Left-Right: A Concurrency Control Technique with Wait-Free Population Oblivious Reads

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

Andreia Correia
Palagate
andreiacraveiroramalhete@gmail.com

Abstract

In this paper, we describe a generic concurrency control technique with Blocking write operations and Wait-Free Population Oblivious read operations, which we named the Left-Right technique. It is of particular interest for real-time applications with dedicated Reader threads, due to its wait-free property that gives strong latency guarantees and, in addition, there is no need for automatic Garbage Collection.

The Left-Right pattern can be applied to any data structure, allowing concurrent access to it similarly to a Reader-Writer lock, but in a non-blocking manner for reads. We present several variations of the Left-Right technique, with different versioning mechanisms and state machines. In addition, we constructed an optimistic approach that can reduce synchronization for reads.

We applied this technique to the mutable TreeSet implementation of the Java library and compared its performance with: Java's TreeSet protected with a recently developed Reader-Writer lock, named ScalableStampedRWLock; and the SnapTreeMap, a relaxed tree with hand-over-hand optimistic validation from the edu.-stanford.ppl.concurrent package. Microbenchmark experiments show that in a setting with dedicated Reader threads, the Left-Right technique has an improved throughput of up to a factor of 5 compared with the SnapTreeMap.

Categories and Subject Descriptors D.1.3 [*Programming techniques*]: Concurrent Programming

General Terms Algorithms, Design, Performance

Keywords Wait-Free Population Oblivious, Concurrent data structures, real-time, Reader-Writer lock

1. Introduction

Concurrent access to data structures in real-time multi-threaded environments is a challenging problem, mainly because there are few practical non-blocking data structures and techniques, and most of them require some kind of automatic Garbage Collection (GC), a feature that many real-time systems do not have.

A common approach when dealing with the need of allowing concurrent read and write access to a data structure or object, is to use a Reader-Writer lock. Recent developments have improved throughput and scalability for read operations [4, 6, 19, 21]. Although flexible, Reader-Writer locks have some drawbacks, one of them being that Readers — threads doing a read-only operation on the data structure or object — are blocked by Writers — threads which try to modify the data structure or object — meaning that, when a Writer is in the critical section, no Reader will be able to make progress. The blocking progress condition of Reader-Writer locks implies that their usage in real-time systems must be carefully considered, so as to not affect the real-time properties of the application.

Another technique is to use a Copy-On-Write (COW) pattern [14]. It consists of copying the entire object or data structure, applying to the new object the desired modification, and then atomically swapping the reference to the previous object with the newly created one using a CompareAndSet (CAS) instruction. This pattern is a simple approach that can allow Lock-Free writes and Wait-Free Population Oblivious (WFPO) reads [13]. The disadvantage of this approach is that it relies on GC, which hinders the algorithm from being ported to environments without GC [8], and cloning the object or data structure can be lengthy, reducing Writer's performance.

To overcome these limitations, we designed the **Left-Right** pattern, which allows multiple Readers to concurrently execute, regardless of whether or not a Writer is simultaneously running, and does not need a GC. Its main innovation is a new concurrency control algorithm whose novel state machine gives wait-free guarantees for read operations, unlike previously known algorithms [23] which are blocking or lock-free.

The Left-Right technique is an easily implementable concurrency pattern that wraps any data structure or object, providing WFPO read operations. The WFPO progress condition gives a strong guarantee to Reader's latency, a particularly important characteristic when using data structures in real-time systems. This pattern can be applied to any data structure, but it is particularly interesting when applied to balanced trees [1, 11], which guarantee worst-case $O(ln\ n)$ instructions for most of its operations, thus giving it deterministic latency for reads in a concurrent setting. Although it is blocking for Writers, the fact that it has a small overhead for Reader synchronization, makes it ideal for usage in *write-few-read-many* scenarios.

On Table 1 we show a comparison between the three generic concurrency techniques.

