COWMutationQ - A Copy-On-Write technique with wait-free progress

Andreia Correia
Concurrency Freaks
andreiacraveiroramalhete@gmail.com

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

Abstract

Several concurrency primitives are known that can provide concurrent access while allowing the code to be written as if it was single-threaded, with the most common of these being Mutual Exclusion Locks and Reader-Writer Locks, both of which have Blocking progress conditions.

A less used pattern is known as the Copy-On-Write (COW) technique, which requires the protected resource to be *copyable*, and provides Wait-Free Population Oblivious (WFPO) progress for read-only access. Two variants of COW are know, which we name COWLock and COWCAS. In COWLock the mutative operations (writes) are protected by a mutual exclusion lock and then the global reference is updated atomically, thus providing Blocking progress for writes and WFPO progress for reads. In COWCAS the mutative operations change the global reference using a Compare-And-Set (CAS) instruction, thus providing Lock-Free progress for writes and WFPO progress for reads.

We propose a third technique which we named COWMutationQ, where the mutative operations are placed in a Wait-Free queue and are then applied sequentially. This pattern provides Linearizable consistency and Wait-Free progress for writes, while maintaning WFPO progress for reads, making it the first practical generic technique to provide complete wait-free access to an object or data structure, as long as they are cloneable in the case of the object, or shallow-copyable in the case of a data structure.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Mutual Exclusion

General Terms Algorithms, Design, Performance

Keywords Copy-On-Write, wait-free, mutual exclusion

1. Introduction

Although many concurrent data structures are known with lock-free and wait-free progress conditions, many developers still prefer to use generic primitives likes Mutual Exclusion Locks and Reader-Writer Locks to protect concurrently accessible resources. These two primitives provide access with Blocking progress condition, with some algorithms like the Ticket Lock, CLH Lock and Tidex Lock providing Starvation Freedom (only on x86), while others do not, like the Test-and-set lock.

Recently, two new generic primitives have been discovered that can be applied for resources that are duplicable, namely, Double Instance Locking [6, 7], and the Left-Right pattern [8]. Double Instance Locking (DIL) is composed of two Reader-Writer Locks and one Mutual Exclusion Lock, and can provide Lock-Free access for read-only operations, while still being Blocking for writes. Even when using Locks that are Starvation-Free, this technique does not provide Starvation-Freedom for read operations. The Left-Right pattern allows for Wait-Free Population Oblivious (WFPO) read-only access if a suitable Read-Indicator is used, but it's Blocking for writes. The Left-Right pattern is inherently Starvation-Free and can provide full Starvation-Freedom when a Starvation-Free Lock and Wait-Free Read-Indicators are used. Both the Double Instance Locking and the Left-Right techniques require the usage of two equal resources (duplicates), which is trivial to accomplish for objects in memory, but non-trivial or even impossible for physical resources.

1.1 Copy-On-Write and its variants

Another generic technique is the Copy-On-Write (COW) which is more common in languages with a Garbage Collector (GC) and functional languages. The idea behind COW is that each thread wishing to do a write/mutation, must create a copy of the current resource (logical or physical) and apply the mutations to its own copy, but read-only operations can just follow the current reference to the most recent (stable) copy. COW comes in two variants which we will refer to as COWLock and COWCAS.

In COWLock, a mutual exclusion lock is used to protect modifications to the main reference, thus guaranteeing that one and only one thread at a time can be copying the current instance and applying the mutations to its own copy, before updating the reference with the newly modified instance. An example implementation in Java of COWLock can be seen in Algorithm 1 as class `COWLock`, and a practical implementation in Java JDK's java.util.concurrent.CopyOnWriteArrayList [3].

In COWCAS, multiple threads may simultaneously perform a copy of the current instance, and each one apply a different mutation to its own copy, and then attempt to change with a `compareAndSet()` (CAS) the reference to the current instance to point to its own instance. the threads for which the CAS fails, must read again the *new* current reference, make a copy of it, and apply the mutation to its copy. This procedure could go on indefinitely with consecutive failures, thus starving one or multiple threads, but this is a consequence of the Lock-Free progress condition for writes. The starvation effect can become particularly nasty if there are multiple mutative methods being called with different execution times, in which case the ones with the longest time to complete will be the most likely to be starved. An example implementation in Java of COWCAS can be seen in Algorithm 1 as class `COWCAS`.

