Technical Report - A Wait-Free Universal Construct for Large Objects

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

Abstract

Several wait-free universal constructs have been proposed in the literature, with most of them executing a copy of the entire state for every mutation on an object. 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 CX construct uses a wait-free queue where each node stores a mutative operation that is to be applied on the object. The construct maintains a bounded number of instances of the object that can potentially be brought up to date. Each of the multiple instances is protected with a scalable reader-writer lock with strong trylock guarantees, thus providing mutual exclusion and safe memory reclamation for the underlying object.

We applied CX to several sequential implementations of data structures 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. The CX construct excels in situations where the size of the data structures makes copies impractical, providing scalable throughput for read-mostly applications.

By simultaneously providing wait-free progress, safe memory reclamation and high reader scalability, CX is the first practical wait-free universal construct capable of being deployed in production settings.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms

Keywords and phrases Concurrent algorithms; wait freedom; parallel computing.

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 [21], CLH lock [19] and Tidex lock [28] 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 like *priority inversion*, *convoying* or *deadlock* [14]. Yet, their main drawback comes from their lack of scalability and suboptimal use of the processing capacity of multi-core systems. 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 [13], without any modification or annotation to the sequential implementation. It supports at least one 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. Although not mentioned in the literature, 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 [14]. 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 was sequential. 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. STMs instrument the loads and stores on the sequential implementation, called *load-transactional* and *store-transactional* respectively. STMs may also require type annotation, function annotation or replacement of allocation/deallocation calls with equivalent methods provided by the 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 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. Finally, to the best of our knowledge, there is currently no STM with wait-free progress.

In this paper, we focus on UCs with the goal of addressing their main limitations in terms of performance and usability, which made them so far unpractical for real-world applications. We introduce CX, a 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* [16] nature—unlike most UCs that serialize read-only operations along with updates.

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 object reference counting (ORCs) with hazard pointers. Moreover, CX is the first wait-free UC with a linear bound on memory usage. *(iii)* The first portable implementation of the PSim UC, 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 before proving its correctness in §4. We provide an in-depth evaluation of CX in §5 and finally conclude in §6.

2 Related Work

In 1983, Peterson [27] 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 [20] proposed the first wait-free UC for any number of threads. 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, 7, 11, 12]. Some researchers have attempted to address the problem of applying wait-free UCs to large objects [1, 2, 20] though none has succeeded in providing a generic solution [31].

Anderson and Moir [2] have proposed a technique designed to work well with large objects. Unfortunately, this is not truly a 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. Not only does this goes against the idea of requiring no adaptation or annotation of the sequential implementation, but this technique is also not universally applicable to all data structures.

Chuong et al. [7] 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. It is unclear how could such a word-based approach be adapted to create a true UC, if at all possible.

Fatourou and Kallimanis [12] have designed and implemented P-Sim, a highly efficient wait-free UC based on fetch-and-add and LL/SC. P-Sim relies on an up-to-date instance that all threads copy from, applying all the operations currently being requested by concurrent threads, independently of being a read or write operation. In the best-case scenario, one copy of the entire object state is done per N concurrent operations, where N is the number of threads. In the worst-case, two copies are done per operation. In any case, P-Sim is impractical for large objects. We have made an optimized implementation of P-Sim we named P-SimOpt that separates read-only operations from update operations, so as to improve read operation scalability, and added wait-free memory reclamation using hazard pointers [23].

More recently, Ellen et al. [11] 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`. These specific implementation constraints make the technique not universally applicable. No implementation has been made publicly available.

3 CX Algorithm

We first introduce in this section the basic principle of the CX construct, before presenting in detail the actual algorithm and discussing its properties.

3.1 Data Structures and Operating Principle

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.[1]

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)`. The stack stores its element 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 $\perp$ of CX's wait-free queue of mutations (Algorithm 1, lines 14–15). The other instances have both `obj` and `head` set to null.