The SnapTreeMap [3] is a recent concurrent data structure developed specifically for trees, that allows read and write operations to perform simultaneously, assuming that no rebalancing is taking place, otherwise it is possible for the optimistic hand-over-hand validation to fail and block progression. In addition, it takes advantage

	RW Lock	COW + CAS	Left-Right
Reads block Reads	no	no	no
Reads block Writes	yes	no	yes*
Writes block Reads	yes	no	no
Writes block Writes	yes	no	yes
Needs a GC	no	yes	no
Deterministic Read Latency	no	yes**	yes
Number of instances	1	$N_{Threads}$	2

Table 1. Comparison table between three generic techniques for concurrency control. * On the Left-Right pattern, the Writer may be blocked by older Readers, but once those finish their task, the Writer will be able to make progress. ** On the COW+CAS pattern, the GC can impact the latency.

of a relaxed balance tree, and partially external trees for deletion, which minimizes the tree's rebalance frequency.

2. Design Overview

The Left-Right pattern is a concurrency control technique with two identical objects or data structures, that allows an unlimited number of Readers to access one instance, while a single Writer modifies the other instance. The Writer starts by writing on the right instance (`rightInstance`) while the Readers read the left instance (`leftInstance`), and once the Writer completes the write, the two instances are *switched* and new Readers will read from the `rightInstance`. The Writer will wait for all the Readers still running on the `leftInstance` instance to finish, and then repeat the write on the `leftInstance`.

In this paper, the two identical instances will be mutable TreeSets from the `java.util` package, based on a Red-Black tree [5], and will be refered from now on as `leftTree` and `rightTree`. The read operations will run in the `leftTree` in a non-exclusive mode, while a single Writer modifies the `rightTree`, or vice-versa. Before exiting, the Writer will modify the second data structure, leaving both of them up-to-date. The synchronization between Writers is achieved with an exclusive lock that is used to protect write-access. The write operation has to ensure that Readers are always running on the data structure that is currently *not* being modified. In summary, read operations can run concurrently with all operations, and will *never have to wait* for a Writer or for other Readers.

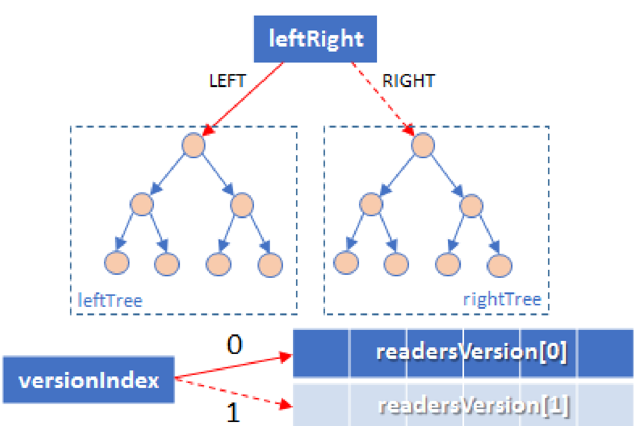

Figure 1. Components of the Left-Right concurrency control.

The components of the mechanism ensuring a Writer performs in exclusivity can be seen in Figure 1 and are the following: a `leftRight` variable which is toggled by the Writer between `LEFT` and `RIGHT`, that indicates which tree the Readers should go into; a `versionIndex` variable, which is also toggled by the Writer between 0 and 1, functioning like a *time tag*; and a Reader's indicator, `readIndicator`, that allows each Reader to *publish* the `versionIndex` it read.

The `readIndicator` is a data structure that provides operations allowing Readers to publish their state, `arrive(versionIndex)` and `depart(versionIndex)`, and for the Writer to determine the presence of Readers, `isEmpty(versionIndex)`. In summary, this is a state publishing data structure.