Algorithm 1 Copy-On-Write patterns: COWLock and COWCAS

```
 1  class COWPattern<T,R,C> {
 2      final AtomicReference<C> ref = new AtomicReference<C>(null);
 3
 4      public <R> R applyRead(T param1, BiFunction<T,C,R> readOnlyFunc) {
 5          return readOnlyFunc.apply(param1, ref.get());
 6      }
 7  }
 8
 9  class COWLock<T,R,C> extends COWPattern {
10      final ReentrantLock mutex = new ReentrantLock();
11
12      public <R> R applyWrite(T param1, BiFunction<T,C,R> mutativeFunc) {
13          mutex.lock();
14          try {
15              C newInst = ref.get().clone();
16              R retVal = mutativeFunc.apply(param1, newInst);
17              ref.set(newInst);
18              return retVal;
19          } finally {
20              mutex.unlock();
21          }
22      }
23  }
24
25  class COWCAS<T,R,C> extends COWPattern {
26      public <R> R applyWrite(T param1, BiFunction<T,C,R> mutativeFunc) {
27          while (true) {
28              C oldInst = ref.get();
29              C newInst = oldInstance.clone();
30              R retVal = mutativeFunc.apply(param1, newInst);
31              if (ref.compareAndSet(oldInst, newInst)) return retVal;
32          }
33      }
34  }
```

Table 1 shows a comparison between the different techniques described in this section and their properties in terms of progress conditions. Regarding the progress of read-only operations, all COW based techniques and the Left-Right pattern can provide WFPO progress. As for mutative operations (writes), only the COWCAS and COWMutationQ are non-blocking, with the COWCAS being easily starvable. Notice that all the COW based techniques are meant to be used with a GC or some kind of memory management technique, and although COWLock and COWCAS can be used with some memory management techniques at the detriment of its progress conditions — for example, Hazard Pointers are lock-free, which means that for read-only operations, we will lose the wait-free progress conditions when using them together with one of the COW techniques —, the COWMutationQ is particularly difficult to use with current memory managment techniques like Hazard Pointers [4] or Reference Counting [9].

	Read Progress	Write Progress	Starvation Freedom ?	Needs GC ?
Mutex	Blocking	Blocking	Yes	No
RW-Lock	Blocking	Blocking	Yes	No
DIL	Lock-Free	Blocking	No	No
Left-Right	WFPO	Blocking	Yes	No
COWLock	WFPO	Blocking	Yes	Yes
COWCAS	WFPO	Lock-Free	No	Yes
COWMutationQ	WFPO	Wait-Free	Yes	Yes

Table 1. Comparison table between the different generic techniques that provide concurrent access to an object or data structure. The first two columns indicate the progress conditions of each algorithm, the third column shows whether it is possible to do an implementation of the algorithm that provides freedom from starvation, and the fourth column indicates whether the algorithm can be implemented without a Garbage Collector or a memory management system.

Algorithm 1 shows generic code for the two variants COWLock and COWCAS when used to protect access to an object of type `C`. In the `COWLock` class we proctect mutations to the reference to the `C` instance with a `mutex`, and after creating a copy of the old `C` instance (line 15) and applying the mutation to the new instance (line 16), we then set the reference atomically to the new instance (line 17). In the `COWCAS` class we protect mutations to the reference to the `C` instance with a `compareAndSet()` operation, and after creating a copy of the old `C` instance (line 29) and applying the mutation to it (line 30), we then attempt a `compareAndSet()` on the reference, from the old instance to the new one, which if unsucessful will cause the a new instance to be copied from the current old and thus discarding the previously cloned instance and the mutation.

Both COWLock and COWCAS variants are linearizable [1] for read-only and mutative methods, and both provide Wait-Free Population Oblivious for the read-only methods.

