A Wait-Free Universal Construct for Large Objects

ANDREIA CORREIA, University of Neuchatel
PEDRO RAMALHETE, Cisco Systems
PASCAL FELBER, University of Neuchatel

Concurrency has been a subject of study for more than 50 years. Still, many developers struggle to adapt their sequential code to be accessed concurrently. This need has pushed for generic solutions and specific concurrent data structures.

Wait-free universal constructs are attractive as they can turn a sequential implementation of any object into an equivalent, yet concurrent and wait-free, implementation. While highly relevant from a research perspective, these techniques are of limited practical use when the underlying object or data structure is sizable. The copy operation can consume much of the CPU's resources and significantly degrade performance.

To overcome this limitation, we have designed CX, a multi-instance-based wait-free universal construct that substantially reduces the amount of copy operations. The construct maintains a bounded number of instances of the object that can potentially be brought up to date. We applied CX to several sequential implementations of data structures, including STL implementations, and compared them with existing wait-free constructs. Our evaluation shows that CX performs significantly better in most experiments, and can even rival with hand-written lock-free and wait-free data structures, simultaneously providing wait-free progress, safe memory reclamation and high reader scalability.

1 INTRODUCTION

Many synchronization primitives have been proposed in the literature for providing concurrent access to shared data, with the two most common being mutual exclusion locks and reader-writer locks. Both of these primitives provide blocking progress, with some mutual exclusion algorithms like the ticket lock [18] or CLH lock [16] going so far as being starvation free. Even today, the usage of locks is still of great relevance because of their generality and ease of use, despite being prone to various issues such as *priority inversion*, *convoying* or *deadlock* [13]. Yet, their main drawback comes from their lack of scalability and suboptimal use of the processing capacity of multi-core systems, except for lock-based techniques with disjoint access. This has led researchers to extensively explore alternatives to support non-blocking data structures, either using ad-hoc algorithms or generic approaches.

Generic constructs are attractive from a theoretical perspective, but so far they have been largely neglected by practitioners because of their lack of efficiency when compared to dedicated algorithms tailored for a specific data structure. The search for a generic non-blocking solution that is also practical has resulted in significant developments over the last decades, notably in the fields of wait-free universal constructs (UCs) and hardware and software transactional memory (HTM and STM).

A wait-free universal construct is a generic mechanism meant to provide concurrent wait-free access to a sequential implementation of an object or group of objects, *e.g.*, a data structure. In other words, it takes a *sequential specification* of an object and provides a concurrent implementation with wait-free progress [11]. It supports an operation, called `applyOp()`, which takes as a parameter the sequential implementation of any operation on the object, and simulates its execution in a concurrent environment. Most UCs can be adapted to provide an API that distinguishes between read-only operations and mutative operations on the object, which henceforth will be referred to as `applyRead()` and `applyUpdate()` respectively.

Software transactional memory, on the other hand, has transactional semantics, allowing the user to make an operation or group of operations seem *atomic* and providing *serializability* between transactions [13].

Authors' addresses: Andreia Correia, University of Neuchatel, andreia.veiga@unine.ch; Pedro Ramalhete, Cisco Systems, pramalhe@gmail.com; Pascal Felber, University of Neuchatel, pascal.felber@unine.ch.

2019. XXXX-XXXX/2019/6-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

STMs and UCs present two separate approaches to developers when it comes to dealing with concurrent code. Both approaches allow the end user to reason about the code as if it were sequential. STMs *instrument* the loads and stores on the sequential implementation. STMs may also require type annotation, function annotation or replacement of allocation/deallocation calls with equivalent methods provided by the STM. Finally, to the best of our knowledge, there is currently no STM with wait-free progress. A recent development, named RomulusLR [7], provides wait-free progress for read-only operations and blocking starvation-free updates.

We observe that the UC literature can be classified into two groups of algorithms, UCs that do not require instrumentation, and UCs that do. *Non-instrumenting* UCs require no annotation of the sequential implementation, allowing the developer to *wrap* the underlying object and creating an equivalent object with concurrent access for all of its methods. *Instrumenting* UCs require the developer to annotate and modify the sequential implementation, similar to what must be done for an STM. This annotation implies effort from the developer and is prone to errors. In addition, the fact that annotation is required at all, makes it difficult or unfeasible to use legacy code or data structures provided by pre-compiled libraries (*e.g.*, `std::set` and `std::map`) because it would require modifying the library's source code.

In this paper, we focus on non-instrumenting UCs with the goal of addressing their main limitations in terms of performance and usability, which made them so far impractical for real-world applications. We introduce CX, a non-instrumenting UC with linearizable operations and wait-free progress that does not require any annotation of the underlying sequential implementation. CX provides fast and scalable read-only operations by exploiting their *disjoint access parallel* [15] nature.

In short, with CX we make the following contributions: *(i)* We introduce the first practical wait-free UC, written in portable C++, with integrated wait-free memory reclamation and high scalability for read-mostly workloads. *(ii)* We address wait-free memory reclamation with a flexible scheme that combines reference counting with hazard pointers *(iii)* We present the first portable implementation of the PSim UC [10], with integrated wait-free memory reclamation, and added high scalability for read-only operations.

The rest of the paper is organized as follows. We first discuss related work in §2. We then present the CX algorithm in §3. We perform an in-depth evaluation of CX in §4 and finally conclude in §5. Proofs of correctness are presented in Appendix.

2 RELATED WORK

In 1983, Peterson [22] was the first to attack the problem of non-blocking access to shared data and to provide several solutions to what he called the *concurrent reading while writing* problem. One of these solutions uses two instances of the same data and guarantees wait-free progress for both reads and writes, allowing multiple readers and a single writer to access simultaneously any of the two instances. However, this approach is based on *optimistic concurrency*, causing read-write races, which has troublesome implications in terms of atomicity, memory reclamation and invariance conservation.

Later, in 1990, Maurice Herlihy [17] proposed the first wait-free UC for any number of threads. This UC requires no annotation or modification to the sequential implementations and is therefore a non-instrumenting UC. His approach keeps a list of all operations ever applied, and for every new operation it will re-apply all previous operations starting from an instance in its initial state. One by one, as each operation is appended, the list of operations grows unbounded until it exhausts all available memory, thus making this UC unsuitable for practical usage.

Since then, several wait-free UCs have been proposed [2, 3, 6, 9, 10]. Researchers have attempted to address the problem of applying wait-free UCs to large objects [1, 2, 17] though none has succeeded in providing a generic solution [24].

Anderson and Moir [2] have proposed a technique designed to work well with large objects. This algorithm is an instrumenting UC because it requires the end user to write a sequential procedure that treats the object as if it was stored in a contiguous array, implying adaptation or annotation of the sequential implementation. This technique is also not universally applicable to all data structures.

Chuong et al. [6] have shown a technique that makes a copy of each shared variable in the sequential implementation and executes the operations on the copy. Although this technique can operate at the level of memory words, it would be vulnerable to race conditions from different (consecutive) operations that modify the same variables. Even if a CAS would be used to modify these variables, ABA issues could still occur. This word-based approach requires instrumentation of the user code.

