Optimizing Linearizable Bulk Operations on Data Structures

Matthew Rodriguez
mar316@lehigh.edu
Lehigh University
Bethlehem, PA

Michael Spear
spear@lehigh.edu
Lehigh University
Bethlehem, PA

ABSTRACT

We study the problem of ensuring the correctness of concurrent programs that perform mutating foreach and range operations over concurrent data structures. We introduce three algorithms which vary in the location and the granularity of concurrency control metadata. Our algorithms make the linearization of bulk operations visible to concurrent elemental operations, which enables them to scale well, keep overhead low, and operate within tight memory bounds. In our experimental evaluation, we demonstrate that our techniques do not hinder the performance of elemental operations in elemental-only workloads, and allow scalability among concurrent mutating bulk operations. Furthermore, in mixed workloads, our algorithms outperform the baseline, sometimes by an order of magnitude or more.

CCS CONCEPTS

• **Computing methodologies → Concurrent algorithms**.

KEYWORDS

Linearizability, Synchronization, Concurrency

ACM Reference Format:
Matthew Rodriguez and Michael Spear. 2020. Optimizing Linearizable Bulk Operations on Data Structures. In *49th International Conference on Parallel Processing - ICPP (ICPP '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3404397.3404414

1 INTRODUCTION

Data structures are at the heart of software development, and concurrency has become one of the most critical techniques for achieving high performance. Consequently, concurrent data structures are a foundational building block for modern software. Modern languages, like Java, provide entire libraries of concurrent data structures, dozens of papers are published on the topic each year, and there are even books dedicated to the topic [7].

Unfortunately, designing concurrent data structures is hard, and one of the most common compromises is to focus on the performance of elemental operations (e.g., insert, lookup, remove, and update of a single element). Atomic bulk operations, such as foreach and range, still pose a significant scaling issue, with no clear single best solution—especially when those operations may mutate the data structure. The simplest solution is two-phase locking [5], but this restricts other threads from accessing large portions of the map at a time, hurting the overall throughput of the system.

Existing research has already proposed and studied several solutions for supporting read-only bulk operations. One common technique is to create an atomic snapshot [1, 13], and perform the bulk operation on that. While this mitigates the impact on throughput, it can cost significant processor time and memory to produce a snapshot of a large map. Furthermore, although the bulk operation sees a single, consistent, unchanging view of the data structure, that view may be considerably out of date by the time the operation completes. There is also no clear way to extend this technique to support mutating bulk operations.

Another common technique is multi-versioning [3] It keeps multiple versions of each element in the data structure, so that when a bulk operation reaches an element, it may use an older version of it in order to preserve atomicity. As with atomic snapshots, this comes at a significant memory cost, and has no clear extension for mutating bulk operations. Modern approaches to multi-versioning can achieve high performance for read-only range queries by keeping a history of every change ever made [6]. Other techniques, which integrate more closely with a garbage collector, can reclaim memory [2]. However, even these more practical approaches can have a high worst-case memory overhead.

In this paper, we propose and evaluate three algorithms for performing mutating atomic bulk operations in linearizable concurrent maps. We are focused on solutions for systems software, and thus (1) we must operate within strict memory bounds; we cannot leak memory or rely on a garbage collector. (2) We are restricted to single-version maps, as multi-versioning costs significant space and does not naturally support mutating bulk operations. (3) We wish to support bulk operations with loop-carried data dependencies, so parallelizing the bulk operation itself is impossible. And (4) while we focus on maps, we seek generic algorithms that can be applied to multiple different map implementations—in this paper, we explore their application to a practical hash table [9], and a high-performance ordered map [12]. Naturally, as our concurrency algorithms and data structure implementations support the map interface, it is trivial to support the set interface as well.

At the heart of the problem lies one fundamental question: when must a thread delay to preserve the correctness of an on-going bulk operation? In order to answer this question, each of our algorithms tracks metadata about each thread currently operating on the map. Where they differ is in the location of the metadata and the granularity of the metadata.

Metadata can be stored in a shared global data structure accessed by all threads, or spread apart and stored locally in the individual

ICPP '20, August 17–20, 2020, Edmonton, AB, Canada
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8816-0/20/08…$15.00
https://doi.org/10.1145/3404397.3404414

nodes of the data structure. Each choice has advantages and drawbacks. If metadata are stored globally, contention is more likely. However, if they are stored locally, there may be more overhead required to keep metadata up-to-date in multiple places. Also, an individual thread will only ever have access to the metadata present in a single node at a time, and is therefore more likely to suffer *false waiting* than if the metadata was stored globally. We define *false waiting* as the undesirable situation in which a thread could correctly operate on a key in the map without violating linearizability, but is unable to be certain of that from the metadata it has access to, and is therefore forced to wait unnecessarily. The exact conditions where false waiting occurs vary by the algorithm, and will be described in more detail in Section 3.

The other dimension is the granularity of metadata. With little metadata, we again face the problem of false waiting due to a lack of information. However, with more metadata, overhead must increase. Furthermore, when metadata are stored locally in nodes, the granularity of metadata is controlled directly by the number of nodes. The granularity of metadata that is ideal for the performance of bulk operations may not be the same as the granularity that is ideal for the overall performance of the map—in such a case, a compromise between the two will have to be made.

In Section 2, we introduce the challenges that our algorithms aim to address. Section 3 then presents our three algorithms. We discuss implementation issues in Section 4, and then conduct a performance evaluation on a large multicore machine in Section 5. Finally, we conclude with a discussion of future research opportunities in Section 6.

2 DEFINITIONS AND CHALLENGES

Linearizability is one of the strongest correctness criteria for concurrent data structures [8]. To determine if a data structure's operations linearize, its behaviors must be related to the behavior of an equivalent sequential data structure. Informally, two properties must hold: (1) At any point in time, the state of the data structure must be equivalent to one in which each operation happened at a unique instant; and (2) For each operation on the data structure, the moment at which the operation appeared to happen must obey real-time order, e.g., it appears to happen after all preceding instructions by the thread, and before subsequent instructions by the thread. Operations that are not concurrent must happen in the real-time order in which they were issued by the corresponding threads, but operations that overlap may occur in an any order, so long as the behavior is indistinguishable from a correct sequential history.

As an example, suppose that thread t_a begins operation ω_a at time b_a, and completes the operation at time c_a. Further, suppose that t_b performs ω_b from time b_b to c_b. If $c_a < b_b$, then the effect of ω_a must be visible to ω_b. However, if $b_a < b_b \wedge c_a > b_b$, then ω_a and ω_b are concurrent. The behavior of the data structure must be equivalent to one in which either ω_a or ω_b happens first, but linearizability does not prescribe which. The flexibility this affords is the basis for good scaling.

For the *elemental* operations of a concurrent data structure (get, insert, remove, or update of a single element), linearizability is well understood. Our focus is when a single operation on a concurrent data structure depends on (and possibly changes) the values stored at multiple elements. We define a *read-only bulk operation* as one whose result depends on the values of many elements in the data structure, but which does not change those values. A *mutating bulk operation* is like a read-only bulk operation, but also *may* change values. For a bulk operation to be linearizable, the execution history of the operation must be equivalent to one in which all of the accesses of elements made during the operation happen without any interleaving by other threads' operations on the data structure. That is, if operation ω_a operates on elements $e_i \dots e_j$, then the result of ω_a should be indistinguishable from one in which no other thread concurrently operated on elements $e_i \dots e_j$.