The COW techniques are not particularly scalable, but they are easy to use and to implement, making them a somewhat popular way to provide concurrent access to objects or data structures. It becomes interesting to consider whether a third technique exists that provides wait-free progress conditions for mutative operations without incurring a significant cost in overhead. Such a technique would be interesting not just from a point of view of providing strong guarantees of latency and fairness, but also becausing having a generic technique that gives non-blocking access to any user-created object, has the advantage of letting the software engineers reason about their code as if it was single-threaded, just like when using a mutual exclusion lock or reader-writer lock, thus allowing them to have non-blocking concurrent access to memory-based objects while using familiar concepts, such as locks. In this paper, we present such a technique, which we named `COWMutationQ`.

The COW technique we have developped, makes use of a linearizable wait-free queue to provide linearizable concurrent access to a resource, as long as such a resource is *copyable*, similarly to the COWLock and COWCAS techniques. The idea is to enqueue all mutations in a wait-free queue, like the one devised by Alex Kogan and Erez Petrank [2], and then let each thread make one and only one copy of the current instance and apply each one of the mutations from the queue, sequentially, until the mutation that the thread itself added is applied. When compared with the other two COW techniques, the COWMutationQ has the advantage that a single copy is done of the current instance and then all mutations are applied to that one copy, while the other two techniques require a copy of the instance to be created per mutation. One of the tricks to accomplish this, is to have the *head* node of the queue and the current instance to be accessed atomically, for which we — funnily enough — use a kind of COWCAS technique. Each thread will traverse the queue until it reaches the node with its own node, and will at most, fail the `compareAndSet()` on the global reference the number of nodes between the current and its own node, thus giving it wait-free progress guarantees.

One alternate technique that springs to mind when first looking at COWMutationQ, is an asynchronous queue of operations with Futures, which works based on *delegation*. In such a technique, mutations are added to a queue and a Future is kept, to later access the result of the mutation. If read-only operations are also placed in the queue, then all the operations will be sequentially consistent, but if a result is needed, from either a read-only operation or a mutation, then this technique will be blocking, and because having read-only operations that return nothing is non-sensical, this technique will always be blocking under such conditions. If the read-only operations are not put in the queue and instead are executed on the current instance, then it will be wait-free for the read-only operations, but it will not be sequentially consistent (and therefore, not linearizable), and moreover when the result of a mutation is needed, it will block on the Future holding the result. Because this technique is blocking when implemented in a linearizable way, we chose not to compare it in our benchmarks, seen as it would be only slightly better than using a global mutex for protection.

We present our COWMutationQ variant in detail in Section 2. We show our empirical evaluation in Section 3, and conclude in Section 4.

2. Algorithm

The COWMutationQ technique is composed of the following elements: A queue of mutations where each node, `MutationNode`, stores a function pointer (lambda) to be applied on the object (or data structure) and the arguments with which the function should be called; A type that combines the head (first node) of the queue and the reference to the object, named `Combined`; A single reference to the tail (last node) of the queue. Notice that there are multiple heads but only one tail; A reference to a Combined instance that contains the current head and object references; The class definition for the nodes of the queue `MutationNode` and the `Combined` can be seen in Algorithm 2.

Algorithm 2 Helper classes of COWMutationQ

```
1   public static class MutationNode<T,C> {
2       final BiFunction<T,C,?> mutation;
3       final T param1;
4       final AtomicReference<MutationNode<T,C>> next;
5
6       public MutationNode(T p1, BiFunction<T,C,?> mutativeFunc) {
7           this.mutation = mutativeFunc;
8           this.param1 = p1;
9           this.next = new AtomicReference<MutationNode<T,C>>(null);
10      }
11  }
12
13  public static class Combined<T,C> {
14      final MutationNode<T,C> head;
15      final C instance;
16
17      public Combined(MutationNode<T,C> head, C instance) {
18          this.head = head;
19          this.instance = instance;
20      }
21  }
```

Notice that in line 2 of Algorithm 2 we define the return type of BiFunction as ? because each instance of MutationNode can represent a different mutative operation, and therefore have a different return value, i.e. one instance can be for an insertion, another instance for a removal, yet another for an iteration, and all these methods may have different return types.