A possible implementation for the `readIndicator` would be to use two single counters, or two sequentially consistent distributed counters, one per `versionIndex`. Furthermore, it is possible for each counter to be split in two, where each one aggregates arrivals and departures, refered as *ingress* and *egress* in [4]. One implementation is a Distributed Cache Line Counter [18] whose `increment()/decrement()` operations can be used as `arrive()/depart()`, scales well with the number of threads, and is wait-free. Another alternative for a `readIndicator` is SNZI [9], but this one is lock-free for the `arrive()/depart()` operations, which means that using it in the Left-Right algorithm would make the Readers have a lock-free progress guarantee instead of wait-free.

Although we didn't focus in NUMA architectures, a specific `readIndicator` can be designed to achieve a minimum of contention and false sharing [22], along with memory allocation. This solution would need four distributed counters with an attributed position per core, corresponding to an *ingress* and *egress* per `versionIndex`. The simplest implementation for the `readIndicator` is to have two counters, updated atomically, one per `versionIndex`, but such a solution would entail high contention for the Reader threads, incurring performance penalties.

In the presented pseudo-code for the `readIndicator` shown in Algorithms [2,3,5], we chose to use a two-dimensional array with one entry for each `versionIndex`, named `readersVersion[][]`. Each Reader thread has two attributed entries, `readersVersion[0]`[$threadIndex$] and `readersVersion[1]`[$threadIndex$], where `threadIndex` is stored in a thread-local variable. Each entry of `readersVersion[][]` is placed in its own cache line, using padding and alignment, to avoid false sharing between Readers. When compared with the single counter approach, having a larger array to scan may harm Writer's performance but will improve Reader's performance.

2.1 Algorithm Description

2.1.1 Read operations (contains())

1. Load the current value of `versionIndex` (can be 0 or 1), store it as X and atomically set Reader's entry to state READING on `readersVersion[`X`][]`.
2. Use the current value of `leftRight` to decide which TreeSet should the read operation run in: If `leftRight` is `LEFT` then call `leftTree.contains()`, and if it is `RIGHT` then call `rightTree.contains()`.
3. Atomically set the Reader's entry to state `NOT_READING` on `readersVersion[`X`][]`.
4. Return the value obtained from the `contains()` function.

2.1.2 Write operations (add()/remove())

Both the `add()` and `remove()` operations have the same algorithm, for simplicity we will refer to these methods as `modify()`.

Algorithm 1: Algorithm for read operations - contains()

```
  Input: Key
  Output: Value
1 localVersionIndex = versionIndex.get();
2 readIndicator.arrive(localVersionIndex);
3 if leftRight.get() == LEFT then
4     Value = leftTree.contains(Key);
5 else
6     Value = rightTree.contains(Key);
7 end
8 readIndicator.depart(localVersionIndex);
9 return Value;
```

Algorithm 2: An implementation of readIndicator.arrive()

```
  Input: X
1 tlsEntry = ThreadLocal.get();
2 readersVersion[X][tlsEntry.threadIndex].set(READING);
```

Algorithm 3: An implementation of readIndicator.depart()

```
  Input: X
1 tlsEntry = ThreadLocal.get();
2 readersVersion[X][tlsEntry.threadIndex].set(NOT_READING);
```

1. Acquire the `writersMutex` to prevent any other thread from executing a `modify()`.
2. Decide on which tree to do the `modify()`, based on the current value of `leftRight`: If `leftRight` is `LEFT` then call `rightTree.modify()`, and if it is `RIGHT` then call `leftTree.modify()`.
3. Toggle `leftRight` using an atomic assignment. This will cause new read operations to run on the recently modified tree. Notice that for simplicity, we chose to encode `LEFT` as having the negative value of `RIGHT`.
4. Compute the new `versionIndex` and scan `readersVersion[`X`][]` yielding execution when an entry is in state `READING`, where X is the newly computed `versionIndex`.
5. Set `versionIndex` to X using an atomic assignment.
6. Scan `readersVersion[`$\bar{X}$`][]` yielding execution when an entry is in state `READING`, where $\bar{X}$ is the previous `versionIndex`.
7. If `leftRight` is in `LEFT` then call `rightTree.modify()`, and if it is in `RIGHT` then call `leftTree.modify()`.
8. Unlock the `writersMutex`.