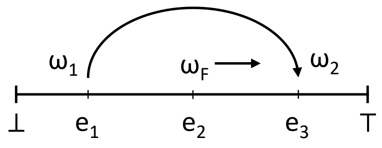


Figure 1: One thread performs elemental operations ω_1 and then ω_2, as another thread performs bulk operation ω_F, creating a cycle in the execution history: $\omega_1 \to \omega_2 \to \omega_F \to \omega_1$.

For a bulk operation on a map to be linearizable, it is often necessary to restrict other threads from accessing certain elements. Such restrictions typically begin *before* the bulk operation reaches those elements, and/or continue *after* the operation is done accessing those elements. We illustrate why using the example in Figure 1.

The horizontal line represents an ordered concurrent map. Two threads are currently using the map. The first, t_1, is performing ω_F, a *foreach* operation, which is a bulk operation which operates on every single key in the map. ω_F is currently processing element e_2. Thus, it has already finished with e_1 and has not yet reached e_3. A second thread, t_2, performs two elemental operations (e.g., lookup, insert, or remove) on the map. The first elemental operation, ω_1, accesses element e_1. With its second, ω_2, it wants to modify e_3.

Suppose it is allowed to do so. ω_F accessed e_1 before ω_1 did, which means it happened before it: $\omega_F \to \omega_1$. Next, because ω_2 accessed e_3 before ω_F, we have $\omega_2 \to \omega_F$. Finally, because ω_1 and ω_2 are two operations performed in order by a single thread, we have $\omega_1 \to \omega_2$. This means we have $\omega_F \to \omega_1 \to \omega_2 \to \omega_F$, which makes a cycle, violating the linearizability of ω_F. Thus, it is necessary to do one of two things. t_2 must either be forbidden from performing ω_2 until ω_F reaches e_3, or it must be forbidden from accessing e_1 until ω_F is completely finished.

The most well-known solution to this problem is two-phase locking [5] (2PL). 2PL requires that a thread lock each element before accessing it, and also that an operation proceed in two phases: one in which it acquires locks, and another in which it releases locks. There are two common 2PL approaches. In *eager acquisition*, ω_F would lock every element (e.g., $\perp \dots \top$) before operating on e_1. Then, after it finished visiting an element e_i, it could unlock that element before moving on to e_{i+1}. In *lazy acquisition*, ω_F would acquire locks when operating on nodes, but would not release any locks until after it finished its operation on the last element (e.g., $\top$). If l_i indicates acquiring a lock on e_i, o_i indicates operating on e_i, and u_i indicates releasing a lock on e_i, then in Figure 1, eager 2PL's history would be

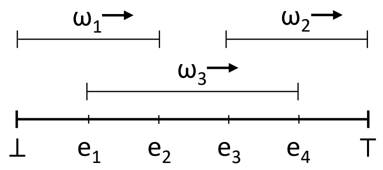


Figure 2: A new range operation ω_3 creates an order between two range operations that do not overlap

$\{l_\perp, l_1, l_2, l_3, l_\top, o_\perp, o_1, u_1, o_2, u_2, o_3, u_3, o_\top, u_\top\}$, whereas lazy 2PL's history would be $\{l_\perp, o_\perp, l_1, o_1, l_2, o_2, l_3, o_3, l_\top, o_\top, u_\perp, u_1, u_2, u_3, u_\top\}$.

We observe that it is possible to maintain linearizability without holding so many locks for so long. In this example, a cycle was created when t_2 went from performing an operation on an element ω_F had already processed—its "left" side—to an element it had not yet processed—its "right" side. If t_2 had only performed operations on one side of ω_F, there would have been no cycle. Furthermore, if t_2 had gone from performing operations on ω_F's right to performing operations on its left, there still would have been no cycle (provided that it did not then go *back* to performing operations on its right side.) If t_2 performs numerous elemental operations on the map, with ω_n being its last operation on the right side of ω_F and ω_{n+1} being its first operation to the left of ω_F, this means ω_F's linearization point is between the two: $\omega_n \to \omega_F \to \omega_{n+1}$. Thus, one way to preserve the linearizability of ω_F is to track whether it has linearized, and forbid threads from accessing elements to its right once it has. This is the first example of a metadatum that our algorithms could track to offer linearizable bulk operations.

Figure 2 presents a more complex example, involving range operations. A range operation is a bulk operation which only operates on keys within a specified range, rather than every element in the data structure as a foreach does. In this example, we have three range operations, ω_1 through ω_3, each being performed by a different thread. At first, we only have ω_1 and ω_2. They are not yet ordered, and there is indeed no need to order them—they operate on disjoint ranges of elements ($\{\perp \dots e_2\}$ and $\{e_3 \dots \top\}$, respectively). Then ω_3 on e_1 through e_4 begins, overlapping both ω_1 and ω_2. Suppose ω_1 has not yet linearized, and has not yet reached e_2. In this case, it would be permissible for ω_3 to order itself before ω_1 and begin immediately: $\omega_3 \to \omega_1$. However, if ω_2 has already accessed e_3, ω_3 will reach e_3 after it and therefore $\omega_2 \to \omega_3$. Thus, we have $\omega_2 \to \omega_3 \to \omega_1$. By transitivity, we have $\omega_2 \to \omega_1$, even though no ordering existed between them before. Thus, if ω_1 is to complete its operation first, it must somehow first indicate that both ω_2 and ω_3 linearized before it. Even though ω_1 and ω_2 have disjoint ranges, ω_1 is being forced to have an effect on ω_2's state. Depending on the implementation, this might sacrifice disjoint access parallelism.

2PL avoids this problem. Without loss of generality, assume the threads use 2PL with lazy acquisition. In the above example, ω_1 will block when it attempts to lock e_1, which is locked by ω_3, but ω_3 will block when it attempts to lock e_3, which is locked by ω_2. ω_3 cannot make progress until ω_2 completes, and ω_1 cannot make progress until ω_3 completes. Hence ω_1 need not interact with ω_2's metadata, but the ordering $\omega_2 \to \omega_3 \to \omega_1$ is preserved.

Listing 1: Data types used by Aggressive Ordering

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

There is an opportunity cost involved in the behavior of 2PL. Consider a scenario where two threads are performing foreach operations, ω_1 and ω_2, with $\omega_1 \to \omega_2$. Assuming no other operations occur in the data structure concurrently with ω_1 and ω_2, there is no reason ω_2 should not be able to "convoy" behind ω_1, acquiring and processing each element immediately after it is released by ω_1. However, with 2PL, ω_2 cannot acquire or operate on even the first element in the map until the first, ω_1, has acquired every lock in the map and progressed to its release phase. As this waiting is more coarse than necessary in this scenario, we call it *false waiting*.

3 COORDINATING LINEARIZABLE BULK OPERATIONS

We now present three algorithms for enabling concurrent bulk linearizable operations, which are designed to reduce false waiting as compared to 2PL. The algorithms consider trade-offs between two issues: the granularity of metadata and the location of metadata. Our algorithms are applicable to a variety of ordered and unordered map and set implementations.