The main steps of this technique when doing a mutation can be summarized from Algorithm 3 as:

1. Read the current `Combined` instance so we have a reference to the object and to a corresponding `MutationNode` in the queue (line 18);
2. Create a new `MutationNode` with the desired mutation and insert it in the queue (lines 17 and 21);
3. Clone the object whose reference was obtained from `combinedRef` and start applying to it the mutations in the queue, starting from head until reaching the node this thread inserted in the queue (lines 23 to 30);
4. Do CAS on `combinedRef` from the current value up to a newly created `Combined` with this thread's `MutationNode` and the object with all the mutations up to it. Retry this if it fails until it succeeds or the current instance of `combinedRef` has an `head` that is *after* this thread's `MutationNode` in the queue (lines 34 to 43);

For read-only operations, it's just a matter of reading the current `Combined` instance `combinedRef` and using its reference to the object, as shown in line 13 of Algorithm 3.

Algorithm 3 COWMutationQ

```
1   class COWMutationQ<T,R,C> {
2       final AtomicReference<Combined<T,C>> combinedRef;
3       final AtomicReference<MutationNode> tail;
4
5       public COWMutationQ(C instance) {
6           sentinel = new MutationNode<>(null, null);
7           tail = new AtomicReference<MutationNode>(sentinel);
8                                   newComb = new Combined(sentinel, instance);
9           combinedRef = new AtomicReference<Combined<T,C>>(newComb);
10      }
11
12      public <R> R applyRead(T param1, BiFunction<T,C,R> readOnlyFunc) {
13          return readOnylFunc.apply(param1, combinedRef.get().instance);
14      }
15
16      public <R> R applyWrite(T param1, BiFunction<T,C,R> mutativeFunc) {
17          MutationNode<T,C> myNode = new MutationNode<T,C>(param1, mutativeFunc);
18          Combined<T,C> curComb = combinedRef.get();
19          // Insert our node in the queue. This may be Lock-Free or Wait-Free
20          // depending on which queue algorithm is used.
21          addToTail(myNode);
22
23          // Clone the current instance and apply all mutations up until our node is reached
24          final C mutatedInstance = (C)curComb.instance.copyOf();
25          for (MutationNode<T,C> mn = curComb.head.next.get(); mn != myNode; mn = mn.next.get()) {
26              mn.mutation.apply(mn.param1, mutatedInstance);
27          }
28          // Save the return value of the last mutation (ours).
29          // We don't care about the other return values.
30          final R retValue = mutativeFunc.apply(param1, mutatedInstance);
31
32          // Create a new Combined with all the mutations up to ours (inclusive)
33          // and try to CAS the ref to it until it has our mutation.
34          final Combined<T,C> newComb = new Combined<T,C>(myNode, mutatedInstance);
35          do {
36              if (curComb != combinedRef.get()) {
37                  curComb = combinedRef.get();
38                  MutationNode<T,C> ltail = tail.get();
39                  // Traverse the list until we reach the end, or our own node
40                  for (MutationNode<T,C> mn = curComb.head; mn != myNode; mn = mn.next.get())
41                      if (mn == ltail) return retValue; // Our mutation is visible
42              }
43          } while (!casRef(curComb, newComb));
44          // Our mutation is now visible to other threads (through combinedRef)
45
46          return retValue;
47      }
48  }
```

Another interesting detail about COWMutationQ is that the `combinedRef` is first read to obtain the head and only then do we insert our `MutationNode` in the queue. This guarantees the invariant that the head of the `Combined` instance will always be before our own `MutationNode`. This detail is of vital important because on the last `do/while()` loop in `applyMutation()` we start from that head until we find the our `MutationNode` or until we find the last read tail `ltail`. If we find our node before the current tail, then it means the the current head is *after* our node, which implies that the current `combinedRef` contains an instance that has our mutation already applied to it. We can exit `applyMutation()` with the certainty that our mutation is now visible to other threads. If we find the current tail *before* our node, then it means that our mutation is not yet visible, and we have to retry the CAS on `combinedRef` to make our mutation (and previous ones) visible to other threads. A naive approach might consider doing the `for()` loop in line 40 up to `null` instead of a local tail (`ltail`), but this would unfortunately be lock-free and not wait-free. The reason for it being lock-free is that another thread could add a new node to the queue, execute and do a successful CAS on `combinedRef`, and then add a new node, and so on, while the current thread is traversing the queue, which would theoretically go on for ever.