2.1.3 Progress Conditions

A method is Wait-Free Population Oblivious if it guarantees that every call completes its execution in a finite number of steps, which does not depend on the number of active threads.

The read operation has no loops and does an Atomic.get() on line 1, an Atomic.set() on line 2, an Atomic.get() on line 3, and finally an Atomic.set() on line 8, as shown in Algorithm 1. This gives a total of four Atomic operations, which means that the number of synchronized instructions is finite and constant, regardless of the number of threads in the system. This ensures that read operations are **Wait-Free Population Oblivious**.
For the write operation, there is a `while()` loop of `readIndicator`'s `isEmpty()` which makes the Writer conditionally wait on previous Reader's progress, because new Readers will have no impact on Writer progress. This means that if eventually the read operations make progress, the Writer will make progress too. The Writer has to acquire a mutually exclusive lock `writersMutex`, which means its progress condition is **Blocking**.

Algorithm 4: Algorithm for write operations - modify()

```
   Input: Key
1  writersMutex.lock();
2  localLeftRight = leftRight.get();
3  if localLeftRight == LEFT then
4      rightTree.modify(Key);
5  else
6      leftTree.modify(Key);
7  end
8  leftRight.set(-localLeftRight);
9  prevVersionIndex = versionIndex.get();
10 nextVersionIndex = (prevVersionIndex + 1)%2;
11 while not readIndicator.isEmpty(nextVersionIndex) do
12     yield();
13 end
14 versionIndex.set(nextVersionIndex);
15 while not readIndicator.isEmpty(prevVersionIndex) do
16     yield();
17 end
18 if -localLeftRight == LEFT then
19     rightTree.modify(Key);
20 else
21     leftTree.modify(Key);
22 end
23 writersMutex.unlock();
```

Algorithm 5: An implementation of readIndicator.isEmpty()

```
  Input: X
1 for i in readersVersion[X].size do
2     if readersVersion[X][i].get() == READING then
3         return false;
4     end
5 end
6 return true;
```

2.2 Synchronization

We will now describe some of the synchronization details, keeping in mind that this algorithm assumes the usage of a memory model with Sequential Consistency similar to the one in the JVM or the default on C11/C++1x. This means that `Atomic.get()` functions are atomic and executed with an acquire-barrier, and `Atomic.set()` are atomic and imply a release-barrier [7].

2.2.1 Versioning Mechanism

A very important detail of the synchronization is in the order of the access to the variables `versionIndex` and `leftRight`. This sequence is reversed in the write and read operations, thus creating an hand-shaking procedure between the Writer and Readers. As can be seen in Algorithm 1, the Reader will load the value of `versionIndex` in line 1, and then load the value of `leftRight` in line 3, whilst in Algorithm 4, the Writer will first set `leftRight` to the new value in line 8, and then set `versionIndex` to the new version in line 14.

By using a reversed sequence, the algorithm guarantees for the Writer, that any Reader that gets the new `versionIndex` will also get the new `leftRight`. Furthermore, if the Reader did not publish the `versionIndex`, it gives a guarantee to the Writer: that the Reader has not yet read the `leftRight`, which means that when it does, it will get the latest up-to-date value of the `leftRight` variable.

2.2.2 Correctness

The sequential logic of the Left-Right technique can be represented with state machine diagrams, where the transitions between states are atomic and *instantaneous*, and the time spent on each state is undetermined. Figure 2 represents the full state machine of the Writer and the Readers.