Fatourou and Kallimanis [10] have designed and implemented P-Sim, a highly efficient wait-free and non-instrumenting UC based on fetch-and-add and LL/SC. P-Sim relies on an up-to-date instance that all threads copy from, using Herlihy's combining consensus [12] to establish which operations are to be applied on the copy, independently of whether these are read or write operations. In the best-case scenario, one copy of the entire object state is made per N concurrent operations, where N is the number of threads. In the worst-case, two copies are done per operation. Unfortunately, P-Sim is impractical for large objects.

More recently, Ellen et al. [9] have shown a wait-free UC based on LL/SC. Their technique provides disjoint access parallel operations with the requirement that all data items in the sequential code are only accessed via the instructions `CreateDI`, `ReadDI` and `WriteDI`, implying the need for instrumentation of the sequential implementations similar to an STM. No implementation has been made publicly available.

3 CX ALGORITHM

The CX wait-free construct uses a wait-free queue where mutations to the object instance are placed, much like Herlihy's wait-free construct, though instead of each thread having its own copy of the instance, there are a limited number of copies that all threads can access. The access to each of these copies is protected by a reader-writer lock, which can be acquired by multiple reader threads in *shared mode*, whereas only one writer thread can get the lock in *exclusive mode*. The reader-writer lock used in CX must guarantee that, when multiple threads compete for the lock using the `trylock()` method, at least one will succeed and obtain the lock. This property, named *strong trylock* [8], combines *deadlock freedom* with linearizable consistency and wait-free progress.

The CX construct (see Figure 1) is composed of: *(i)* `curComb`: a pointer to the current `Combined` instance; *(ii)* `tail`: a pointer to the last node of the queue; and *(iii)* `combs`: an array of `Combined` instances.

In turn, a `Combined` instance consists of: *(i)* `head`: a pointer to a `Node` on the queue of mutations; *(ii)* `obj`: a copy of the data structure or object that is up to date until `head`, *i.e.*, any mutative operation that was enqueued after `head` has not yet been applied to `obj`; and *(iii)* `rwlock`: an instance of a reader-writer lock that protects the content of the `Combined` instance.

Finally, a `Node` holds: *(i)* `mutation`: a function to be applied on the object; *(ii)* `result`: the value returned by the update function, if any; *(iii)* `next`: a pointer to the next `Node` in the mutation queue; *(iv)* `ticket`: a sequence number to simplify the validation in case of multiple threads applying the same mutations; *(v)* `refcnt`: a reference counter for memory reclamation, as well as some other fields for internal use. The definitions for the main data structures of CX are shown in Algorithm 1.

Algorithm 1 CX data structures

```
1  template<typename C, typename R = uint64_t>
2  class CX {                    // C = "Object type" ; R = "Result type"
3    const int maxThr;
4    std::atomic<Combined*> curComb {nullptr};
5    std::function<R(C*)> mut0 = [](C* c) {return R{};};
6    Node* sentinel = new Node(mut0, 0);
7    std::atomic<Node*> tail {sentinel};
8    Combined* combs;
9    struct Node {
10     std::function<R(C*)> mutation;          // Not a pointer
11     std::atomic<R> result;                  // Not a pointer
12     std::atomic<Node*> next {nullptr};
13     std::atomic<uint64_t> ticket {0};
14     const int enqTid;                       // Used internally by queue
15   };
16   struct Combined {
17     Node* head {nullptr};
18     C* obj {nullptr};
19     StrongTryRWRI rwLock {maxThr};
20   };};
```

Figure 1 illustrates the data structures and principle of CX on a concurrent stack. Mutative operations in the wait-free queue are represented by rounded rectangles, with node A corresponding to operation `push(a)`.

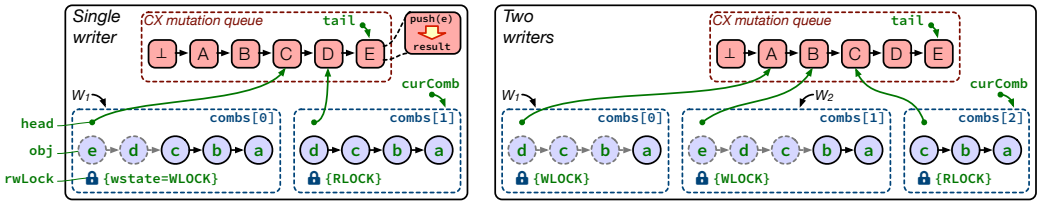

Fig. 1. Illustration of CX's principle on two scenarios with one (left) and two (right) writers pushing elements a through e in a shared stack.

The stack stores its elements in a linked list of nodes (circles), with dashed lines indicating the nodes that are not yet added to the specific instance of the data structure.

The CX construct relies on the copies of the object present in the `combs[]` array. Initially, one of the `Combined` instance in the array holds an initialized object `obj` and its `head` pointer refers to the sentinel node ⊥ of CX's wait-free queue of mutations. The other instances have both `obj` and `head` set to null.

To improve *readers* performance, CX has distinct code paths for *readers* and *updaters*. Readers call `applyRead()`, which tries to acquire the shared lock on the reader-writer lock instance of the current `Combined` instance, `curComb`. Updaters call `applyUpdate()`, which scans the `combs[]` array and attempts to acquire the exclusive lock on the reader-writer lock of one of the `Combined` instances (which is guaranteed to succeed after a maximum of `numReaders+2×numUpdaters` trials).

An updater thread has to account for a possible read operation that will be executed in the most up-to-date copy, which is referenced by `curComb`. The updater is responsible for leaving `curComb` referencing a copy that contains its mutative operation, and this copy is left with a shared lock held so as to protect it from being acquired in exclusive mode by updater threads, including itself. When a copy of the object is required, the copy procedure will try to acquire `curComb` in shared mode and copy its `obj` while holding the shared lock, therefore guaranteeing a consistent replica. This implies that two `Combined` instances may be required for each updater thread: the original copy and the new replica. If we consider that the construct will be accessed by at most `numReaders` dedicated readers and `numUpdaters` dedicated updaters, then the maximum number of `Combined` instances in use at any given time will be one per reader plus two per updater, *i.e.*, `numReaders+2×numUpdaters` (`2×maxThreads` if every thread can potentially update the data structure).

Once an updater thread secures a `Combined` instance with exclusive access and ensures it has a copy of the object, the updater is responsible for applying all the mutations present in the mutation queue following the sequential order from the `head` of the `Combined` instance until its own mutative operation, which was previously added to the queue of mutations. Each node has a `ticket` that simplifies the validation in case the mutation has already been applied and `head` is more recent than the node N containing the mutative operation to be made *visible*. The concept of visibility refers to a state of the object, where effects of operations on the object are available to all threads. After the `Combined` instance is brought up-to-date with a copy of the object containing the updater's mutation, the updater thread has to make its mutation visible to other threads by ensuring that `curComb` advances to a `Combined` instance whose `head` has a `ticket` greater than or equal to N's `ticket`.

We define a valid copy of the object as an instance that can be brought up to date, applying all the mutations starting from the `head` of the `Combined` where the copy is stored, until N's mutation. An invalidation of a copy occurs when there is memory reclamation of the queue's nodes. In case a copy is invalidated, a new copy can be created from `curComb`.