2.1 Progress Conditions

A method is Wait-Free Bounded [1] if it guarantees that every call finishes its execution in a finite and bounded number of steps, and this bound may depend on the number of threads.

We shall now show that apart from the call to `addToTail()` in line 21 of Algorithm 3 whose progress depends on the queue/list chosen, the method `applyWrite()` is Wait-Free Bounded. To do this, we will show that the method `applyWrite()` will always complete in a finite number of steps.

The `for()` loop in lines 25 and 26 of Algorithm 3 will complete in a finite number of steps because there can only be a finite number of `MutationNode` instances between the current Combined instance `curComb` and `myNode`. However, the number of `MutationNodes` traversed may end up being larger than the number of active threads because one or several threads may introduce multiple nodes between the time it takes another thread to read `combinedRef` and to insert itself in the queue.

The `for()` loop in lines 40 and 41 of Algorithm 3 will traverse the queue up until it finds this thread's `myNode` or the end of the queue, `ltail`, thus completing in a finite number of steps.

The `do/while()` loop starting in line 35 of Algorithm 3 contains a loop that completes in a finite number of steps and itself will complete when `null` is found, or when the `compareAndSet()` is successful in line 43, which will fail at most the number of nodes in the queue between the current Combined and `myNode`, thus completing in a finite number of steps.

An important detail is that the number of steps needed to complete `applyWrite()` can be proportional to the number of threads currently attempting a mutative operation, and as such, the progress condition is Wait-Free Bounded instead of being Wait-Free Population Oblivious.

Notice also that this algorithm is Linearizable [1]. A mutative operation becomes visible to other threads — regardless of them doing read-only operations or mutations — when a successful CAS is done in `combinedRef` (line 43) by the thread that wishing to apply the mutation, or by a thread for which its `MutationNode` was queued *after* the node with the mutative operation in question.

One caveat of COWMutationQ when compared with COWLock and COWCAS, is that the parameters passed to the mutative operation must be (logically) immutable. Suppose for an instant that a thread that is applying the mutation with a given param $p1$ has sucessfully CAS'ed its Combined instance and made its mutation visible to other threads, and so now it can return to the calling method, and after doing so, it changes the contents of $p1$. It could then happen that there is still some thread that is applying the mutations in sequence and is still going to apply the mutation of the previous thread, with the $p1$ parameter, and in doing so, not only apply the wrong parameter, but also break mutual exclusivity. In practice, re-using the parameters passed to such methods is rarely a necessity and even then, a copy can be done of the parameters themselves if needed.

2.2 Lock-Free variant

As shown on the previous section, the COWMutationQ technique can be used with different queues. One possibility is to use a lock-free queue like the one developped by Maged Michael and Michael Scott [5], which is a simple and efficient algorithm. This approach means that mutations will be lock-free, which may seem counterproductive in comparison to using a COWCAS which has the same progress conditions, but notice that COWCAS can be extremely unfair if concurrent threads attempt different mutative operations (with different expected completion times), while on the COWMutationQ with a lock-free queue, the only part of the algorithm that may incur in starvation is the insertion in the queue, which takes the same time irrespective of the mutative operation that is placed in the node, and it is just as (un)likely to starve as inserting items into a ConcurrentLinkedQueue from j.u.c, which is based on the same algorithm. The simplicity of implementation makes this approach attractive, and as we will see on section 3, the throughput is very similar to the wait-free variant.

2.3 Usage Heuristics

Now that we have three different techniques based on Copy-On-Write (COW), it becomes and interesting exercice to determine which one to use under which situations. All three techniques have different progress conditions for mutations, which means that if fairness or strong latency guarantees for mutatitive operations are needed, then the obvious choice is COWMutationQ due to it being wait-free. Yet, there are other factors to take into account, such as throughput under high and low contention, and for that, we can take a look at how long each of the steps of the different algorithms take.