To improve *readers* performance, CX has a distinct code path 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`$+2\times$`numUpdaters` trials, as will be discussed later).

[1] For the sake of accuracy and completeness, we show the actual C++ code in the algorithms of this paper instead of informal pseudo-code.

Algorithm 1 CX data structures

```
template<typename C, typename R = bool>
class CX {
    const int maxThr;
    std::atomic<Combined*> curComb {nullptr};
    std::function<R(C*)> mut0 = [](C* c) { return R{}; };
    Node* sentinel = new Node(mut0, 0);
    std::atomic<Node*> tail {sentinel};
    Combined* combs;

    CX(C* inst, int nThr) : maxThr{nThr} {
        combs = new Combined[2*maxThr];
        for (int i = 0; i < maxThr; i++) {
            preRetired[i] = new CircularArray<Node>(hp, i);
        }
        combs[0].head = sentinel;
        combs[0].obj = inst;
        combs[0].rwLock.handoverLock();
        sentinel−>refcnt.store(1, memory_order_relaxed);
        curComb.store(&combs[0]);
    }

    ~CX() {
        for (int i=0; i<2*maxThr; i++) delete combs[i].obj;
        for (int i=0; i<maxThr; i++) delete preRetired[i];
        delete[] combs;
        delete sentinel;
    }

    struct Node {
        std::function<R(C*)> mutation;                 // Not a pointer
        std::atomic<R> result;                         // Not a pointer
        std::atomic<Node*> next {nullptr};
        std::atomic<uint64_t> ticket {0};
        std::atomic<int> refcnt {0};                   // Memory reclamation
        const int enqTid;                              // Used internally by queue
        Node(auto& mutFunc, int tid) {
            mutation = mutFunc;
            enqTid = tid;
        }
    };

    struct Combined {
        Node* head {nullptr};
        C* obj {nullptr};
        StrongTryRWRI rwLock {maxThr};

        void updateHead(Node* mn) {
            mn−>refcnt.fetch_add(1);
            if (head != nullptr) head−>refcnt.fetch_add(−1);
            head = mn;
        }
    };
};
```

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 `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`$+2\times$`numUpdaters` (with a global maximum of $2\times$`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. 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`.

The definition of a valid copy of the object corresponds to 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. As will be later shown, an invalidation of a copy occurs when there is memory reclamation of the queue's nodes.

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 processes 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`

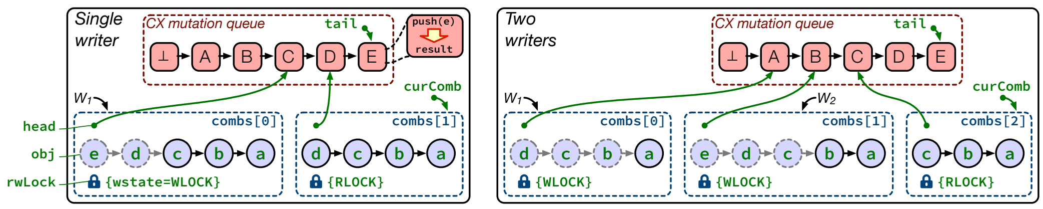

Figure 1 Illustration of CX's principle on two scenarios with one (left) and two (right) writers pushing elements `a` through `e` in a shared stack.

points to `combs[1]`, which holds an up-to-date stack (will 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.

3.2 Algorithm Walkthrough

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` with the desired mutation and insert it in the queue (lines 2 and 4).
2. Acquire an exclusive lock on one of the `Combined` instances in the `combs[]` array (lines 7 to 12).
3. Verify if there is a valid copy of the data structure in the `Combined` instance and make a copy if necessary (lines 20 to 33).
4. Apply all mutations starting at `head` of the `Combined` instance until reaching the `Node` inserted in the first step (lines 19 to 40), and update `head` to point to this node (line 41).
5. Compare-and-set (CAS) `curComb` from its current value to the just updated `Combined` instance (line 52). Upon failure, retry CAS until successful or until `head` of the current `curComb` instance is *after* the newly inserted `Node` in the queue.