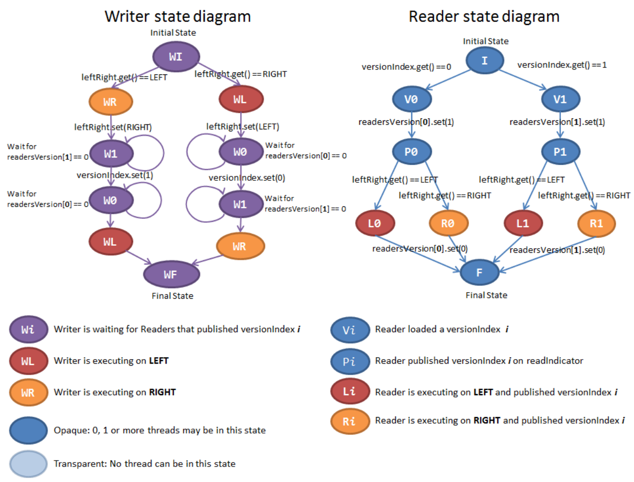

Figure 2. Full state machines for the Writer and Readers. Some of the transitions on the Reader's diagram are not allowed when the Writer is in certain states.

Figures 3,4 and 5 represent all the steps a Writer takes to update both `leftInstance` and `rightInstance` while making sure there is no Reader thread executing on the instance being modified, which we will from now on refer to as the Writer running in *exclusivity*. Figures 4 and 5 correspond respectively to all even and odd number of the write operations, except for the first write operation which is presented in figure 3.

For the first write operation, there can not exist a Reader that has loaded `versionIndex` as 1, because at the start, the Writer could not have toggled `versionIndex`, and its starting value is 0. This is the reason why all the states at the Reader's state machine that depend on the `versionIndex` to be 1 are transparent, which means there are no Readers on those states.

We will proceed by explaning figures 4 and 5, keeping in mind that the first write operation is a special case of 5. As can be observed in both figures, the Writer controls the flow of the Readers by toggling the `leftRight` and `versionIndex` variables. The Writer starts by modifying the instance opposite to where the Readers are currently running. Without any validation, the Writer is sure there is no Reader on that instance, because the previous write operation has already ensured it in order to be able to finish. It is interesting to notice that every time a write operation modifies the second instance, it will guarantee that it performs in exclusivity, and will also automatically guarantee that the next write operation will also perform in exclusivity when modifying the same instance, which will be its first instance.

Another important result is that both end states 4.E and 5.E, are the starting machine states of each other, where 4.E corresponds to 5.A, and 5.E to 4.A, this is again because the previous write operation is leaving the Reader's state machine in a configuration that is expected by the next write operation.