We assume that the universe of keys stored by a map (or set) can split into "partitions", such that a bulk operation can easily find and operate on one partition at a time, according to a predetermined order. Many data structures have natural, inherent partitions; examples include the "buckets" of a closed-addressing hash table [11] or the leaf nodes of a B+ tree [4]. Associating concurrency control metadata with partitions, instead of individual key/value pairs, reduces the overhead of metadata manipulation for bulk operations. Visiting partitions in an agreed-upon order prevents deadlocks and cycles. Section 4 discusses implementation challenges related to partitioning in high-performance data structures.

3.1 Aggressive Ordering

Our Aggressive Ordering (AO) algorithm assigns a total global order to all bulk operations, even when they are read-only or operate on disjoint ranges. In terms of our design considerations, AO uses *coarse grained* global metadata, coupled with a small amount of extra state in each partition.

The metadata for AO appears in Listing 1. The global metadata consists of two atomic integers. lastID generates IDs for new bulk operations. In our discussion of AO, the notation ω_i will denote a bulk operation with the ID i. All bulk operations are ordered by their ID, ascending—$\forall n, m \in \mathbb{N}$, if $n < m$, then $\omega_n \to \omega_m$. The local metadata for AO is a single integer colocated with each partition in the data structure. Our AO implementations, which use spinlocks, colocate this integer with the lock. The colocation is not a requirement, but it offers good performance in the common case. Listing 1 presents the AOPartitionLock as the combination of an atomic integer (lastVisitorID) and a lockBit. lastVisitorID records the ID of the last bulk operation to visit the partition.

Listing 2: Operation coordination for Aggressive Ordering

```
   function $elemental$(k)
1      $\Omega_P \leftarrow$ linearizedID // get earliest linearization time
2      p $\leftarrow$ $getPartition$(k)
       // wait for preceding, active bulk ops to visit p
3      while p.lastVisitorID $< \Omega_P$ do wait
4      p.$acquireLock$()
5      . . . // perform elemental operation
6      $\Omega_L \leftarrow$ p.lastVisitorID // remember preceding bulk ops
7      p.$releaseLock$()
       // linearize all preceding bulk ops
8      atomic linearizedID $\leftarrow$ $max$(linearizedID, $\Omega_L$)
   function $bulk$()
       // get order for this op
9      $\omega \leftarrow$ lastID.$incrementAndFetch$()
10     foreach p $\in$ partitions do
           // wait for preceding bulk ops to visit p
11         while p.lastVisitorID $< \omega - 1$ do wait
12         p.$acquireLock$()
13         . . . // perform operations on p
           // allow subsequent ops to access p
14         p.lastVisitorID $\leftarrow \omega$
15         p.$releaseLock$()
       // ensure all ops know this bulk op has linearized
16     atomic linearizedID $\leftarrow$ $max$(linearizedID, $\omega$)
```

Listing 2 sketches how this metadata is used to coordinate operations. We begin by discussing bulk operations. Since AO gives a total order to each operation, each bulk operation begins by obtaining a unique order (line 9). As bulk operation ω_{n+1} traverses partitions, it waits until a partition's lastVisitorID equals n; only then is it allowed to operate on the partition. When it completes, it updates lastVisitorID, which enables the next bulk operation to proceed with the partition. When a bulk operation finishes with its last partition, it updates linearizedID. Delays between lines 15 and 16 can result in threads attempting to update linearizedID out of order. The problem is avoided by atomically updating the value only if the update would increase linearizedID.

Elemental operations use linearizedID to place a lower bound on the logical time at which they can linearize. An elemental operation ω_e begins by reading linearizedID and storing its value in Ω_P (line 1). Ω_P represents the set of bulk operations that must be ordered before ω_e, making it the set of ω_e's *predecessors.* In AO, Ω_P consists of all bulk operations ω_i where $i \leq P$, and so it can be implemented simply and inexpensively as an integer that stores the value P. ω_e must wait for each of them to finish with p before it operates on it; this is achieved on line 3. As discussed in Section 2, an elemental operation can occur *to the right of* a bulk operation which has not yet linearized. Suppose ω_e on key K in partition p sees linearizedID $= P$ on line 1, telling us that ω_P is the last bulk operation that linearized before ω_e. Then suppose that on line 3, p.lastVisitorId $= L$, telling us that ω_L is the last bulk operation that visited p, and therefore it and all its predecessor must be *linearized* by ω_e before it returns, if they are not already. Note that while we know $\forall i \in \mathbb{N}$ where $i \leq P$, ω_i has linearized, we do not know if it has finished; it may have linearized itself upon completion, or been linearized by some other elemental operation $\omega_{e'}$, either directly or transitively by linearizing a later bulk operation.

There are three cases to consider. (1) When $L > P$, since ω_L has visited p, and $\omega_P \longrightarrow \omega_L$, we can conclude ω_P has also visited the partition. That is, ω_e is *left of* ω_L, and transitively ω_P, and so it can proceed. However, because ω_e by definition must linearize before it returns, and because $\omega_L \longrightarrow \omega_e$, it also must linearize ω_L and all its predecessors before it returns (line 8). (2) When $L = P$, since ω_L and ω_P are two names for the same operation, ω_P has already visited p, and line 8 will be a no-op, as it has already linearized. (3) When $L < P$, we have $\omega_L \longrightarrow \omega_P \longrightarrow \omega_e$. Thus p is to the *right of* ω_P. Since ω_P is linearized, but has not accessed P yet, ω_e waits on line 3, until L reaches or exceeds P.

It is important to note that line 1 sets the earliest point when ω_e may happen, but due to concurrent bulk operations, ω_e may happen much later without violating linearizability: concurrent operations can correctly complete in any order. Note, too, that the order of elemental operations with respect to each other is simply the order in which they reach line 5, and the order of bulk operations is the order in which they complete line 9.

When there are no bulk operations, AO should introduce little overhead: elementals perform a single extra global memory read of an unchanging value (line 1) and a single extra test (line 3); the store on line 8 will be a no-op. Similarly, concurrent bulk operations only access global metadata twice (lines 9 and 16), and only wait for each other when they conflict on the same partition. This is a significant advantage relative to 2PL: in a bulk-only workload, it is possible to achieve concurrency equal to the number of partitions.

However, AO will *always* order two bulk operations, even when they should be able to correctly run in parallel, such as when they are both read-only or operate on disjoint ranges. Thus, one operation may be unnecessarily ordered with and forced to wait on another operation.

Related is the problem that there is no efficient way to perform range operations in AO—a thread performing a range operation ω_i will have to go from the very start of the map to the very end, and increase the lastVisitorID of every single partition to i, whether the partition contains elements in its range or not, to unblock any operations that may be waiting for that partition's lastVisitorID to reach i. This particular pathology is unique to AO—the other two algorithms presented later in the paper do not suffer from it. Finally, it is not always best to order all bulk operations by their start time. AO solves the three-range problem discussed in Section 2 in a rather unsatsifying way: because ω_3 starts after ω_1, it is unavoidably ordered after it. Thus, ω_3 will have to wait until ω_1 is finished with the partition containing e_1 before it can start, delaying it unnecessarily.