When applying a mutation to the underlying object, the first step is to create a new node with the mutation (line 2 of Algorithm 2) and insert it in CX's queue. Each node contains a `mutation` field, of type `std::function`, that stores the mutation. A monotonically increasing `ticket` is assigned to the node to uniquely identify the mutation (line 5).

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 8). The StrongTryRWRI [8] reader-writer lock provides a *strong* `exclusiveTryLock()` method, guaranteeing that the lock will be acquired in at most 2×`maxThreads` attempts.

If the locked `Combined` instance has an invalidated or null `obj` (line 20), we need to make a copy of the current object. To do so, we first acquire the shared lock on `curComb` (line 22), before updating `head` (line 28) and copying `obj` (line 30). It is worth mentioning that copy-on-write (COW) based techniques usually do 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.

Algorithm 2 CX algorithm

```
1   R applyUpdate(std::function<R(C*)>& updateF, int tid) {
2      Node* myNode = new Node(updateF, tid);
3      hp.protectPtrRelease(kHpMyNode, myNode, tid);
4      enqueue(myNode, tid);
5      const uint64_t myTicket = myNode−>ticket.load();
6      Combined* newComb = nullptr;
7      for (int i = 0; i < 2*maxThreads; i++) {
8         if (combs[i].rwLock.exclusiveTryLock(tid)) {
9            newComb = &combs[i];
10           break;
11        }
12     }
13     Node* mn = newComb−>head;
14     if (mn != nullptr && mn−>ticket.load() >= myTicket) {
15        newComb−>rwLock.exclusiveUnlock();
16        return myNode−>result.load();
17     }
18     Combined* lcomb = nullptr;
19     while (mn != myNode) {
20        if (mn == nullptr || mn == mn−>next.load()) {
21           if (lcomb != nullptr ||
22             (lcomb = getCombined(myTicket, tid)) == nullptr)
23              if (mn != nullptr) newComb−>updateHead(mn);
24              newComb−>rwLock.exclusiveUnlock();
25              return myNode−>result.load();
26           }
27           mn = lcomb−>head;
28           newComb−>updateHead(mn);
29           delete newComb−>obj;
30           newComb−>obj = new C(*lcomb−>obj);
31           lcomb−>rwLock.sharedUnlock(tid);
32           continue;
33        }
34        Node* lnext = hp.protectPtr(kHpHead, mn−>next.load(), tid);
35        if (mn == mn−>next.load()) continue;
36        auto res = lnext−>mutation(newComb−>obj);
37        lnext−>result.store(res, memory_order_relaxed);
38        hp.protectPtrRelease(kHpNext, lnext, tid);
39        mn = lnext;
40     }
41     newComb−>updateHead(mn);
42     newComb−>rwLock.downgradeToHandover();
43     for (int i = 0; i < maxThreads; i++) {
44        lcomb = curComb.load();
45        if (!lcomb−>rwLock.sharedTryLock(tid)) continue;
46        if (lcomb−>head−>ticket.load() >= myTicket) {
47           lcomb−>rwLock.sharedUnlock(tid);
48           if (lcomb != curComb.load()) continue;
49           break;
50        }
51        Combined* t = lcomb;
52        if (curComb.compare_exchange_strong(t, newComb)) {
53           lcomb−>rwLock.handoverUnlock();
54           Node* node = lcomb−>head;
55           lcomb−>rwLock.sharedUnlock(tid);
56           while (node != mn) {
57              Node* lnext = node−>next.load();
58              preRetired[tid]−>add(node);
59              node = lnext;
60           }
61           return myNode−>result.load();
62        }
63        lcomb−>rwLock.sharedUnlock(tid);
64     }
65     newComb−>rwLock.handoverUnlock();
66     return myNode−>result.load();
67  }

68  R applyRead(std::function<R(C*)>& readF, int tid) {
69     Node* myNode = nullptr;
70     for (int i=0; i < MAX_TRIES + maxThreads; i++) {
71        Combined* lcomb = curComb.load();
72        if (i == MAX_TRIES) {
73           myNode = new Node(readFunc, tid);
74           hp.protectPtr(kHpMyNode, myNode, tid);
75           enqueue(myNode, tid);
76        }
77        if (lcomb−>rwLock.sharedTryLock(tid)) {
78           if (lcomb == curComb.load()) {
79              auto ret = readF(lcomb−>obj);
80              lcomb−>rwLock.sharedUnlock(tid);
81              return ret;
82           }
83           lcomb−>rwLock.sharedUnlock(tid);
84        }
85     }
86     return myNode−>result.load();
87  }

88  Combined* getCombined(uint64_t myTicket, int tid) {
89     for (int i = 0; i < maxThreads; i++) {
90        Combined* lcomb = curComb.load();
91        if (!lcomb−>rwLock.sharedTryLock(tid)) continue;
92        Node* lhead = lcomb−>head;
93        uint64_t lticket = lhead−>ticket.load();
94        if (lticket < myTicket && lhead != lhead−>next.load())
95           return lcomb;
96        lcomb−>rwLock.sharedUnlock(tid);
97        if (lticket >= myTicket && lcomb == curComb.load())
98           return nullptr;
99     }
100    return nullptr;
101 }
```