If we define T_C as the time is takes to make a copy of the current instance, T_M the time it takes to apply the mutation, T_Q as the time it takes to create a mutation node and insert it into the queue, and T_L the time needed to lock and unlock the mutual exclusion lock used by COWLock, then we can define the following times for each of the techniques, where N is the number of threads:

- COWLock: $T_{COWLock} = (T_C + T_M + T_L)N/2$;
- COWCAS: $T_{COWCAS} = (T_C + T_M)N/2$ with the time it takes for a `compareAndSet()` being small enough to be neglected by comparison;
- COWMutationQ: $T_{COWMutationQ} = T_C + T_Q + T_M N$;

When the mutation itself is the most time consuming part ($T_M >> T_C, T_Q, T_L$), then all three techniques will have similar results in throughput with a small advantage to COWCAS because it doesn't spend time in the mutex or adding nodes to a queue. If the copy-/clone is the most time consuming part ($T_C >> T_M, T_Q, T_L$), then COWMutationQ will take a total time of about T_C, while COWLock and COWCAS will take about $T_C N/2$, because both COWLock and COWCAS require on average a copy of the instance to be made for every mutation, while on COWMutationQ we need to do a single copy of the instance for about every N competing threads. This means that, assuming T_L and T_Q are small in comparison to T_C and T_M, when $T_M >> T_C$, it is preferable to use COWLock or COWCAS, but if $T_C > T_M$ then COWMutationQ should provide better results, particularly as the number of threads N increases.

3. Experimental Results

In order to compare the different techniques that provide generic concurrent access to an object or data structure, we created a micro benchmark where each of these techniques is used to protect access to a mutable Red-Black TreeSet taken from java.util.concurrent (j.u.c). Although COW techniques are usually associated with immutable data structures, we decided to use a mutable tree, because immutable tree implementations can take an unreasonable amount of time to fill up with thousands of items. Each loop iteration of the micro benchmark consists of randomly selecting either a read-only operation or a read-write operation, depending on the percentage of writes for that particular run. A read-only operation consists in randomly selecting one of the items in the TreeSet and calling `contains(item)`. A read-write operation consists of randomly selecting one of the items in the TreeSet and then removing it from the set and adding it back immediately afterwards with `remove(item)` followed by `add(item)`. In terms of number of operations, the read-write operations are accounted as 2 due to having two separate method calls.

We ran the micro benchmark on a dual AMD Opteron 6272 with a total of 32 cores, running Windows 7 with JDK 8 (u45). Each data point shown in figure 1 is the median of 5 runs, with each run measured over a period of 20 seconds. The COW based techniques have a need for a Garbage Collector (GC), so we decided to use Java as the language to do the comparison on.

We ran our micro benchmarks for the following implementations:

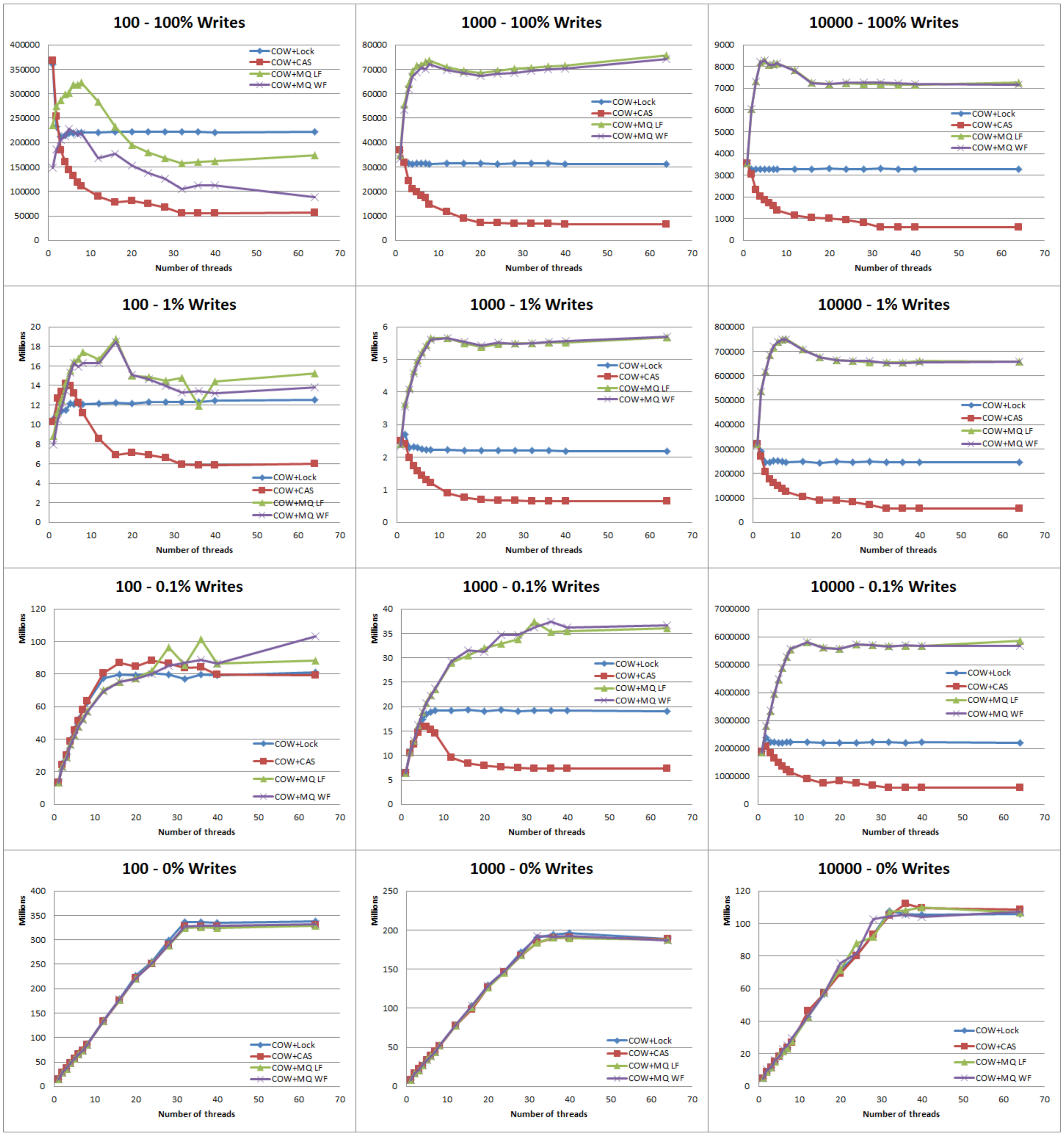

Figure 1. Plots comparing the four different COW techniques, when used to protect a TreeSet with 100, 1000, and 10000 items in the tree respectively. The vertical axis on each plot shows the total number of operations per second (`contains()+add()+remove()`).

- COWLock: TreeSet instances protected with the COWLock technique using a ReentrantLock from j.u.c, as described in Algorithm 1;
- COWCAS: TreeSet instances protected with the COWCAS technique, as described in Algorithm 1;
- COWMutationQ WF: TreeSet instances protected with the COWMutationQ as described in Algorithm 3, where the queue/list is based on the wait-free queue by Kogan and Petrank [2];
- COWMutationQ LF: TreeSet instances protected with the COWMutationQ as described in Algorithm 3, where the queue/list is based on the lock-free queue by Michael and Scott [5];

Overall, the COWMutationQ matches or exceeds the other two COW variants in throughput, with the only exception being for a low number of threads competing to perform mutations on a very small data structure, like the scenario shown in the top leftmost plot in figure 1. If we look at the scenario with 1000 items in the middle column or the one for 10000 items in the tree in the rightmost column of figure 1, we see that the two COWMutationQ variants have better throughput in all scenarios except the read-only workload (0% Writes) where all COW techniques have the same performance, i.e. linear scalability with the number of threads up until the number of cores is reached.

In our microbenchmark we tested with a Red-Black tree which means that the time number of steps to perform a mutation is $O(ln_2(n))$ with n being the number of items in the tree, while the number of steps to copy the tree should be $O(n)$. As shown in section 2.3 the relative time it takes to complete the copy of the instance and the mutation will affect which of the COW methods will give a better overall througput, which means that using them to encapsulate other data structures with different relative ratios of mutation-to-copy, may provide different behaviors than the ones shown in these plots.