3.2 Dynamic Ordering

Our second algorithm, Dynamic Ordering (DO), uses significantly more *global* metadata than AO. Since many of AO's shortcomings are caused by a lack of information about whether bulk operations require ordering, DO is designed to utilize its additional metadata to order bulk operations more carefully. The metadata for DO appears in Listing 3. Note that DO does not require any metadata in each partition. For consistency, we represent this by including per-partition spinlocks (DOPartitionLock).

BulkOpMetadata holds the metadata for a single bulk operation ω. startKey and endKey specify the range of ω. If ω is a foreach operation, these are $\perp$ and $\top$ respectively. If we know ahead of time that ω will not modify any element in the map, then readOnly is set.

Listing 3: Data types used by Dynamic Ordering

```
Type DOGlobalMetadata
  bulkOps      :  List⟨Set⟨BulkOpMetadata⟩⟩
Type BulkOpMetadata
  const startKey    :  Key
  const endKey      :  Key
  const readOnly    :  Boolean
  lastKey           :  Key
  linearized        :  Boolean
Type DOPartitionLock
  isLocked     :  Atomic⟨Boolean⟩
```

Listing 4: Operation coordination for Dynamic Ordering

```
   function $elemental$(k)
       // get all linearized ops that must precede this elemental
1      $\Omega_P \leftarrow$ atomic bulkOps.$queryPred$(k)
2      p $\leftarrow$ $getPartition$(k)
       // wait for preceding, active bulk ops to pass k
3      foreach $\omega_P \in \Omega_P$ do atomic $\omega_P.waitUntilPast$(k)
4      p.$acquireLock$()
5      . . . // perform elemental operation
6      p.$releaseLock$()
       // find the set of bulk ops linearized by this elemental
7      $\Omega_L \leftarrow$ atomic bulkOps.$queryPast$(k)
8      atomic foreach $\omega_L \in \Omega_L$ do $\omega_L.markLinearized$()
   function $bulk$(start, end, ro)
       // establish order for $\omega$
9      atomic $\omega \leftarrow$ bulkOps.$insEarly$(start, end, ro, $\bot$, false)
10     foreach p $\in$ partitions do
11         lockedP $\leftarrow$ false
12         while $\neg$lockedP do
13             atomic
                   // Check if $\omega$ can access p
14                 if $\neg$bulkOps.$anyBlockers$($\omega$, p) then
15                     p.$acquireLock$()
16                     $\omega$.lastKey $\leftarrow$ p.$maxKey$()
17                     lockedP $\leftarrow$ true

18         . . . // perform operations on p
19         p.$releaseLock$()
       // recursively mark $\omega$ and preceding bulk ops linearized
20     atomic bulkOps.$markLinearized$($\omega$)
21     bulkOps.$remove$($\omega$)
```

Otherwise, ω is *mutating* and readOnly is cleared. The linearized boolean is cleared at the start, and set once ω has linearized. Finally, lastKey represents the last key that ω can be certain it has finished with. startKey, endKey, and readOnly are immutable. The use of the lastKey field represents a significant difference from AO—the progress of a bulk operation through the keys of the map is tracked by lastKey rather than some visitorID stored in the partitions.

The global metadata consists of bulkOps, a list of sets of BulkOpMetadata objects. Bulk operations in the same set are *unordered* with respect to each other. Bulk operations ω_i and ω_j only require ordering if their ranges overlap and at least one of them is mutating. If two operations are ordered, then the operation ordered earlier must be placed *closer* to the head of the list than the other. In this way, bulkOps behaves somewhat like a queue.

Although an improvement over AO, representing the ordering as a list of sets still enforces some arbitrary ordering—the ordering between bulk operations is, theoretically, a directed acyclic graph rather than a list of sets. We implemented a true DAG as well, but its performance was unacceptable. Using a list of sets leads to a few edge cases being missed, but significantly improves performance.

To start a new bulk operation ω, thread t invokes the function *bulk*() in Listing 4, and passes in parameters that indicate the range of the op and if it is read-only. Using these parameters, it initializes a BulkOpMetadata object for ω and then invokes *insEarly*() to find the appropriate place for it in bulkOps (line 9). *insEarly*() will start from the tail of bulkOps and work its way towards the head. For each set it examines, it will iterate through the bulk operations in the set and sort them into three types, depending on their relationship to ω. The first type, Ω_U, is operations that ought to be *unordered* with respect to ω; they have no need to be ordered with respect to ω unless other bulk operations force a transitive ordering between them. The second type, Ω_B, is for operations which must be ordered *before* ω, either because they have already linearized or have already processed elements in ω's range. The last type, Ω_A, is for operations which must have some ordering with ω, but for which either ordering—ω first or ω second—is still valid. DO will always choose to order these after *after* ω.

insEarly() tries to place ω as far forward in bulkOps as it can, ahead of any sets that contain no operations of type Ω_B, and potentially splitting an existing set if it is a mix of multiple types. If the tail set consists only of operations of Ω_B, or bulkOps is an empty list, then ω becomes the tail. To illustrate how *insEarly*() works, consider the three-range problem described in Section 2. ω_1 begins, and, seeing bulkOps empty, initializes it with ω_1. When ω_2 begins, it categorizes ω_1 as Ω_U, as they operate on disjoint ranges. It then places itself into the same set, to represent that they are unordered. When ω_3 begins, it categorizes ω_2 as Ω_B, as it has already operated on e_3. Then, it categorizes ω_1 as Ω_A. It cannot categorize it as Ω_U, as their ranges overlap and they are both mutating. However, even though they are not unordered, either ordering is still valid, as ω_1 has not yet linearized nor locked the partition containing e_1. $\omega_3 \rightarrow \omega_1$ allows concurrency between the two operations, and so DO chooses this ordering by categorizing ω_3 as Ω_A. Thus, ω_3 splits the set containing ω_1 and ω_2 into two and inserts itself in a new set between them, resulting in $\omega_2 \rightarrow \omega_3 \rightarrow \omega_1$.

Once ω places itself into the appropriate place in bulkOps, it can then operate on each partition in its range, in order (line 10). To lock partition p, ω must ensure no other operation blocks it by invoking *anyBlockers*() (line 14). This method checks every operation ω_P in every set ahead of ω in the list, and returns true if even one of them blocks ω from accessing p. A bulk operation ω_P blocks ω from operating on p if ω_P is in a set ahead of ω in bulkOps, there exists a key K in p which is in both operations' ranges and which ω_P has not yet accessed, and at least one of ω and ω_P is mutating. If there are none, then ω will acquire the lock on p, and update its lastKey field to inform other operations of its progress (lines 15-16). All of this must be done in an atomic block to ensure no other operation inserts itself into bulkOps ahead of ω in the meantime (line 13). As the lock will not be taken until *anyBlockers*() returns false, this is tried repeatedly in a loop until it is successful (line 12). Once the lock is successfully taken, ω can operate on the elements in the partition and release the lock (lines 18-19). Once ω has processed all partitions in its range, it will atomically and recursively mark ω and all of its predecessors linearized if they are not already, and finally it will remove itself from bulkOps (lines 20-21).