Next, we apply the mutations on `obj`, starting from the corresponding `head` node (line 13) until our newly added node is found (line 19), always saving the result of each mutation in the corresponding `node.result` (line 37). 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 16, 25, 61 and 66) 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. In turn, this also implies that `R` must fit in a `std::atomic` data type to ensure wait freedom—a limitation common to all other wait-free universal constructs.

After the mutations have been applied, we advance `curComb` with a CAS (line 52) 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`. Switching between `Combined` instances using a CAS may seem prone to ABA problems at first. By carefully considering the algorithm, one can observe that such issues cannot actually occur. The reason is that CAS is performed between two `Combined` instances for which we are holding the locks in shared mode, and `newComb`'s lock was in fact downgraded from an exclusive lock. This guarantees that before the CAS is executed, `lComb` cannot be re-used as a `newComb` by another thread because this would imply the other thread could acquire the lock of `lComb` in exclusive mode (line 8), which is impossible because the current thread is holding it in shared mode (line 45). This allows CX to avoid creating new `Combined` instances on every operation [4], or to rely on sequences to guarantee a unique identifier for every successful CAS [12]. In addition, `curComb` always transitions to a `Combined` instance with a `head.ticket` higher than the previous one thanks to the test on line 46. This guarantees that operations for 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 `newComb`, before returning the result of our mutation.

3.3 Read-Only Operations

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 71 and the call to `sharedTryLock()` on line 77, 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, but it is not required to apply all mutations up to its own. 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. This approach is similar to the *fast-path-slow-path* methodology by Kogan and Petrank [18].

It is interesting to note that an updater thread processing a read operation may not arrive to the same result as the reader that requested it. The updater will execute the read operation after applying all preceding mutations, which may not correspond to the same result as applying the read operation in the most up to date instance. This is still a correct behavior, given that any of the results will only be returned during the execution of the reader thread. In case a reader thread finishes the read operation before any updater has stored the result of that same read operation, that result may not be linearizable, however, the reader thread requesting it has already left without using it.

3.4 Reader-Writer Lock with Strong Trylock

The CX construct is composed of a finite number of `Combined` instances, each potentially containing a copy of the object. The access to each `Combined` instance and, consequentially, 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]. This is a strong guarantee that few reader-writer lock algorithms provide.