Consider Figure 1 to better understand how mutations are propagated to the available copies. In the left figure, a single writer W_1 has been pushing values `a`, `b`, `c` and `d`, and is in the process of executing `push(e)`. The mutative operation has already been inserted at the tail of CX's wait-free queue but not yet applied to the stack. At that point `curComb` points to `combs[1]`, which holds an up-to-date stack (with all 4 elements inserted) protected by a shared lock. Hence the writer cannot use this instance and instead acquires `combs[0]` in exclusive mode. The next steps for the writer will be to apply the operations starting from the `head`, *i.e.*,

push d and e, to bring the data structure up to date, update head to point to the last applied mutation (node E), atomically set curComb to point to combs[0], and finally downgrade the lock to read mode. The figure on the right presents a similar scenario but with two concurrent writers W_1 and W_2, with the first one executing push(d) on combs[0] and the second one push(e) on combs[1]. The order of operations is determined by their position in the wait-free queue, i.e., d is inserted before e and both writers will apply the operations on the Combined instances in this order.

3.1 Algorithm Walkthrough

Algorithm 2 CX ApplyRead and ApplyUpdate pseudo-code

```
 1: function applyRead(readFunc,tid):
 2:   for _i ← 0, MAX_READ_TRIES + MAX_THREADS do
 3:     if _i = MAX_READ_TRIES then
 4:       _myNode ← enqueue(updFunc)
 5:     _comb ← curComb
 6:     if _comb.rwlock.sharedTryLock(tid) then
 7:       if _comb == curComb then
 8:         _ret ← readFunc()                    ▷ Function call
 9:         _comb.rwlock.sharedUnlock(tid)
10:         return _ret
11:       _comb.rwlock.sharedUnlock(tid)
12:   end for              ▷ A writer must have completed its operation...
13:   return _myNode.result          ▷ ...and the result is in myNode
14: end function

15: function applyUpdate(updFunc):
16:   _myNode ← enqueue(updFunc)                                  {1}
17:   _tkt ← myNode.ticket
18:   _c, _idx ← exclusiveTryLock()                               {2}
19:   _mn ← _c.head
20:   if _mn ≠ null ∧ _mn.ticket ≥ _tkt then
21:     _c.rwLock.exclusiveUnlock()
22:     return _myNode.result
23:   _comb ← null
24:   _combIdx ← -1
25:   while _mn ≠ _myNode do
26:     if _mn = null ∨ _mn = _mn.next then                       {3}
27:       _combIdx ← getCombined(_tkt)
28:       if _comb ≠ null ∨ _combIdx = -1 then
29:         _c.head ← _mn
30:         _c.rwLock.exclusiveUnlock()
31:         return _myNode.result
32:       _comb ← combs[_combIdx]
33:       _mn ← _comb.head
34:       _c.updateHeadObj(_comb, _mn)
35:       _comb.rwLock.sharedUnlock()
36:       continue
37:     _mn ← _mn.next
38:     _mn.result.store(_mn.updFunc(_comb.obj))                  {4}
39:   _c.head ← _mn                                               {4}
40:   _c.rwLock.downgradeToHandover()                             {5}
41:   for _i ← 0, MAX_THREADS do
42:     _combIdx ← curComb
43:     _comb ← combs[_combIdx]
44:     if ¬sharedTryLockCheckTkt(_comb) then
45:       continue
46:     if curComb.cas(_combIdx, _idx) then                       {6}
47:       _comb.rwLock.handoverUnlock()
48:       _comb.rwLock.sharedUnlock()
49:       return _myNode.result
50:     _comb.rwLock.sharedUnlock()
51:   end for
52:   _c.rwLock.handoverUnlock()
53:   return _myNode.result
54: end function
```

The core of the CX algorithm resides in the applyUpdate() mutative operation, shown in Algorithm 2. The main steps of the algorithms are:

(1) Create a new Node _myNode with the desired mutation and insert it in the queue (line 16).
(2) Acquire an exclusive lock on one of the Combined instances in the combs[] array (line 18).
(3) Verify if there is a valid copy of the data structure in _c and make a copy if necessary (line 26).
(4) Apply all mutations starting at head of the Combined instance until reaching the Node inserted in the first step (lines 25 to 38), and update head to point to this node (line 39).
(5) Downgrade lock on _c (line 40).
(6) Compare-and-set (CAS) curComb from its current value to the just updated Combined instance (line 46). Upon failure, retry CAS until successful or until head of the current curComb instance is *after* _myNode.

When applying a mutation to the underlying object, the first step is to create a new node with the mutation (line 16 of Algorithm 2) and insert it in CX's queue. Each node contains a mutation field that stores the mutation. A monotonically increasing ticket is assigned to the node to uniquely identify the mutation (line 16).

The next step consists in finding an available Combined instance on which to apply the new mutation. To that end, the thread must acquire a Combined's lock in exclusive mode (line 18). The StrongTryRWRI [8] reader-writer lock provides a *strong* exclusiveTryLock() method, guaranteeing that the lock will be acquired in at most 2×maxThreads attempts.

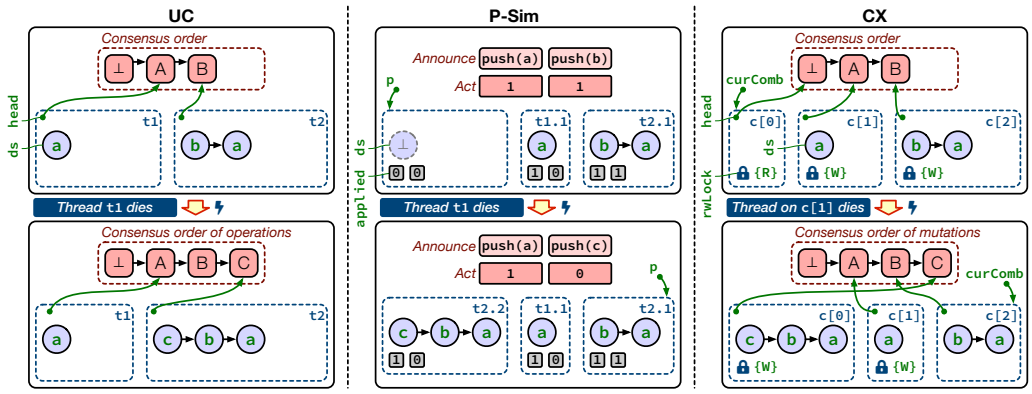

Fig. 2. Comparison of UC [17], P-Sim [10] and CX.

If the locked `Combined` instance has an invalidated or null `obj` (line 26), we need to make a copy of the current object. To do so, we first acquire the shared lock on `curComb` (line 27), before updating `head` and copying `obj` (line 34). It is worth mentioning that copy-on-write (COW) based techniques usually make one such copy for every mutative operation, while CX does this *once* for every new used `Combined` (of which there are `2×maxThreads`) plus the number of times a copy is invalidated.