When t_e starts elemental operation ω_e on key K, it atomically searches in bulkOps and saves the set Ω_P of bulk operations that

Listing 5: Data types used by Localized Ordering

```
Type OLPartitionLock
  started     :  AtomicQueue⟨ThreadID⟩
  completed   :  AtomicQueue⟨ThreadID⟩
```

Listing 6: Operation coordination for Localized Ordering

```
function $elemental$(k)
1      p ← $getPartition$(k)
       // Order this operation within p, then await turn
2      p.started.$enqueue$(self)
3      while p.started.$head$() ≠ self do wait
4      . . . // perform elemental operation
       // Allow subsequent ops, but obey ordered departure
5      p.completed.$enqueue$(self)
6      p.started.$dequeue$()
7      while p.completed.$head$() ≠ self do wait
8      p.completed.$dequeue$()
function $bulk$(startKey, endKey)
9      $prev$ ← nil
10     foreach p ∈ partitions(startKey, endKey) do
           // Order this operation within p
11         p.started.$enqueue$(self)
           // Allow subsequent ops on prev
12         if $prev \neq nil$ then $prev$.started.$dequeue$()
13         $prev$ ← p
           // Wait for permission in p
14         while p.started.$head$() ≠ self do wait
15         . . . // perform operations on p
           // Prevent subsequent ops from finishing on p
16         p.completed.$enqueue$(self)
17     if $prev \neq nil$ then $prev$.started.$dequeue$()
       // Let other ops finish on partitions this bulk visited
18     foreach p ∈ partitions(startKey, endKey) do
19         while p.completed.$head$() ≠ self do wait
20         p.completed.$dequeue$()
```

block it (line 1). Ongoing bulk operation ω_P blocks ω_e if ω_P has linearized, K is in ω_P's range, ω_P has not yet processed K, and either ω_e or ω_P is mutating. t_e then invokes *waitUntilPast*() on each, as above (line 3), before proceeding to lock the partition, perform ω_e, and unlock it (lines 4-6). When ω_e completes, it must ensure that any bulk operations that processed K are linearized (lines 7-8). For simplicity, all of the other bulk operations in that set are linearized as well. Transitivity applies: all predecessors of an operation that ω_e linearizes must be linearized as well.

3.3 Localized Ordering

Our third algorithm, Localized Ordering (LO), is built around the following observation: 2PL achieves linearizability by delaying operations from *starting*, but the sufficient condition is to delay them from *returning*. Our LO algorithm relaxes 2PL by adding more per-partition metadata, without the need for any global state.

Listings 5 and 6 show the metadata and code for LO. To coordinate access to partitions, we make use of two queues, started and completed. We assume that each is thread-safe. These two queues combine ideas from queue locks [10] and ticket locks [7].

To understand LO, consider a simpler algorithm, with one queue per partition. In that algorithm, an elemental operation ω_e would acquire a lock on partition p by enqueueing its thread ID, and then waiting for its entry to reach the head of the queue. To unlock, ω_e would dequeue itself. A bulk operation ω_B would enqueue itself into every partition before dequeueing itself from any partition, hence ensuring two phases for its lock acquisitions and releases. The queue effectively serves as a spinlock, and the correctness of the algorithm follows directly from the correctness of 2PL.

Suppose an elemental operation ω_e wishes to operate on a partition p_i after ω_B finishes with it. We have $\omega_B \rightarrow \omega_e$, but because ω_B has finished with p_i, it can correctly do so immediately. However, ω_e cannot return from operating on p_i, because its thread might then go on to execute some operation $\omega_{e'}$ to a partition p_j that is in the range of ω_B, but which ω_B has not yet reached. Correctness would be violated because $\omega_{e'}$ is *to the right of* the in-progress ω_B.

The same problem arises if bulk operation $\omega_{B'}$ attempts to access p_i, and p_i is the last partition in $\omega_{B'}$'s range. However, bulk operations introduce a second issue. Suppose that ω_B has completed its operation on p_i, and is about to operate on p_{i+1}. Let $\omega_{B'}$ operate on p_i. Clearly $\omega_B \rightarrow \omega_{B'}$ in partition p_i. We must therefore ensure that $\omega_B \rightarrow \omega_{B'}$ in partition p_{i+1}.

The role of the second queue is to increase concurrency in both of these cases, without sacrificing correctness. To acquire permission to operate on partition p_i, operation ω must enqueue in p_i.started and wait until it reaches the head position. However, it now enqueues in p_i.completed before dequeueing from p_i.started. This has the effect of relinquishing permission to access the data in p_i. ω then waits until it is the head of p_i.completed, at which point it can dequeue and return. To ensure consistent ordering of ω_B and $\omega_{B'}$ on partitions p_i and p_{i+1}, ω_B does not dequeue from p_i.started until it has enqueued in p_{i+1}.started (lines 9, 11-12, and 17).

For workloads consisting only of elemental operations, these changes are uninteresting: ω_j waits on line 7 only if $\omega_i \rightarrow \omega_j$ and ω_i delays between line 6 and line 8. However, in the presence of bulk operations, it achieves three properties. First, when ω_e is to the right of ω_B, it can order before ω_B. Second, when ω_e is to the left of ω_B, it can operate on its partition, but must wait before returning. Third, when $\omega_B \rightarrow \omega_{B'}$, $\omega_{B'}$ can access its partitions before ω_B completes: for each partition p_i, it will access p_i after ω_B, and it will also return only after ω_B has completed its operation on every p_i accessed by both operations.

4 IMPLEMENTATION CHALLENGES

It is simplest to think of our algorithms in the context of a closed-addressing hash table with a fixed number of buckets. However, real data structures introduce important considerations, especially with regard to partitioning. We consider two data structures in our evaluation, which introduce real-world challenges. The first is the Interlocked Hash Table (IHT) [9], an unordered map implemented as a $log(log(n))$-depth tree of nodes with increased arity. The second is a variant of the concurrent unrolled skip list [12].

4.1 Unordered Maps

In the IHT, data is stored in leaf nodes, called Element Lists, or ELists. These immediately serve as the partitions. ELists can split, but they never merge. Additionally, since there is no inherent ordering among keys, it can only support foreach operations, not range.

When an EList becomes full and splits, it becomes an internal node (a Pointer List, or PList) with twice as many children as its parent. In the AO algorithm, the last ω_F to visit the EList is stored in the AOPartitionLock. This state must be replicated to all of the children of the new PList. The LO algorithm is more complicated: The LOPartitionLock can hold an arbitrary number of (elemental and bulk) operations that have not started operating on the EList yet, and an arbitrary number of operations that are awaiting order on the EList. Our solution is to remove elemental operations from the started queue, and then replicate both queues into every child of the new PList. A bulk operation in started detects the change from EList to PList and recurses; an elemental operation in started detects the change and restarts; and both types of operations in completed recurse into the children of the PList and wait to remove themselves from every descendent. This corner case is rare for large unordered maps. Lastly, in the DO algorithm, no state is stored in partitions, so splitting and merging is trivial.

4.2 Ordered Maps

In our chunked skip list, partitions can split and merge, and there are range operations in addition to foreach. The splitting and merging occurs in order to keep partitions within an acceptable range of the target partition size, a tunable parameter discussed further in Section 5, even if the skip list is not afforded the luxury of a uniform key distribution.