Not only is this strong guarantee needed, but it is also desirable to have a reader-writer lock with good scalability for read-mostly workloads, so as to provide high throughput in the `applyRead()` method where the main bottleneck is the call to `sharedTryLock()`. 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 [6] 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 (U); shared (S), *i.e.*, read-only; exclusive (X), *i.e.*, read-write; or handover (H). 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. The `rwlock` implementation allows the unlock of the shared mode by a different thread from the one which acquired the lock in shared mode. Our extension of StrongTryRWRI provides three new methods to manage handover: `handoverLock()`, `handoverUnlock()` and `downgradeToHandover()`.

3.5 Memory Reclamation

CX relies on a wait-free queue that imposes a sequential order for all the mutations on an object. Each thread will apply that sequence of operations to a copy of the object in isolation. The wait-free queue is necessary for CX to maintain consistency between concurrent threads but it has an impact in memory allocation and its subsequent memory reclamation. Several wait-free queues have been proposed recently (*e.g.*, [17, 12, 30]) and, in our implementation, we use a variant of [30] notably because it provides a very simple `enqueue()` method and supports wait-free memory reclamation.

Every `Combined` instance contains a pointer to a copy of the object `obj` and a corresponding `head` that references a node in the mutation queue. If the node is disposed of (or equivalently "self-linked") during memory reclamation, this will invalidate the copy and force a future updater thread to make a new copy. This can have a negative impact in terms of performance on medium to large data structures. On the other hand, it can also be costly to apply all the mutations necessary to bring a very old copy up to date. There is an inherent trade-off in collecting the nodes of the queue and the consequent invalidation of existing copies of the object. The relative cost of applying a mutation, or deciding for the invalidation and subsequent full copy, is at the core of the design of our memory reclamation.

Access to each object instance is regulated by the reader-writer lock in the respective `Combined` instance. Consequently, invalidated objects are deleted by the next updater thread that acquires the exclusive lock, without need for a memory reclamation technique. The `Combined` instances in the `combs[]` array are themselves deleted only by CX's destructor and, therefore, require no memory reclamation.

The only objects in CX requiring a concurrent memory reclamation technique to track their lifetime are the nodes in the mutation queue. There are peculiarities regarding memory reclamation of the queue's nodes. The most complex to solve is related with the nodes referenced by the `head` of each `Combined` instance. When a new copy of the data structure is needed for a given `Combined` instance, it will be copied from a pre-existing instance tied to one of the other `Combined` instances (the `curComb`) together with its associated `head` pointer. This means that a simple scan of the `heads` of all the `Combined` instances in `combs[]` would result in an erroneous memory reclamation scheme, because while a thread is verifying the possible deletion of a node, a copy of its pointer may be made on the opposite direction of the scan, to a previously scanned `head`, thus leading to the deletion of a node that is still being referenced.

A solution to this specific problem is to use reference counting. Every node of the queue has a `refcnt` variable that stores the number of references that the heads of all `Combined` instances have to that specific node. When a copy is being made, the `refcnt` of the new head will be incremented and the `refcnt` of the previous head of that `Combined` instance decremented, if the previous head is non-null (see `updateHead()` in Algorithm 1). Because the copy procedure of a data structure requires that the updater thread first acquires a shared lock on the instance that it will copy from, this guarantees not only a consistent copy, but also that no other updater can be working on that copy, which in turn ensures that the `refcnt` for the node of the corresponding `head` is greater than zero and consequently that the node is not deleted. Notice that our usage of the `Node.refcnt` variable is non-standard. In typical reference counting techniques, there is an atomic counter that keeps track of the number of *threads* which may hold references to a given object/node. In our technique, the atomic counter `refcnt` keeps track of how many other *objects* have pointers to this object/node. We refer to this approach as *object reference counting* (ORC) to distinguish it from regular reference counting.

CX's memory usage is proportional to the number of `Combined` instances and, therefore, linear with the number of threads, while on other UCs the memory usage is determined by the bound of the memory reclamation technique. Hazard pointers have a memory usage whose bound is quadratic with the number of threads, while other lock-free techniques have even higher bounds on memory usage. This implies that UCs that use such techniques for the underlying objects have a quadratic or higher bound on memory usage.