3.1 Latency measurements

Using the same machine as on the previous section, we ran a latency measuring benchmark that consisted of creating 16 threads with 8 of them doing only `contains()` and the remaining 8 threads doing `add()` and `remove()` operations. The time taken to complete each individual call to one of these three operations is measured using `System.nanotime()` and saved as an histogram. Table 2 shows the results as a function of the percentage of events that take the indicated amount of time, or less, to complete.

	COW CAS	COW Lock	COW MQ (LF)	COW MQ (WF)
contains() 90%	1	0<	3	3
contains() 99%	6	1	9	8
contains() 99.9%	11	2	14	13
contains() 99.99%	17	9	21	19
add() 90%	341	55	185	182
add() 99%	>1000	74	235	230
add() 99.9%	>1000	>1000	298	285
remove() 90%	347	55	185	182
remove() 99%	>1000	73	235	230
remove() 99.9%	>1000	>1000	298	285

Table 2. Latency values in microseconds for the different COW algorithms. Lower values are better. The two rightmost columns are for the lock-free and wait-free variants of COWMutationQ.

Using the COWCAS as an example, the table can be read as follows: 99% of the calls to the `contains()` method take 7 microseconds or less to complete.

On the first four lines of table 2 we show the results for `contains()`, a read-only operation. In this case, COWLock is the best of all four algorithms, but we should take into consideration that the COWMutationQ are not too far behind, and that this difference is due to COWLock doing less mutative operations, and as such, there will be less cache misses when doing a read-only operation, which gives better times than the COWMutationQ variants.

On the lines for the latencies of `add()` and `remove()`, we can see that the COWMutationQ variants have at least one order of magnitude above COWCAS and COWLock for the 99% and above, as is to be expected due to their progress conditions providing strong maximum latency guarantees. When it comes to the 99.9%, the COWMutationQ with a wait-free queue is, unsurprisingly, the best.

4. Conclusion

This paper presents a new Copy-On-Write based technique which, unlike previous COW techniques, provides a wait-free progress condition for mutations (read-write operations). Similarly to the other COW techniques, this is a generic pattern that can be applied to any resource that is *copyable*, and thus provide wait-free concurrent access to any object or data structure.

Due to the fact that on COWMutationQ the nodes are not dequeued from the queue and we simply let the GC clean up the nodes that are no longer accessible by any thread, it makes this technique notoriously hard to use in systems without GC. Even so, taking into consideration that both COWLock and COWCAS are themselves tricky to use without a GC, this isn't much of a disadvantage for COWMutationQ.

Although our implementation uses lambdas, COWMutationQ can be used in languages with just function pointers, or even by encoding the method as an integer in the MutationNode and then using a case/switch to decide which method to use for each node.

Due to its higher throughput and lower latency for all operations when compared to the other COW techniques, we believe that COWMutationQ is a useful and interesting technique that should be part of every software engineer's toolset for multi-threaded software development.

Acknowledgments

We would like to thank Alex Kogan and Erez Petrank for making the source code to their wait-free queue publicly available, which allowed us to make a faster prototype for the COWMutationQ wait-free implementation.

References

[1] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[2] A. Kogan and E. Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *ACM SIGPLAN Notices*, volume 46, pages 223–234. ACM, 2011.

[3] D. Lea. CopyOnWriteArrayList. `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyO` 2013.

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

[5] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM, 1996.

[6] P. Ramalhete and A. Correia. Double Instance Locking. `http://concurrencyfreaks.com/2013/11/double-instance-locking.html`, 2013.

[7] P. Ramalhete and A. Correia. Double Instance Locking Presentation. `https://github.com/pramalhe/ConcurrencyFreaks/blob/master/Presentat` 2013.

[8] P. Ramalhete and A. Correia. Left-Right: A Concurrency Control Technique with Wait-Free Population Oblivious Reads. Technical report, 01 2013. URL `https://github.com/pramalhe/ConcurrencyFreaks/blob/master/papers/le`

[9] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222. ACM, 1995.