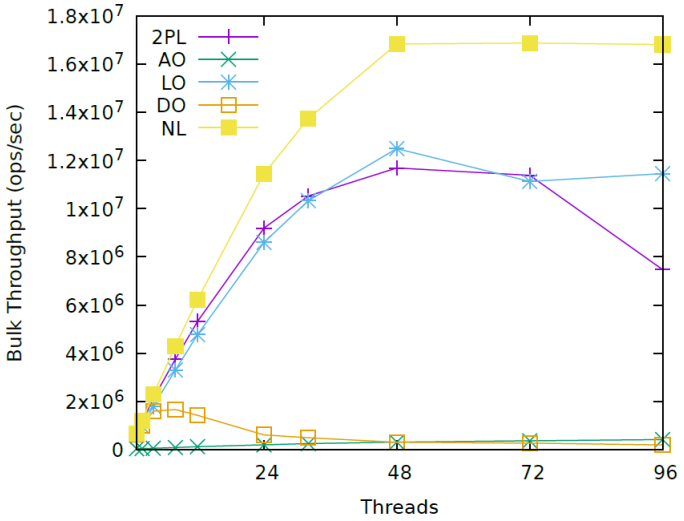


Skip List

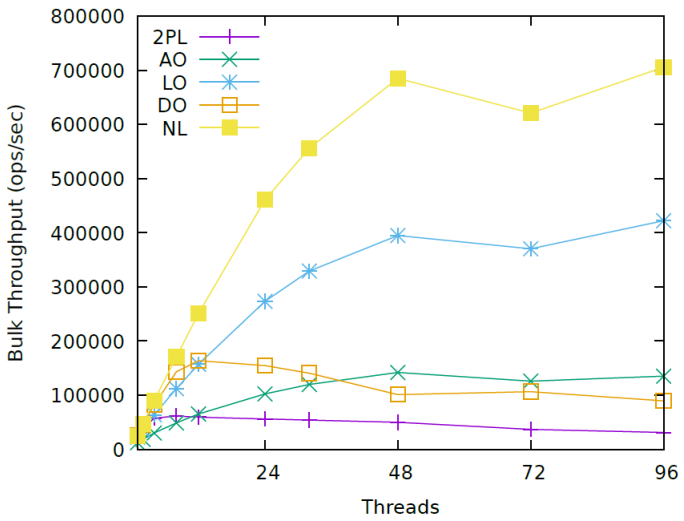


IHT

Range-Only Workload

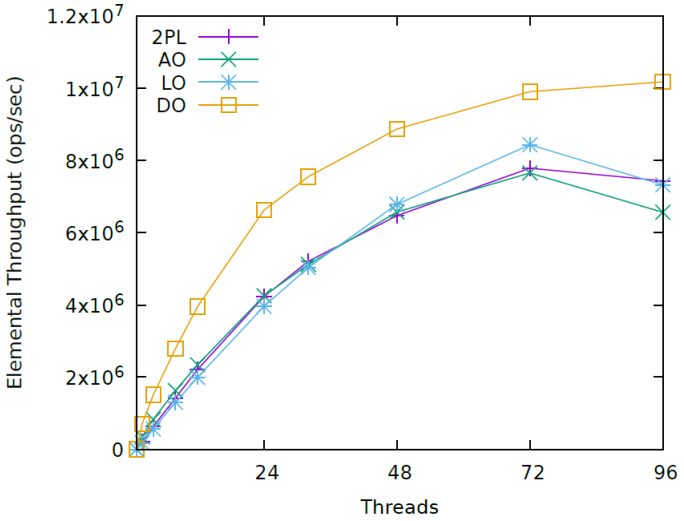


Range Length 4,096

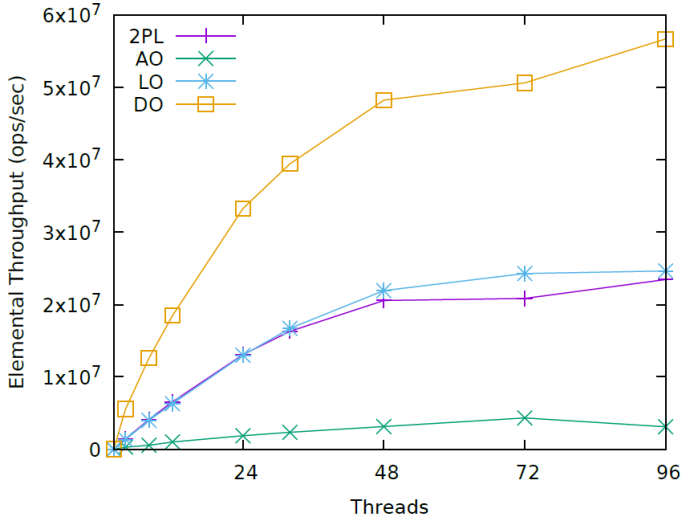


Range Length 131,072

Mixed Workload



1 Foreach Thread



2 Range Threads, Length 131,072

Conclusion

LEHIGH UNIVERSITY

Conclusion

- We introduce three algorithms for scalable linearizable bulk ops
- Performance exceeds 2PL
- Each algorithm has a niche; no single one is best
- Please see the paper for discussion about partition size

Thank you