4 Correctness

We discuss the correctness of CX using standard definitions and notations for linearizability [15], 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* $op1.res <_{hb} op2.inv$ *then* $op1 <_{hb} op2$.

▶ **Definition 2.** *(Subhistory) Given a history H, a subhistory S of H is such that if $op2$ belongs to S and $op1 <_{hb} op2$ in H, then $op1$ belongs to S and $op1 <_{hb} op2$ in S.*

▶ **Definition 3.** *(Partial Subhistory) Given a history H and $op1$, $op2$ update operations, a partial subhistory S of H is such that if $op2$ belongs to S and $op1 <_{hb} op2$ in H, then $op1$ belongs to S and $op1 <_{hb} op2$ 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*
L1 : complete(H') is equivalent to S
L2 : if an operation $op1 <_{hb} op2$ 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 consider 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 previously 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 52 between two `Combined` instances, referred as $lComb$ and $newComb$. The lock associated with the $newComb$ instance is acquired in exclusive mode in line 8 and is later downgraded to shared mode in line 42. 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 52, 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×`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 8 and a shared lock at line 22 or 45. 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 precedent `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 $2 \times (\texttt{maxThreads} - 1)$ `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

$2 \times (\texttt{maxThreads} - 1)$ `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 8 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 19 to 40, 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 $k = j + 1, ..., l$, 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 41, $Comb_i$ 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 42. ◀

▶ **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 52) if $head_{i,j}$ does not satisfy the condition at line 46. 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 16, 25, 61 or 66). After the exclusive lock is acquired for the `Combined` instance $Comb_k$ with state $<O_{k,m}, head_{k,m}>$ at line 14, 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 16 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 21 to 32 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 (line 98); 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 $j + \texttt{maxThreads} \leq m$. Assuming no operation can return before ensuring its operation is visible at *curComb* then $l \leq j + \texttt{maxThreads}$, implying that $l \leq m$. This proves that after `maxThreads` transitions of *curComb* it must contain the update operation with sequence l.

On line 61 the update operation returns after ensuring that *curComb* references a `Combined` instance with sequence number lower than l (line 46) 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 66, *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 $op_r <_{hb} op_u$, 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 $op_u <_{hb} op_r$ 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 enqueueing in the wait-free queue on line 4. 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 52, which makes the mutation visible to readers on the load of line 71.

4.1 Wait-Freedom

The `applyUpdate()` method has only one loop where the number of iterations is not predetermined (line 19). 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 $Comb_i$. For the process to execute the loop at line 19, $Comb_i$'s' state must be $<O_{i,j}, head_{i,j}>$ where $l > j$, otherwise `applyUpdate()` would return at line 16. 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 20). In case the process is unable to do a copy, it will return at line 25, 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 30. 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 19 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 4 and the trylock methods, `exclusiveTryLock()`, `exclusiveUnlock()`, `sharedTryLock()`, `sharedUnlock()` and `downgrade()`. By definition, all these methods return in a finite number of steps, from which we conclude that `applyUpdate()` has wait-free progress.

The `applyRead()` method iterates for maximum of `MAX_TRIES` + `maxThreads`. It calls the `sharedTryLock()` and `sharedUnlock()` methods, which by definition return in a finite number of steps, thus resulting in wait-free progress for `applyRead()`.

5 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.

Besides CX, we used two other UCs which we now describe. `PSim` is a UC with wait-free progress. We adapted the original implementation available on github, and added memory reclamation using hazard pointers (HP) [23]. `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 added a modified version of CX, called CXTimed, which restricts the amount of available `Combined` instances to four for a period of time. In our implementation the amount of time is how much time it takes to do a copy of the object. 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 reduces even further the amount of object copies during program execution.