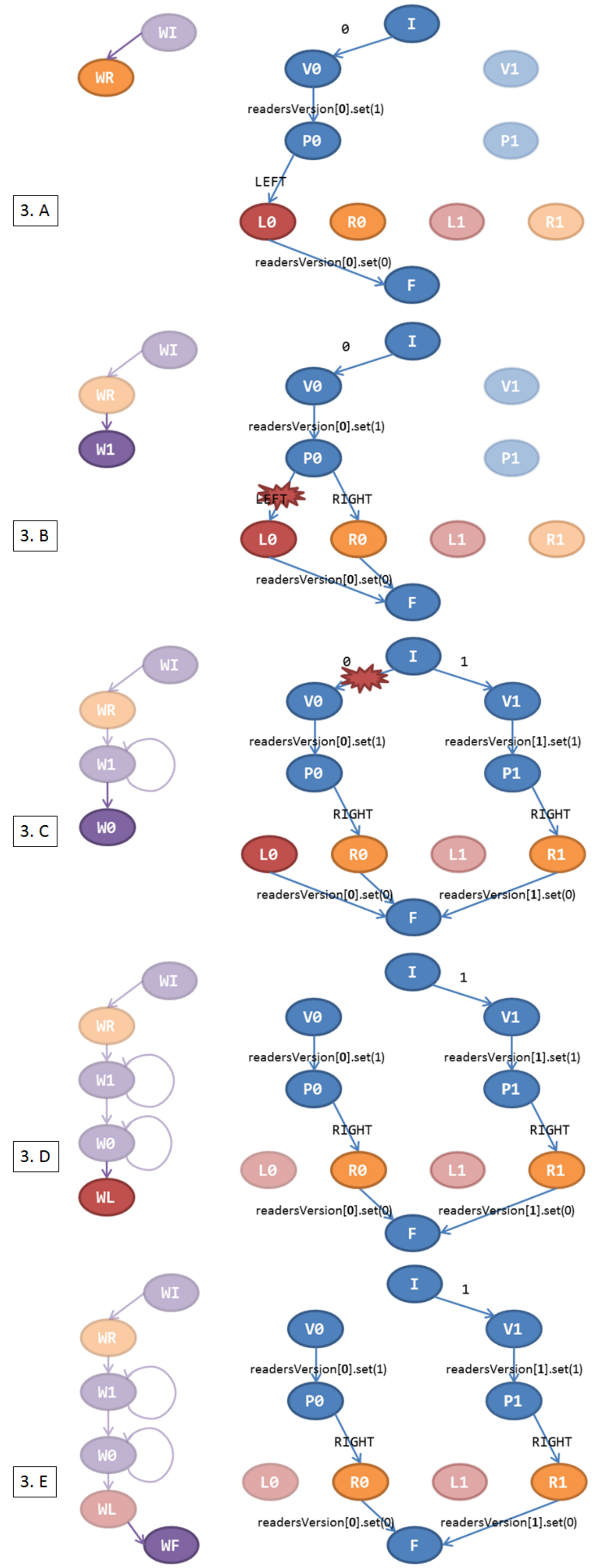

Figure 3. For the first Writer, the `leftRight` is `LEFT` and `versionIndex` is 0.

After modifying `leftInstance` or `rightInstance`, shown in figures 4.A and 5.A respectively, the next step is to toggle the variable `leftRight` as can be seen in figures 4.B and 5.B. From this moment on, there can be Readers executing on the `leftInstance` or on the `rightInstance`.

The next task for the Writer is to guarantee that all Reader threads will be running on the instance indicated by the recently modified `leftRight`. Considering figure 4, the Writer will start by validating that there is no Reader that published `versionIndex 0`, this includes all Readers that are at states **P0**, **L0** and **R0**. Notice that the Writer can not distinguish between states **P0**, **L0** or **R0**, it only knows that is *possible* for multiple Reader threads in any of those states to be executing on the `rightInstance`, the instance that is to be modified. In case there is still one Reader running that published `versionIndex 0`, the Writer will wait, represented by state **W0**. Starting by validating `versionIndex 0` guarantees that the Writer will not starve, because new read operations will load `versionIndex 1`, which means that read operations that can publish `versionIndex 0` are finite. This can be easily seen on figure 4.B where the only possible transition is from state **I** to state **V1**.

After validating that all the Reader threads that published `versionIndex 0` are finished, the Writer can proceed to toggle `versionIndex` from `1` to `0`. Figure 4.C shows that all new read operations will load `versionIndex 0` and `leftRight LEFT`. But there are still *old* read operations that have loaded `versionIndex 1` and may be running on the `rightInstance`, on state **R1**.

Now that the `versionIndex` has been toggled to `0`, all new read operations will load `versionIndex 0` which means there is a finite number of read operations that can publish on the `readIndicator` `readersVersion[1]` this guarantees that the write operation is not starved. Again, the proceedure will be the same, the Writer will wait for all Readers that published `versionIndex 1` to be finished, states **P1**, **L1** and **R1**. Once the `readIndicator` is empty for `versionIndex 1`, it guarantees that **R1** is empty. Because there can be Readers that are on state **V1**, read operations that loaded `versionIndex 1` but still did not publish on the `readIndicator`, are not seen by the Writer and can transition to **P1** and **L1**, which explains the state machine on figure 4.D. Finally the Writer can proceed to modify the second instance because both states **R0** and **R1** are empty, so all read operations are running on the `leftInstance`, as shown in figure 4.E.