Next, we apply the mutations on `obj`, starting from the corresponding `head` node (line 19) until our newly added node is found (line 25), always saving the result of each mutation in the corresponding `node.result` (line 38). The rationale for saving the result is that, if another thread calling `applyUpdate()` sees that its own mutation is already visible at `curComb`, it can directly return the result of the mutation (lines 22, 31, 49 and 53) without having to actually execute the previous mutations in the queue. This approach implies that multiple threads may be write-racing the same value into `node.result`, which means that it must be accessed atomically, further implying that `R` must fit in a `std::atomic` data type to ensure wait freedom.

After the mutations have been applied, we downgrade the lock on `_c` and advance `curComb` with a CAS (line 46) so as to make the current and previous mutations visible to other threads: `curComb` will now reference a `Combined` instance that contains the effects until `head.ticket`, also `curComb` will be protected by a shared lock to guarantee the instance is always available to execute a read operation. In addition, `curComb` always transitions to a `Combined` instance with a `head.ticket` higher than the previous one thanks to the test on line 44. This guarantees that operations whose effects are visible on `curComb` will remain visible. Finally, we can unlock the unneeded `Combined` instance to make it available for exclusive locking by other threads searching for their `_c`, before returning the result of our mutation.

As mentioned previously, a read operation will attempt to acquire a shared lock in the most up to date instance `curComb`. The reader may be unsuccessful in acquiring the lock if an updater has already acquired the lock. Such a situation can occur if, between the load on line 5 and the call to `sharedTryLock()` on line 6, the `curComb` advances to another `Combined` instance and an updater takes the exclusive lock on the previous instance. This results in a lock-free progress condition for read operations, as it would only fail to progress in case an updater had made progress. To guarantee wait-free progress, the reader must publish its operation in the queue after `MAX_READ_TRIES` attempts (line 4), but it is not required to apply all mutations up to its own operation. After a maximum of `maxThread` transitions of `curComb` the reader's operation will be processed by an updater thread and become visible. In case there is no updater thread to process the read operation, this implies that there is also no updater thread to block the reader thread from acquiring the lock on `curComb`.

To better understand the differences between Herlihy UC [17], P-Sim [10] and CX, we show in Figure 2 a comparison example where two writer threads are pushing elements to a stack. Both threads reach a consensus where the operation `push(a)` will be executed followed by `push(b)`. Thread T_1 terminates abruptly while executing operation `push(a)`, while thread T_2 continues execution pushing element `c` to the stack. With Herlihy's UC, every thread maintains its own copy of the data structure and each copy has

to know the exact order in which the operations must be executed. Because T_2 has no way to know if T_1 has died or is just delayed, the order of the operation from A onwards has to be kept, which will cause it to grow indefinitely, eventually exhausting the system's memory.

P-Sim uses a copy-on-write approach with Herlihy's combining consensus [12]. Each thread performs a copy of the object referenced by P and applies to its copy all operations newly published in the *announce* array. There is no consensus order that all threads agree upon; instead, the thread that is able to transition P to its copy is the one establishing the order of the operations. In case thread T_1 dies, its copy (`t1.1`) is left unreclaimed but this has no impact on the execution of thread T_2.

CX has a pool of `2×maxThreads` copies of the data structure that all threads can use to execute their operation, and uses a *turn queue* for consensus [23]. In the example, there are four available copies in the pool but the two threads only used three `Combined` instances. In the scenario where a thread dies while executing on the second one (`c[1]`), this instance is no longer available to the remaining threads. But in case the thread died after releasing the exclusive lock, then the instance would remain available to be used. Also, the order of the mutations can be disposed off up until `curComb.head`, because at any given time there is an object referenced by `curComb`, that is protected by a shared lock, from which any thread can execute a copy. In addition, there are at most `maxThread` operations after `curComb.head` that remain to be executed.

3.2 Reader-Writer Lock with Strong Trylock

Access to each `Combined` instance and, consequently, to each copy of the object is managed by a reader-writer lock, `Combined.rwlock`. In order to ensure wait-free progress, the reader-writer lock has to guarantee that from all the threads competing for the lock at least one will acquire it, a guarantee sometimes called *deadlock freedom for trylock*, and furthermore the `trylock()` method must complete in a finite number of steps [8].

Based on these requirements, we chose to use the StrongTryRWRI reader-writer lock proposed in [8]. This lock's high scalability is capable of matching other state of the art reader-writer locks [5] while providing `downgrade()` functionality and strong trylock properties. In addition, CX requires *lock handover* between different threads when in shared mode. In CX, the `rwlock` of each `Combined` instance can be in one of four logical states: unlocked; shared, *i.e.*, read-only; exclusive, *i.e.*, read-write; or handover. The handover state, which is not typical in `rwlock` implementations, represents a state in which the lock is left in shared (read-only) mode without any thread actually using it, with the purpose of preventing writers from acquiring the lock in exclusive mode, `handoverLock()` or `downgradeToHandover()` will leave the lock in handover state. The `rwlock` implementation allows the unlock of the shared mode by a different thread from the one which acquired the lock in shared mode, `handoverUnlock()`.

3.3 Wait-Freedom

The `applyUpdate()` method has only one loop where the number of iterations is not predetermined (line 25). For it to terminate, the traversal of the wait-free queue must encounter the node containing the process' update operation. The process starts by appending its update operation with sequence number l to the wait-free queue and will proceed to acquire an exclusive lock for `Combined` instance $Comb_i$. It is guaranteed by Proposition 2 that each process will always have available two `Combined` instances to execute, even if all other threads fail holding two Combined instances locked, one in exclusive and another in shared mode. For the process to execute the loop at line 25, $Comb_i$'s' state must be $<O_{i,j}, head_{i,j}>$ where $l > j$, otherwise `applyUpdate()` would return at line 22. From Proposition 3, $Comb_i$'s state will transition to $<O_{i,l}, head_{i,l}>$ in $l-j$ iterations, unless a copy of object O is required (line 26). In case the process is unable to do a copy, it will return at line 31, thus terminating the loop. Otherwise, the copy from a `Combined` instance $Comb_k$ referenced by *curComb* with state $<O_{k,m}, head_{k,m}>$ is executed at line 34. The copy from O_k is guaranteed to execute in a finite number of steps because $Comb_k$ is protected by a shared lock, which guarantees that no update operation is taking place during the copy procedure. After the copy is completed, $Comb_i$ will be in the state $<O_{i,m}, head_{i,m}>$. The copy was performed from a `Combined` instance

referenced by *curComb*, and Corollary 1 guarantees that $l - m \leq$ `maxThreads`. Consequently, the loop at line 25 will iterate at most `maxThreads` times after the copy procedure. In all possible scenarios, the loop will always iterate a finite number of steps. The `applyUpdate()` method also calls `enqueue()` at line 16 and the try-lock methods, `exclusiveTryLock()`, `exclusiveUnlock()`, `sharedTryLock()`, `sharedUnlock()`, `downgradeToHandover()` and `handoverUnlock()`. By definition, all these methods return in a finite number of steps, from which we conclude that `applyUpdate()` has wait-free progress. In addition, assuming the sequential copy of an object is bounded, then the `applyUpdate()` method is also wait-free bounded. The loop at line 25 is the only one that can be unbounded. Reclamation of the nodes is done once the thread-local circular buffer is full, which implies the queue is composed of a limited number of nodes. Considering that $size_{circbuff}$ represents the size of the circular buffer, then the maximum amount of iterations is bounded by:

$$size_{circbuff} \times maxThreads + maxThreads \quad (1)$$

The `applyRead()` method iterates for a maximum of `MAX_TRIES+maxThreads`. It calls `sharedTryLock()` and `sharedUnlock()`, which by definition return in a finite number of steps, resulting in wait-free progress. Refer to Appendix A for the definitions of variables, Propositions and Corollary.