Our solution to splitting is the same as in the IHT. With regard to merging, AO does not present any challenges: when a node must be merged, we merge it with its predecessor. Since bulk operations do not revisit the node to release locks, and since we lock partitions in a hand-over-hand manner, there is no further bookkeeping.

When merging in the LO algorithm, our solution is to employ lazy merging. That is, when partition P_j should be merged into its predecessor P_i, we delay the merge until the next time an operation accesses P_i. This simplifies the four merge cases substantially. (1) If elemental operation ω_e is in the started queue of P_j, we filter ω_e from the queue when we merge P_j into P_i, and we ensure ω_e detects the merge and restarts. (2) If ω_e is in the completed queue of P_j, we merge P_j but leave it as an empty partition; it is reclaimed once the queues become empty. (3) A bulk operation ω_B never enqueues itself into P_j when P_j is ready to merge: the merge is intiated by an operation on P_i, and at the time when ω_B enqueued itself in P_j.started, it held the lock on P_i, and determined that a merge was not necessary. (4) When ω_B is in P_j.completed and P_j should be merged, we follow the same approach as in case 2.

5 EVALUATION

In this section, we measure the effectiveness of the AO, DO, and LO algorithms on improving the performance of bulk operations. We explore four questions:

(1) Does tuning a data structure configuration for efficient bulk operations harm the performance of elemental operations?
(2) Do the new algorithms hurt elemental performance in the absence of bulk operations?
(3) Are the new algorithms effective in improving the scalability of bulk operations?
(4) Do the algorithms provide robust performance when there is a mixture of bulk and elemental operations?

To evaluate these questions, we conducted a set of microbenchmark experiments. We consider two baselines. The first is 2PL. We only present 2PL with lazy acquisition: it performed better for bulk operations than eager acquisition, and also is more similar to our LO algorithm. The second baseline, NL, does not provide linearizable bulk operations: like 2PL, it uses hand-over-hand locking of partitions. However, it does not hold locks on partitions that it has finished processing, and thus can observe the anomalies in Section 2. NL serves as an upper bound on performance.

Table 1: Best configurations for IHT and skip list

Concurrency Algorithm	Elemental Tuning	Foreach Tuning	Compromise Tuning
Two-Phase Locking	32	4096	128
Aggressive Ordering	32	4096	128
Dynamic Ordering	32	4096	512
Localized Ordering	32	4096	128
Non-Linearizable	32	1024	128

(a) Interlocked Hash Table

Concurrency Algorithm	Elemental Tuning	Foreach Tuning	Compromise Tuning
Two-Phase Locking	32	2048	256
Aggressive Ordering	32	2048	512
Dynamic Ordering	32	8192	1024
Localized Ordering	32	2048	512
Non-Linearizable	32	2048	1024

(b) Unrolled Skip List

We evaluate the IHT on three workloads. In *elemental* experiments, we pre-fill the map with half of the keys in a 20-bit range, then perform a mix of 80% contains, 10% insert, and 10% remove operations, using keys drawn from a uniform distribution. In the *foreach* workload, the map is prefilled in the same way, but threads perform a mix of 80% read-only and 20% mutating foreach. In the *mixed* workloads, either one or two threads executes the *foreach* workload while all others execute the *elemental* workload.

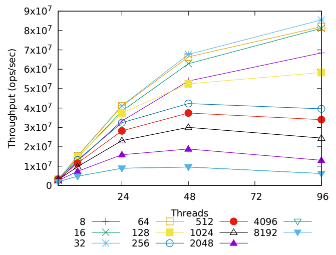
We evaluate the unrolled skip list using the same *elemental* and *foreach* workload mixes. We also conducted *range-L* workloads, where all threads execute range operations, starting from a random key $k \in \{0 \ldots 2^{20} - L\}$ and continuing to key $k + L$. We present results for a small and large value of L; 80% of range operations are read-only. The skip list *mixed-foreach* workloads are configured the same as IHT *foreach*; *mixed-range-L* replaces *foreach* with a *range-L*.

We measured performance on a machine running Ubuntu 18.04.4 on two 2.1GHz Intel Xeon Platinum 8160 processors with 192GB of RAM. Each processor has 24 cores / 48 threads, for a total of 96 hardware threads. All code was written in C++, and compiled with g++ 7.3.0. Each experiment was run for 5 seconds, and we report the average of five trials. We did not observe significant variance.

5.1 Tuning

The IHT EList size and root PList size are configurable, as is the size of skiplist data layer chunks. An important consideration is how to tune these settings. If partitions are too large, then concurrency suffers for both elemental and bulk operations. If partitions are too small, then latency suffers for bulk operations. To understand the impact of partition size, we varied it per workload per algorithm. The values we considered in our tuning were powers of 2 between 8 and 8192. Table 1 reports the best partition sizes for elemental operations and for bulk operations. Tuning was performed based on peak performance, not 96-thread performance.

Small partition sizes work best for elemental operations, with 32 consistently providing the best performance in both the skip


(a) Elemental-Only

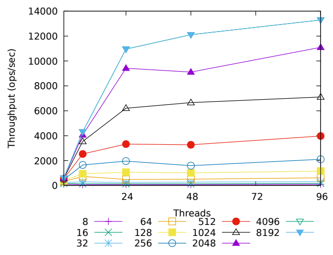

(b) Foreach-Only

Figure 3: Impact of tuning on skip list, DO algorithm

list and IHT. For bulk operations, values ≥ 1024 always performed best. Figure 3 presents a snapshot of the performance impact of this configuration on the skip list with the DO algorithm, for elemental and foreach workloads. For elemental workloads, a poor choice can degrade performance by almost an order of magnitude. In the foreach experiments, the impact is 2-3 orders of magnitude. While the trends are unsurprising, the magnitude necessitated that we choose a compromise configuration for each data structure and concurrency algorithm.

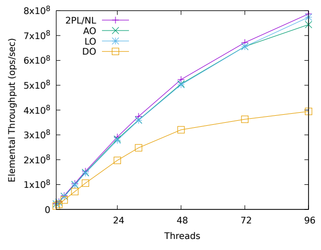
Typically, compromise tunings offered around half of the performance of the ideal tuning for each workload. Our compromise tunings ended up being scattered between 128 and 1024 elements per partition. In Figure 3, we see that a tuning value of 1024 has middling performance for both workloads. 512 is another option which has better performance for elementals, but 1024's lead in bulk operations is much wider than 512's lead for elementals, so 1024 was chosen. Table 1 reports these compromise partition sizes.[1]

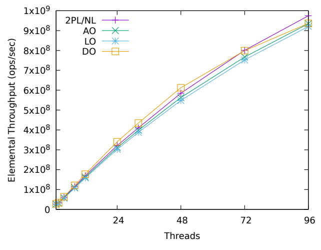
In the IHT, it is also possible to adjust the size of the root PList. We calculated the value that would give us the desired number of buckets at the desired depth, based on the chosen EList size. For example, with 2^{19} elements in total, a target of 256 elements per EList, and desired depth of 2, then that means we should have 2^{11} ELists in total. Choosing a root PList size of 32 yields exactly that many buckets in PLists in the second layer of the IHT, and so that is the value that is used. Our tuning experiments showed that having two layers of PLists was ideal for all EList sizes of 32 or more.