Depending on the benchmark, we also compared with commonly available lock-free data structures. `MagedHP` is a Harris linked list set modified by Michael [22], using HP as memory reclamation [23]. `MagedHE` is the same lock-free set, using hazard eras (HE) as memory reclamation [29]. `Natarjan` is the balanced tree by Natarajan and Mittal [25], using HE. `MagedHash` is the hash table by Michael [22], using HP.

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 sizes. Each run takes 20 seconds, where a data point corresponds to the median of 5 runs. All implementations will be available publicly, with a simple usage example [9].

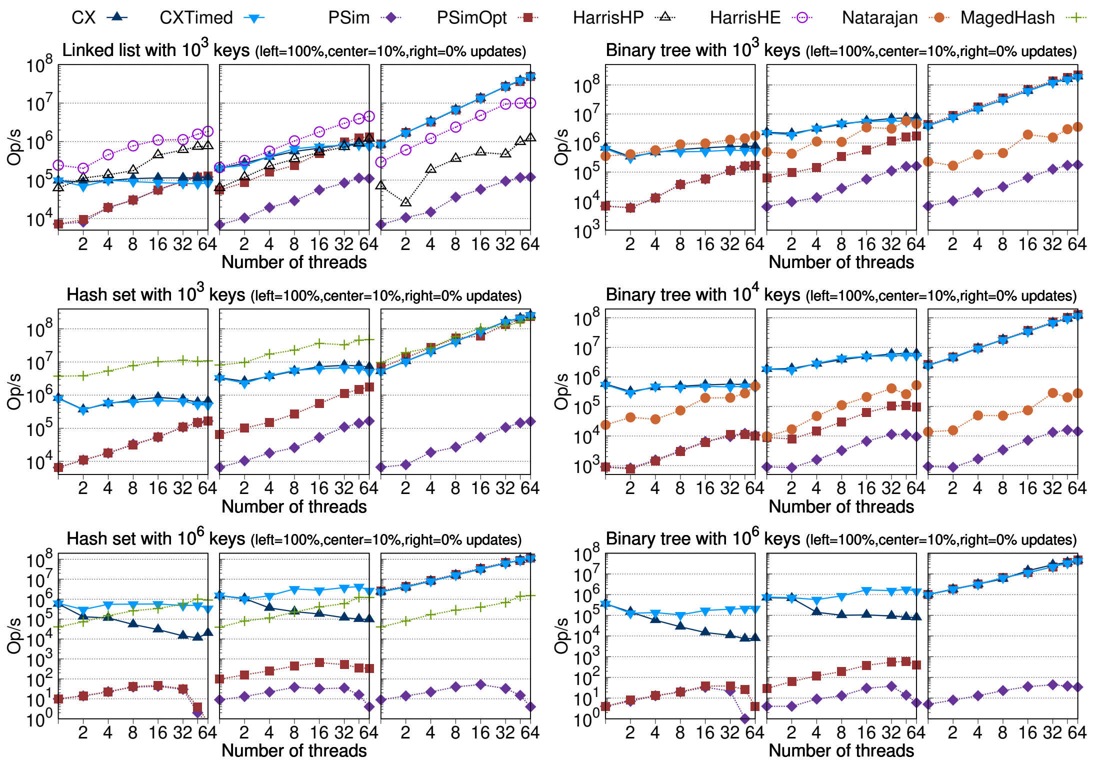

Figure 2 Left column (top to bottom): sets implemented using a linked list with 10^3 keys and using a hash table with 10^3 and 10^6 keys. Right column: sets implemented using a binary search trees with 10^3, 10^4 and 10^6 keys. The results are presented with a logarithmic scale on both axis.

The results of our experiments are shown in Figure 2, 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 with HE 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 with HP 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.

In our experiments, CX can even outperform Maged-Harris with HE in a 1% updates scenario (not shown).