4 EVALUATION

We now present a detailed evaluation of CX and compare it with other state-of-the-art UCs and non-blocking data structures, using synthetic benchmarks. Our microbenchmarks were executed on a dual-socket 2.10 GHz Intel Xeon E5-2683 ("Broadwell") with a total of 32 hyper-threaded cores (64 HW threads), running Ubuntu LTS and using gcc 7.2.

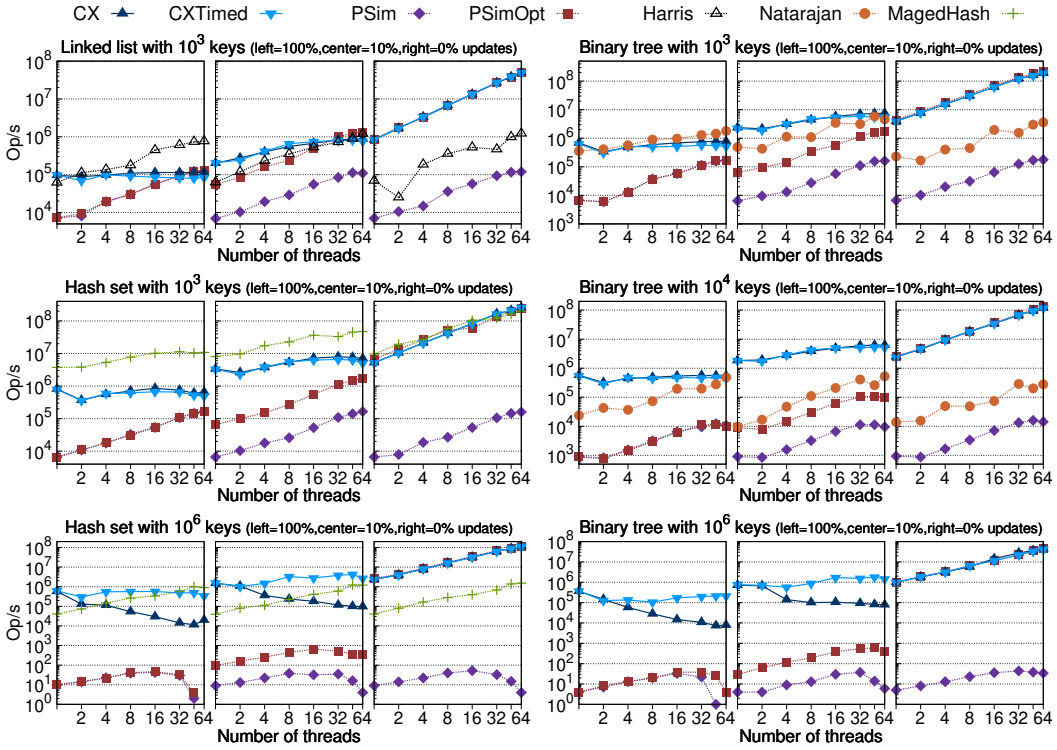

Fig. 3. Left column (top to bottom): sets implemented using a linked list with 10^3 keys and using a hash table with 10^3 and 10^6 keys. Right column: sets implemented using the balanced binary search trees std::set, with 10^3, 10^4 and 10^6 keys. The results are presented with a logarithmic scale on both axes.

Besides CX, we used two other UCs which we now describe. `PSim` [10] is a UC with wait-free progress. We adapted the original implementation available on `github`. `PSimOpt` is an extension to `PSim`, where read-only

operations have a different code path that allows them to scale, using a technique similar to the one we developed for CX. We also added a modified version of CX, called CXTimed, which restricts the amount of available Combined instances to four for a bounded period of time. For CXTimed, a thread is initially restricted to acquiring an exclusive lock on the first 4 Combined instances for a duration that corresponds to the time it takes to do a copy of the object. After that amount of time has elapsed and its operation remains to be executed, the thread will acquire an exclusive lock on one of the 2×maxThreads instances. This can be seen as a blocking fast path with only 4 available Combined instances that can always revert to a slower path with wait-free progress. This approach further reduces the amount of object copies, because with high probability the first 4 instances are kept up to date.

Depending on the benchmark, we also compared with commonly available lock-free data structures. MagedHP is a Harris linked list set modified by Michael [19]. Natarajan is the relaxed tree by Natarajan and Mittal [21]. MagedHash is the hash table by Michael [19]. All implementations use Hazard Pointers [20].

All these data structures are *sets* and the microbenchmarks described next have the same procedure. A set is filled with 1,000 keys and we randomly select doing either a lookup or an update, with a probability that depends on the percentage of updates for each particular workload. For a *lookup*, we randomly select one key and call contains(key); for an *update*, we randomly select one key and call remove(key), and if the removal is successful, we re-insert the same key with a call to add(key), thus maintaining the total number of keys in the set (minus any ongoing removals). Depending on the scenario, the procedure may be repeated for sets of different key ranges. Each run takes 20 seconds, where a data point corresponds to the median of 5 runs. All implementations will be available publicly.

The results of our experiments are shown in Figure 3, with a log-log scale. As expected, a sorted linked list protected by CX is surpassed in most workloads by Maged-Harris' lock-free set because of the serialization of all operations in the wait-free queue necessary to reach consensus. It is interesting to notice, however, that Maged-Harris algorithm is not able to outperform CX in the scenario of 10% updates. CX read operations do not require any pointer tracking during traversal because the data structure where the operation is executed is protected by a shared lock, which is not the case for traversals with Maged-Harris.

Let us now consider experiments with hash sets. The MagedHash algorithm uses a pre-allocated array of 1,000 buckets and its advantage over CX is significant, due to CX serializing all mutative operations, while updates on the MagedHash are mostly disjoint. However, when we insert one million keys, the fact that there are only 1,000 buckets causes increased serialization of the update operations, giving CX an edge in nearly all scenarios. Currently there is no known efficient hand-made lock-free resizable hash set with lock-free memory reclamation.

Regarding balanced trees, three different workloads were executed with 10^3, 10^4 and 10^6 keys, shown in Figure 3. Natarajan's tree is not shown for 10^6 keys because it takes two hours to fill up, making it unsuitable for such a scenario. This occurs because it is a *non-balanced* tree and our benchmarks execute a consecutive fill of the keys, causing this tree to effectively become a linked list of nodes because it is *never* rebalanced. There are a few lock-free balanced trees in the literature [4], however, there is no known implementation with hazard pointers or any other lock-free memory reclamation. Balanced trees like the std:set sequential implementation we use in CX do not suffer from these issues. For a small tree with 10^3 keys, with 100% updates, CX is the most efficient for single-threaded execution and is not far behind the lock-free tree for the remaining thread counts. As the ratio of read-only operations increase, CX improves and at 10% updates it is able to beat the lock-free tree, irrespective of the number of threads. For a tree with 10^4 and 10^6 keys, CX has the advantage on all tested scenarios.