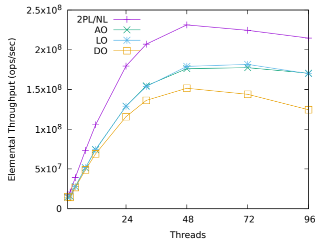
5.2 Elemental-Only Workloads

Our first set of experiments considers the overhead our algorithms introduce in workloads that do not perform bulk operations. Figure 4a shows elemental throughput on an IHT *elemental* workload with compromise tuning. The overhead of most of our algorithms is low, with AO only slightly behind 2PL, and LO following closely behind. DO suffers, but that is due entirely to the compromise tuning—Figure 4b shows the same experiment but with elemental tuning, and DO is able to keep pace with the other algorithms.

Figure 4c and Figure 4d show the same results for the unrolled skip list. Again, the gap between our algorithms and 2PL is mostly due to compromise tuning. When elemental tuning is used, the gap narrows considerably. Furthermore, the shape of the curves is similar, suggesting that degraded elemental performance is not due


(a) IHT, compromise tuning


(b) IHT, elemental tuning


(c) Skip list, compromise tuning

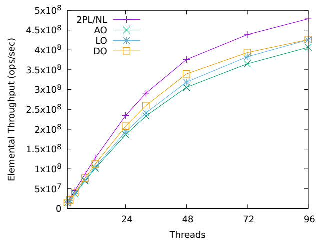
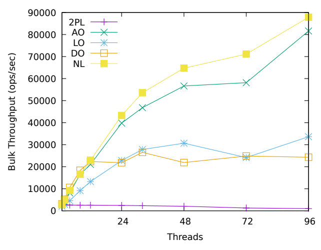
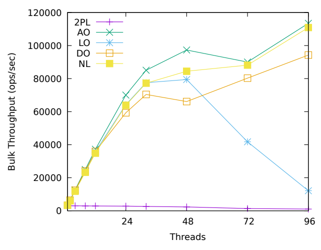
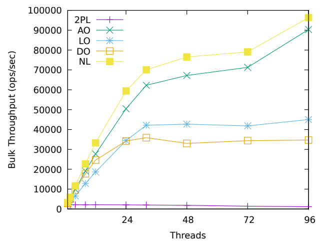

(d) Skip list, elemental tuning

Figure 4: Elemental-only workload throughput


(a) IHT, compromise tuning


(b) IHT, foreach tuning


(c) Skip list, compromise tuning

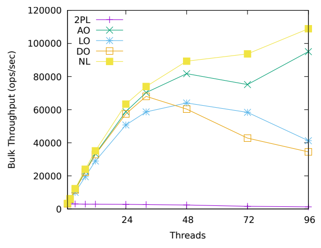

(d) Skip list, foreach tuning

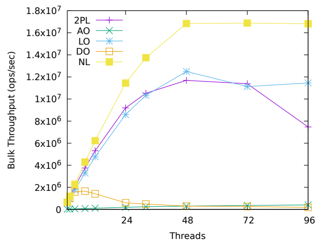
Figure 5: Foreach-only workload throughput

to new scalability bottlenecks, but to increased interaction with metadata. This suggests that elemental tuning is probably best for workloads where bulk operations are rare.

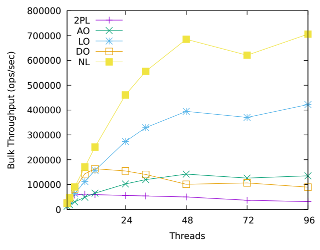
5.3 Bulk-Only Workloads

Figure 5a shows *foreach* performance for an IHT with compromise tuning. As expected, 2PL does not scale at all. DO and LO significantly outperform 2PL, but their scaling is limited by extensive interaction with metadata (DO) or contention for per-partition queues (LO). AO performance is on par with NL, despite providing

[1]Note to reviewers: we plan to release the full suite of results across all tuning sizes as a companion technical report.



(a) 2^{12} keys



(b) 2^{17} keys

Figure 6: Range-only workload, compromise tuning

much stronger correctness guarantees. In fact, the compromise tuning is the entire reason for the gap: with ideal tuning (Figure 5b), AO matches or surpasses NL, and DO and LO performance improve significantly. Figures 5c and 5d show similar results for the unrolled skip list: our algorithms scale while providing linearizability, and AO, which is best suited for this workload, is on par with the non-linearizable baseline (NL).

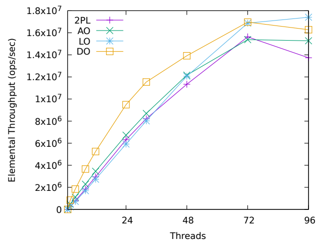
As the skip list is an ordered data structure, we also present *range-L* workloads with $L = 2^{12}$ and $L = 2^{17}$, chosen to illustrate small and large ranges, respectively (Figure 6). Despite its inability to scale at all in a *foreach* workload, 2PL scales well for the smaller L value, since threads typically perform non-overlapping operations. When there is not much overlap, false waiting seems to not be a significant issue for 2PL. LO manages to keep pace with 2PL, and even outperform it significantly at 96 threads. In contrast, AO performs poorly: it does not have enough metadata to take advantage of the fact that ranges rarely overlap, instead treating each range operation as a foreach. DO also does not perform well, despite being designed to address this shortcoming in AO. As the number of threads increases, so does contention on the global data structure, and thus DO only scales up to 8 threads.

With a larger L value, the range operations behave more like foreach operations. When ranges cover 2^{17} keys (thus operating on 2^{16} elements on average), 2PL performs worst, AO and DO provide a measurable advantage, and LO achieves about half the performance of the non-linearizable NL algorithm.

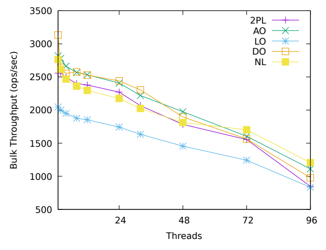
5.4 Mixed Workloads

Lastly, we investigate the performance of our algorithms on mixed workloads. For all experiments in this section, the elemental performance of NL was omitted: by sacrificing linearizability, it can achieve performance more than an order of magnitude higher than 2PL or our algorithms, which impacts the readability of the figures. Note, too, that there are a fixed number of foreach/range threads, with all additional threads performing elemental operations. Thus elemental throughput is able to scale as the thread count increases, but bulk throughput can only decline, as the threads performing bulk operations will contend with more and more elemental threads.