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 access parallel. 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 2. Natarajan's tree is not shown for 10^6 keys because it takes hours to fill up, making it unsuitable for such a scenario. 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 increases, 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 but, in order to achieve that, an optimized lock-free implementation is required and it must use an efficient lock-free memory reclamation technique, such as HE, which can be up to $5\times$ faster than HP.

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 truly universal 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`.

6 Conclusion

Over the past three decades, many non-blocking concurrent data structures have been designed, from simple lock-free stacks [32] to sophisticated wait-free binary search trees [26]. The design and implementation of *correct* non-blocking data structures imposes a significant challenge for their designers, with several published algorithms having been found later to contain errors [33, 24, 10, 5]. The appeal of generic techniques like wait-free universal constructs (UC) stems from this difficulty in proving and validating *hand-written* non-blocking data structures. These constructs are capable of transforming any sequential implementation of a data structure into a correct wait-free data structure, with linearizable consistency for *all* operations, even iterators, which are notoriously difficult to do for hand-written non-blocking data structures.

CX, and particularly CXTimed, allows 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. CX shows that even in scenarios with 100% update operations, the performance results are encouraging for complex data structure such as binary search trees. Moreover, CX has integrated wait-free memory reclamation, a feature that most hand-written lock-free data structures do not provide. The secret behind this huge leap in performance from previous UCs is the significant reduction of copy operations, where available copies are instead reused and updated.

References

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

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

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

4 Maya Arbel-Raviv and Trevor Brown. Poster: Reuse, don't recycle: Transforming algorithms that throw away descriptors. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 429–430. ACM, 2017.

5 Sebastian Burckhardt, Rajeev Alur, and Milo MK Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *ACM SIGPLAN Notices*, volume 42, pages 12–21. ACM, 2007.

6 Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J Marathe, and Nir Shavit. Numa-aware reader-writer locks. In *ACM SIGPLAN Notices*, volume 48, pages 157–166. ACM, 2013.

7 Phong Chuong, Faith Ellen, and Vijaya Ramachandran. 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*, pages 335–344. ACM, 2010.

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

9 Andreia Correia, Pedro Ramalhete, and Pascal Felber. CX. https://github.com/pramalhe/CX, 2018.

10 Simon Doherty, David L Detlefs, Lindsay Groves, Christine H Flood, Victor Luchangco, Paul A Martin, Mark Moir, Nir Shavit, and Guy L Steele Jr. Dcas is not a silver bullet for nonblocking algorithm design. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 216–224. ACM, 2004.

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

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

13 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1): 124–149, 1991.

14 Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

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

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

17 Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *ACM SIGPLAN Notices*, volume 46, pages 223–234. ACM, 2011.

18 Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In *ACM SIGPLAN Notices*, volume 47, pages 141–150. ACM, 2012.

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

20 A Maurice Herlihy. Methodology for implementing highly concurrent data objects. In *Proceedings of ACM PPoPP*, pages 197–206, 1990.

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

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

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

24 Maged M Michael and Michael L Scott. Correction of a memory management method for lock-free data structures. Technical report, ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE, 1995.

25 Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *ACM SIGPLAN Notices*, volume 49, pages 317–328. ACM, 2014.

26 Aravind Natarajan, Lee H Savoie, and Neeraj Mittal. Concurrent wait-free red black trees. In *Symposium on Self-Stabilizing Systems*, pages 45–60. Springer, 2013.

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

28 Pedro Ramalhete and Andreia Correia. Tidex: a mutual exclusion lock. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 52. ACM, 2016.

29 Pedro Ramalhete and Andreia Correia. Brief announcement: Hazard eras-non-blocking memory reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 367–369. ACM, 2017.

30 Pedro Ramalhete and Andreia Correia. 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*, pages 453–454. ACM, 2017.

31 Michel Raynal et al. Distributed universal constructions: a guided tour. *Bulletin of EATCS*, 1(121), 2017.

32 R Kent Treiber. *Systems programming: Coping with parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

33 JD Valois. Errata. lock-free linked lists using compare-and-swap. *Unpublished manuscript*, 1995.