The two CX implementations evaluated in this section give high scalability for read-mostly workloads regardless of the underlying data structure. Read-only operations in CX can almost always acquire the shared lock after a few trylock attempts, which implies that the synchronization cost is a few sequentially consistent stores. This high throughput surpasses equivalent lock-free data structures, while providing wait-free progress and linearizable consistency for any operation. For high update workloads, equivalent lock-free data structures may have higher performance than CX.

As for other UCs, PSim drags far below in all tested scenarios due to serializing all operations, even though it has been until now the best of the non-interposing UCs, easily surpassing Herlihy's original wait-free UC (not shown in this paper). Our optimized implementation with scalable reads, PSimOpt, greatly improves the throughput on workloads with 0% updates but as soon as the number of update operations increase it shows similar performance when compared with PSim.

4.1 Memory Usage

For large data structures, the amount of memory required to execute the program can be a determining factor when choosing a more suitable concurrency synchronization. We conducted an experiment meant to evaluate the trade-off between memory usage and throughput. The experiment follows the same procedure as for update-only workloads, using the balanced binary search tree available in STL, with pre-filled trees of 1 and 10 million keys. The maximum memory usage is measured executing the same microbenchmark. Each data point of Figure 4 is the highest value of two runs, each run executing for 100 seconds.

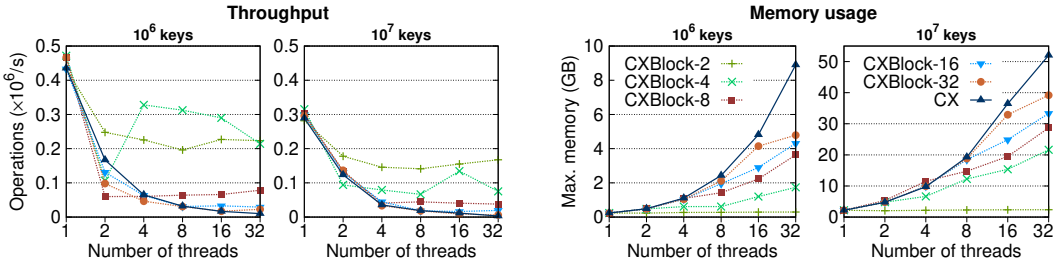

Fig. 4. Throughput (left) vs. maximum memory usage (right) with 100% updates for a pre-filled std::set of 10^6 and 10^7 keys. CXBlock-k represents the CX universal construct where $maxObjs = k$. CX is the wait-free UC where the maximum number of objects is 64.

We observe in Figure 4 that, unsurprisingly, the configuration with the lowest memory usage is when CX has *maxObjs* set to 2. It is only using two Combined instances with a constant maximum memory usage around 280 MB. When increasing the size of the data structures from 10^6 to 10^7 keys, we observe that memory usage grows as expected by a factor of ten, around 2 GB.

On the other hand, CX with wait-free progress (*i.e.*, when *maxObjs* is set to 64) is the configuration with the highest memory requirements. As the number of threads grows, we observe both an increase in memory usage and a decrease in throughput, but this is compensated by the additional guarantees of resilience in case of thread failures. Our experiments also show that, for the tree with 1 million keys, CXBlock-4 sometimes achieves better performance than CXBlock-2. We can reach the conclusion that, if the application can relax the progress guarantees for update operations, then a suitable configuration would be to use up to 4 object instances. This would provide a good trade-off between memory usage, throughput and progress.

5 CONCLUSION

The appeal of generic techniques like wait-free universal constructs (UC) stems from the difficulty in designing *hand-written* non-blocking data structures. These constructs can transform any sequential implementation of a data structure into a correct wait-free data structure, with linearizable consistency for *all* operations.

CX is the first non-instrumenting UC capable of transforming *any* sequential implementation of a data structure with unforeseen method implementations to be considered for multi-threaded applications with performance that rivals and surpasses hand-made lock-free implementations.

Moreover, CX has integrated wait-free memory reclamation, a feature that most hand-written lock-free data structures do not provide. Using CX we have implemented the first wait-free binary *balanced* tree, showing that CX makes it possible to create new data structures for which no hand-made counterparts exist

yet. CX's huge leap in performance compared with previous UCs is due to the significant reduction of copy operations, where available copies are instead reused and updated.

REFERENCES

[1] Yehuda Afek, Dalia Dauber, and Dan Touitou. 1995. Wait-free made fast. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*. ACM, 538–547.

[2] James H Anderson and Mark Moir. 1995. Universal constructions for large objects. In *International Workshop on Distributed Algorithms*. Springer, 168–182.

[3] James H Anderson and Mark Moir. 1995. Universal constructions for multi-object operations. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. ACM, 184–193.

[4] Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A general technique for non-blocking trees. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 329–342.

[5] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J Marathe, and Nir Shavit. 2013. NUMA-aware reader-writer locks. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 157–166.

[6] Phong Chuong, Faith Ellen, and Vijaya Ramachandran. 2010. A universal construction for wait-free transaction friendly data structures. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 335–344.

[7] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. ACM, 271–282.

[8] Andreia Correia and Pedro Ramalhete. 2018. Strong trylocks for reader-writer locks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 387–388.

[9] Faith Ellen, Panagiota Fatourou, Eleftherios Kosmas, Alessia Milani, and Corentin Travers. 2016. Universal constructions that ensure disjoint-access parallelism and wait-freedom. *Distributed Computing* 29, 4 (2016), 251–277.

[10] Panagiota Fatourou and Nikolaos D Kallimanis. 2011. A highly-efficient wait-free universal construction. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 325–334.

[11] Maurice Herlihy. 1991. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1 (1991), 124–149.

[12] Maurice Herlihy. 1993. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15, 5 (1993), 745–770.

[13] Maurice Herlihy and J Eliot B Moss. 1993. *Transactional memory: Architectural support for lock-free data structures*. Vol. 21. ACM.

[14] Maurice Herlihy and Jeannette Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492.

[15] Amos Israeli and Lihu Rappoport. 1994. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*. ACM, 151–160.

[16] Peter Magnusson, Anders Landin, and Erik Hagersten. 1994. Queue locks on cache coherent multiprocessors. In *Parallel Processing Symposium, 1994. Proceedings., Eighth International*. IEEE, 165–171.

[17] A Maurice Herlihy. 1990. Methodology for Implementing Highly Concurrent Data Objects. In *Proceedings of ACM PPoPP*. 197–206.

[18] John M Mellor-Crummey and Michael L Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)* 9, 1 (1991), 21–65.

[19] Maged M Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 73–82.

[20] Maged M Michael. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on* 15, 6 (2004), 491–504.

[21] Aravind Natarajan and Neeraj Mittal. 2014. Fast concurrent lock-free binary search trees. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 317–328.