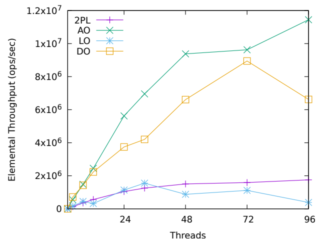
Figures 7a and 7b show elemental and bulk throughput of a mixed workload with 1 thread performing foreach operations on an IHT with compromise tuning. In this workload, the foreach thread is always able to make progress, but elemental operations have a 50% chance of having to wait on the foreach. The algorithms all



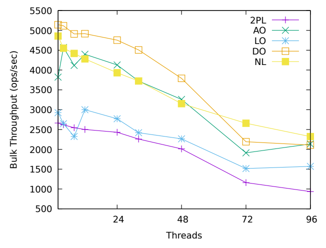
(a) Elemental throughput, 1 foreach thread



(b) Bulk throughput, 1 foreach thread



(c) Elemental throughput, 2 foreach threads



(d) Bulk throughput, 2 foreach threads

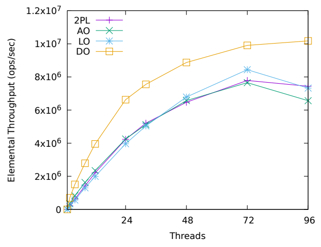
Figure 7: *Mixed* workloads, IHT, compromise tuning

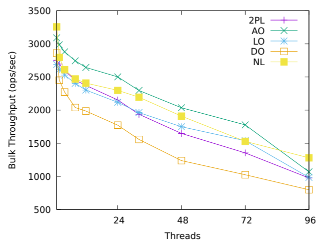
allow the elemental operations to make progress, but elemental throughput is much lower than in previous experiments. LO foreach performance is the worst: it experiences contention when releasing locks, due to the presence of other operations in the queues, and thus *unlocking* becomes a more significant overhead than in 2PL. In contrast, DO performs very well, because the sole foreach thread does not experience contention on the global data structure.

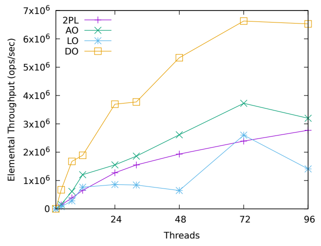
Figures 7c and 7d show a similar experiment, but with two threads performing foreach operations. In this workload, the two foreach threads effectively have the entire data structure locked at all times in 2PL and LO, which prevents elemental operations from making progress (or, in the case of LO, returning after completing their operation). Thus we see an extreme drop in elemental performance for these algorithms, whereas AO and DO perform well. AO and DO also provide the best performance for the bulk threads, which are able to execute concurrently with each other.

Figure 8 shows the results of the same experiment on the unrolled skip list. The trend of DO delivering the best performance to elemental threads remains, but this comes at a decrease in foreach performance: DO naturally gives more priority to elementals than any of the other algorithms, and the impact increases when a second foreach thread is added. Again, we see poor performance for 2PL, with the relative merit of LO increasing on account of its ability to exploit parallelism among the foreach threads.

Lastly, Figure 9 presents the most complex configuration: a mixed workload consisting of two threads performing range operations of length 2^{17}, on a skip list with compromise tuning. Again, DO delivers the best performance for elemental operations, with competitive foreach performance. While the pathological behavior of LO disappears once bulk operations do not conflict with *every* elemental, AO now experiences its pathology: since it does not track ranges, every bulk operation behaves like a foreach.


(a) **Elemental throughput, 1 foreach thread**


(b) **Bulk throughput, 1 foreach thread**


(c) **Elemental throughput, 2 foreach threads**

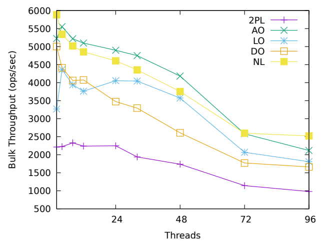
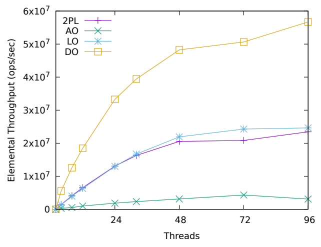

(d) **Bulk throughput, 2 foreach threads**

Figure 8: *Mixed* workloads, skip list, compromise tuning


(a) Elemental throughput

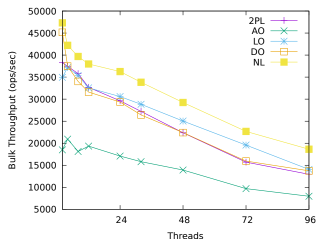

(b) Bulk throughput

Figure 9: *Mixed* workload, 2 range threads, 2^{17} range size, skip list, compromise tuning

6 CONCLUSIONS AND FUTURE WORK

In this paper, we introduced three algorithms that enable scalable, linearizable bulk operations on concurrent maps, with better performance than two-phase locking. The novel idea is to make the linearization of a bulk operation into a state that is visible to concurrent operations. This allows operations to know when they can overtake each other without violating correctness. This, in turn, reduces false waiting. Our algorithms vary in how much metadata they store, and whether they store it globally or in the partitions of the data structure itself.

Our experiments explore the complex space of data structure tuning, and show that tuning for elemental operations can be at odds with tuning for bulk operations. We then show that our algorithms do not introduce significant overhead for elemental operations, but do enable significant scalability for bulk operations. In mixed workloads, at least one of our algorithms always outperformed two-phase locking. However, there was no single best algorithm for all workloads.

As future work, we believe it will be important to devise heuristic strategies, perhaps based on machine learning, that can choose the best approach on a workload-by-workload basis, or even dynamically switch between algorithms and alter tuning parameters on-the-fly. The algorithmic support for dynamically switching among strategies appears tractable, since the per-partition and global metadata required by each strategy can be combined with little added space overhead. Learning how to make good selections at run time will, at the very least, use metrics like frequency of bulk operations, and range size. Our experiments show separation for the algorithms based on these metrics, and thus we are cautiously optimisic that a small set of relatively straightforward features will provide a satisfactory workload characterization. While we believe that dynamically choosing algorithms will be straightforward, altering the partition size may be more difficult: it could require an expensive restructuring of the entire data structure if performed eagerly.

ACKNOWLEDGMENTS

We thank Victor Luchangco and our reviewers for many helpful recommendations. This work was supported in part by the NSF under Grant CNS-1814974. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[1] Yehuda Afek, Danny Dolev, Hagit Attiya, EliGafni, Michael Merritt, and Nir Shavit. 1990. Atomic snapshots of shared memory. In *Proceedings of the 9th ACM symposium on Principles of Distributed Computing*. Quebec, Canada.
[2] Maya Arbel-Raviv and Trevor Brown. 2018. Harnessing Epoch-Based Reclamation for Efficient Range Queries. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Vienna, Austria.
[3] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control-Theory and Algorithms. *ACM Transactions on Database Systems* 8, 4 (Dec. 1983), 465–483.
[4] Anastasia Braginsky and Erez Petrank. 2012. A Lock-Free B+tree. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*. Pittsburgh, PA.
[5] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM* 19, 11 (1976), 624–633.
[6] Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. 2018. Persistent Non-Blocking Binary Search Trees Supporting Wait-Free Range Queries. In *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures*. Phoenix, AZ.
[7] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann.
[8] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492.
[9] Louis Jenkins, Tingzhe Zhou, and Michael Spear. 2017. Redesigning Go's Built-In Map to Support Concurrent Operations. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques*. Portland, OR.
[10] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems* 9, 1 (1991).
[11] Maged Michael. 2002. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*. Winnipeg, Manitoba, Canada.
[12] Kenneth Platz, Neeraj Mittal, and S. Venkatesan. 2019. Concurrent Unrolled Skiplist. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems*. Dallas, TX.
[13] Aleksandar Prokopec, Nathan Bronson, Phil Bagwell, and Martin Odersky. 2012. Concurrent Tries with Efficient Non-Blocking Snapshots. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming*.