[22] Gary L Peterson. 1983. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5, 1 (1983), 46–55.

[23] Pedro Ramalhete and Andreia Correia. 2017. POSTER: A Wait-Free Queue with Wait-Free Memory Reclamation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 453–454.

[24] Michel Raynal et al. 2017. Distributed Universal Constructions: a Guided Tour. *Bulletin of EATCS* 1, 121 (2017).

A CORRECTNESS

We discuss the correctness of CX using standard definitions and notations for linearizability [14], which we briefly recall below. A concurrent execution is modeled by a history, *i.e.*, a sequence of events. Events can be operation invocations and responses, denoted respectively as $op.inv$ and $op.res$. Each event is labeled with the process and with the object O to which it pertains. A subhistory of a history H is a subsequence of the events in H. A response matches an invocation if they are performed by the same process on the same object. An operation in a history H consists of an invocation and the next matching response. An update operation may cause a change of state in the object, with a visible effect to other processes, while a read-only operation has no effects visible to other processes. An invocation is pending in H if no matching response follows it in H. An extension of H is a history obtained by appending responses to zero or more pending invocations in H, and *complete*(H) denotes the subhistory of H containing all matching invocations and responses. All references to specific lines of code in this section refer to Algorithm 2.

▶ **Definition 1.** (Happens before) *if* $op_1.res <_{hb} op_2.inv$ *then* $op_1 <_{hb} op_2$.

▶ **Definition 2.** (Subhistory) *Given a history H, a subhistory S of H is such that if op_2 belongs to S and $op_1 <_{hb} op_2$ in H, then op_1 belongs to S and $op_1 <_{hb} op_2$ in S.*

▶ **Definition 3.** (Partial subhistory) *Given a history H and update operations op_1 and op_2, a partial subhistory S of H is such that if op_2 belongs to S and $op_1 <_{hb} op_2$ in H, then op_1 belongs to S and $op_1 <_{hb} op_2$ in S.*

▶ **Definition 4.** (Linearizability) *A history H is linearizable if H has an extension H' and there is a legal sequential subhistory S such that:* (i) *complete(H') is equivalent to S; and* (ii) *if an operation $op_1 <_{hb} op_2$ in H, then the same holds in S.*

CX's correctness relies on a linearizable wait-free queue, and a linearizable reader-writer lock with strong guarantee for trylock methods. The queue represents the sequence of update operations applied to the object O, establishing the partial history of the concurrent execution. For simplicity, we assume that the queue is formed by a sequence of nodes and each has a unique sequence number, which is monotonically increasing for consecutive nodes.

CX requires a linearizable wait-free enqueue method and a linearizable traversal of the queue that provides a partial subhistory of the history H. Regarding the linearizable reader-writer lock, it must provide strong guarantees for the `sharedTrylock()` and `exclusiveTrylock()` methods. Both methods must satisfy the property of *deadlock-freedom*, *i.e.*, the critical section will not become inaccessible to all processes, and their invocation must complete in a finite number of steps.

We denote by $Comb_i$ the i_{th} `Combined` instance of the `combs[]` array. At any given moment, $Comb_i$ is in a state represented by the pair $<O_{i,j}, head_{i,j}>$. $O_{i,j}$ represents the i_{th} simulation of object O where j corresponds to the sequence number in a node of the wait-free queue. As such, $O_{i,0}$ is the initialized object and $O_{i,j}$ is the simulation of object O after sequentially applying all update operations up to the operation with sequence number j. We assume all $Comb_i$ instances start in the initial state $<O_{i,0}, head_{i,0}>$ where $head_{i,0}$ is the sentinel node of the wait-free queue. $head_{i,j}$ represents a node with sequence number j in the wait-free queue. $O_{i,j}.op_{j+1}()$ represents the execution of operation $op_{j+1}()$ on $O_{i,j}$, and the resulting object $O_{i,j+1}$ will contain the effects of $op_{j+1}()$. We define *curComb* as the `Combined` instance on which read-only operations execute.

▶ **Proposition 1.** *curComb can only transition between different* `Combined` *instances both protected by a shared lock.*

Proof. At the start of the execution, *curComb* references a `Combined` instance protected by a shared lock. The state transition of *curComb* occurs in line 46 between two `Combined` instances, referred as *lComb* and *newComb*. The lock associated with the *newComb* instance is acquired in exclusive mode in line 18 and is later downgraded to shared mode in line 40. It is not possible for *newComb* to be the *lComb* because *lComb* is protected by a shared lock. From the moment a process q acquires the exclusive lock protecting *newComb*, until the state transition in line 46, other processes may change *curComb* from *lComb* to reference another `Combined`

instance. However, those processes will not be able to change *curComb* to reference *newComb*. Any other process attempting to transition *curComb* will have to first acquire an exclusive lock on a `Combined` instance, and it is impossible that this instance is *newComb* because *newComb*'s lock is held by process q. As such, *curComb* can only transition to a different `Combined` instance and that instance's lock is held in shared mode. □

► **Proposition 2.** *At most* $2 \times$ `maxThreads` `Combined` *instances are necessary to guarantee that an update operation will acquire an exclusive lock on one of the* `Combined` *instances.*

Proof. A process executing `applyUpdate()` will require at most two `Combined` instances at any given time, the acquisition of an exclusive lock at line 18 and a shared lock at line 27 or 44. Assuming the reader-writer trylock methods guarantee that no available `Combined` instance can remain inaccessible to all competing processes, this implies that any process that failed to acquire a lock in a `Combined` instance is sure that the instance is in use by a competing process. By induction, lets consider that processes $q_1,\ldots,q_{n-1}$ use $2\times$(`maxThreads`-1) `Combined` instances. The last process q_n will have available the last two `Combined` instances. Considering that process q_1 releases the shared lock of $Comb_i$ and leaves the other in handover state. In a subsequent call to `applyUpdate()`, process q_1 will acquire the exclusive lock on $Comb_i$ because this is the first available `Combined` instance when traversing the `combs[]` array, and the shared lock will be acquired on one of the $2\times$(`maxThreads`-1) `Combined` instances that can potentially be *curComb*. In the event that process q_n transitions *curComb* to one of its two available instances, then process q_1 can acquire that `Combined` instance in shared mode but it will leave a precedent `Combined` instance available to be acquired in exclusive mode by process q_n. This shows that there will always be two `Combined` instances available to process q_n, the maximum it may need. □

► **Proposition 3.** *For any* $Comb_i$*, an update operation with sequence* l *will transition atomically from* $<O_{i,j},head_{i,j}>$ *to* $<O_{i,l},head_{i,l}>$.

Proof. Every `Combined` instance $Comb_i$ is protected by a reader-writer trylock, granting exclusive access in line 18 by proposition 2 to only one process. This process will be allowed to mutate its state from the pair $<O_{i,j},head_{i,j}>$ to a subsequent state. Only a process that is executing an update operation can acquire an exclusive lock on $Comb_i$. Its update operation was previously appended to the queue and we assume the sequence number of the operation is l. Subsequently, the process will execute the statements from line 25 to 38, where the initial simulated object is $O_{i,j}$. $O_{i,j}$ will be subjected to the execution of the sequence of operations $op_k()$ where $k=j+1,\ldots,l$, transitioning the simulated object to $O_{i,l}$. The traversal of the queue is required to be linearizable. The sequence of operations observed by a process traversing the queue is the same for all other processes. All concurrent mutative operations applied to object O were previously appended to the queue, and the queue establishes the partial history of the concurrent execution. The simulated object has now mutated to $O_{i,l}$ and in line 39, $Comb_i$ state will transition to $<O_{i,j},head_{i,j}>$ where $head_{i,l}$ represents the node containing the last operation applied to the simulated object $O_{i,l}$. Only after the transition to state $<O_{i,l},head_{i,l}>$ is completed, will the `Combined` instance $Comb_i$ be made available to other processes, in line 40. □

► **Proposition 4.** *curComb always transitions from* $Comb_i$ *to* $Comb_k$*, with respective states* $<O_{i,j},head_{i,j}>$ *and* $<O_{k,l},head_{k,l}>$*, where* $i \neq k$ *and* $l > j$.

Proof. Proposition 4 follows from Proposition 1 and Proposition 3. In addition, the state transition can only occur (line 46) if $head_{i,j}$ does not satisfy the condition at line 44. This condition guarantees that $l > j$. □

► **Lemma 1.** *An update operation with sequence number* l *can only return, after ensuring curComb transitions to a* $Comb_i$ *with state* $<O_{i,j},head_{i,j}>$ *where* $l \leq j$.

Proof. An update operation with sequence number l will complete as soon as `applyUpdate()` returns (lines 22, 31, 49 or 53). After the exclusive lock is acquired for the `Combined` instance $Comb_k$ with state $<O_{k,m},head_{k,m}>$ at line 20, it will validate if the sequence number of $head_{k,m}$ is greater than or equal to l,

implying that $l \leq m$. If $Comb_k$ can have been acquired in exclusive mode, then any previous update operation with sequence number m that updated $Comb_k$ with update operations until $head_{k,m}$ had to guarantee that *curComb* was referencing a `Combined` instance $Comb_i$ with state $<O_{i,j}, head_{i,j}>$ where $m \leq j$. This proves that an update operation with sequence l can return at line 22 with the guarantee that *curComb* was at least referencing a `Combined` instance $Comb_i$ with state $<O_{i,j}, head_{i,j}>$ where $l \leq j$.

Execution from lines 26 to 36 occurs when the `Combined` instance acquired in exclusive mode requires a copy of *curComb*. The method `getCombined()` can only return null in two cases: in case *curComb* sequence j is higher than or equal to l; otherwise, if after `maxThreads` trials it fails to acquire the shared lock of the current *curComb*, represented as a specific $Comb_i$ with state $<O_{i,j}, head_{i,j}>$. This can only occur if *curComb* changed at least `maxThreads` times. By Proposition 4, *curComb* must be referencing a `Combined` instance with state $<O_{k,m}, head_{k,m}>$ where $j+$`maxThreads` $\leq m$. Assuming no operation can return before ensuring its operation is visible at *curComb* then $l \leq j +$ `maxThreads`, implying that $l \leq m$. This proves that after `maxThreads` transitions of *curComb* it must contain the update operation with sequence l.

On line 49 the update operation returns after ensuring that *curComb* references a `Combined` instance with sequence number lower than l (line 44) and successfully transitions *curComb*, which by Proposition 4 will map to a `Combined` instance with state $<O_{k,l}, head_{k,l}>$ where $i \neq k$ and $l > j$.

When the update operation returns after `maxThreads` failed attempts to acquire the shared lock of the current *curComb* on line 53, *curComb* must contain the update operation with sequence l. □

From Lemma 1, we can directly infer the following corollary.

► **Corollary 1.** *curComb always references a* `Combined` *instance with state* $< O_{i,j}, head_{i,j} >$ *where* $l - j \leq$ `maxThreads`*, with* l *the sequence number at the tail of the wait-free queue.*

We now introduce the remaining lemmas that will allow us to prove linearizability of the CX universal construct.

► **Lemma 2.** *Given* op_1 *and* op_2 *two update operations on object* O*, if* $op_1 <_{hb} op_2$ *and* op_2 *belongs to* S *then* op_1 *belongs to* S*, where* S *is a subhistory of* H *on* O*.*

Proof. Follows from proposition 3. □

► **Lemma 3.** *Given* op_u *an update operation and* op_r *a read-only operation, both on object* O*, if* $op_u <_{hb} op_r$ *then* op_r *will have to see the effects of* op_u *on* O*.*

Proof. If op_r execution accesses a `Combined` instance that does not contain the effects of op_u, then *curComb* has not yet transitioned to an instance that contains op_u. By Lemma 1, the op_u is only considered to take effect after *curComb* transitions to a Combined instance which O_i contains the effects of op_u. This means that op_r could take place before op_u, implying $op_r <_{\text{hb}} op_u$, thus contradicting the initial assumption. □

► **Lemma 4.** *Given* op_r *a read-only operation and* op_u *an update operation, both on object* O*, if* $op_r <_{hb} op_u$ *then* op_r *will not see the effects of* op_u *on* O*.*

Proof. op_u can only return after guaranteeing that *curComb* has transitioned to a `Combined` instance that contains the effects of op_u, by Lemma 1. Any read operation accesses only the Combined instance referenced by *curComb*. If op_r accesses a Combined instance that contains the effects of op_u then it would be possible to consider as if op_u had occurred before op_r, because the current *curComb* already contains op_u. This contradicts the definition of happens-before, if the response of op_u can occur before the invocation of op_r then $op_u <_{\text{hb}} op_r$ which contradicts the initial assumption. Implying *curComb* can not contain the effects of op_u and, therefore, op_r will not see the effects of op_u over object O. □

► **Lemma 5.** *Given* op_1 *and* op_2 *two identical read-only operations on object* O*, if* $op_1 <_{hb} op_2$ *then* op_2 *returns the same result as* op_1*, unless an update operation* op_u *interleaves.*

Proof. By Lemma 1, only update operations can transition the state of *curComb*. All read operations access only the Combined instance referenced by *curComb*. If there is no update operation op_3 interleaving between op_1 and op_2, the read operations will necessary access the same Combined instance yielding the same result. □

The proof of Lemma 3, 4 and 5 rely on the fact that any update operation must guarantee that *curComb* contains the effects of its operation, by Lemma 1, and that the read operation always executes on the object referenced by *curComb*.

▶ **Theorem 1.** *The CX universal construct provides linearizable operations.*

Proof. Follows from Lemma 2, 3, 4 and 5. □

Processes calling `applyUpdate()` are imposed a FIFO linearizable order by the wait-free queue, forcing each process to see the mutations to be applied in the same global order, and if needed, apply these mutations on its local data structure copy `Combined.obj`. This means that the linearization point from writers to writers is the enqueuing in the wait-free queue on line 16. For writers to readers, the mutation becomes visible when *published* in `curComb`. As such, the linearization point from writers to readers is the CAS in `curComb` on line 46, which makes the mutation visible to readers on the load of line 5.