This sequence of figures demonstrate the validity of the synchronization between Writer and Readers, when a Writer is toggling Readers from `RIGHT` to `LEFT`. A similar demonstration can be done on the opposite direction, which is shown on figure 5.

On the electronic version of this document, Figure 13 displays an animation of the state machines of the Readers as a Writer progresses.

2.3 Linearizability

As far as the Writers are concerned, during the `modify()` operation described in Algorithm 4, the `writersMutex` ensures there is a single Writer at a time and, therefore, any atomic step in the `modify()` can be chosen as the linearization point.

For the Readers, it must be the point after which the logical change by the Writer becomes visible. Any Reader reading `leftRight` in line 3 of Algorithm 1 will see the changes done by the Writer if it reads the `leftRight` after the Writer has updated it. Otherwise, it sees the data structure in the previous state.

2.4 Example scenario

We will now show an example scenario with multiple threads (one Writer and five Readers), with temporal progression going from top to bottom, starting with the main variables in the states:

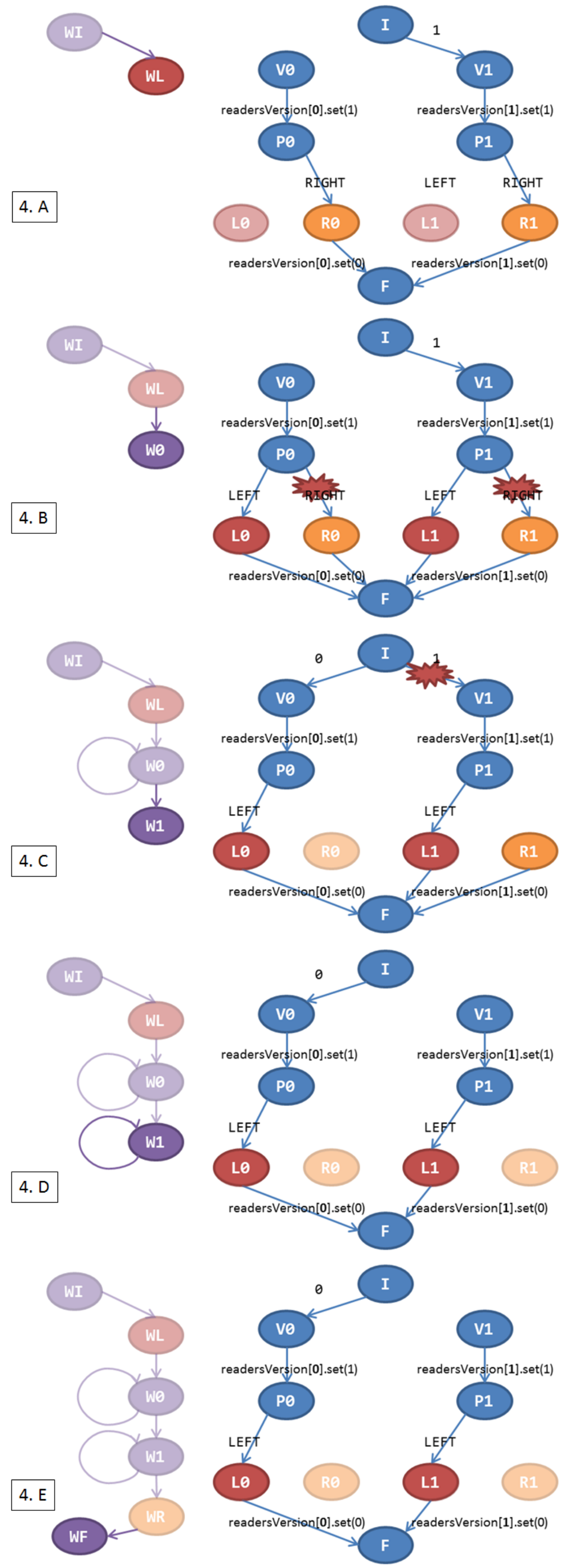

Figure 4. State machine of Writer and Readers when the Writer starts on the `rightInstance`.

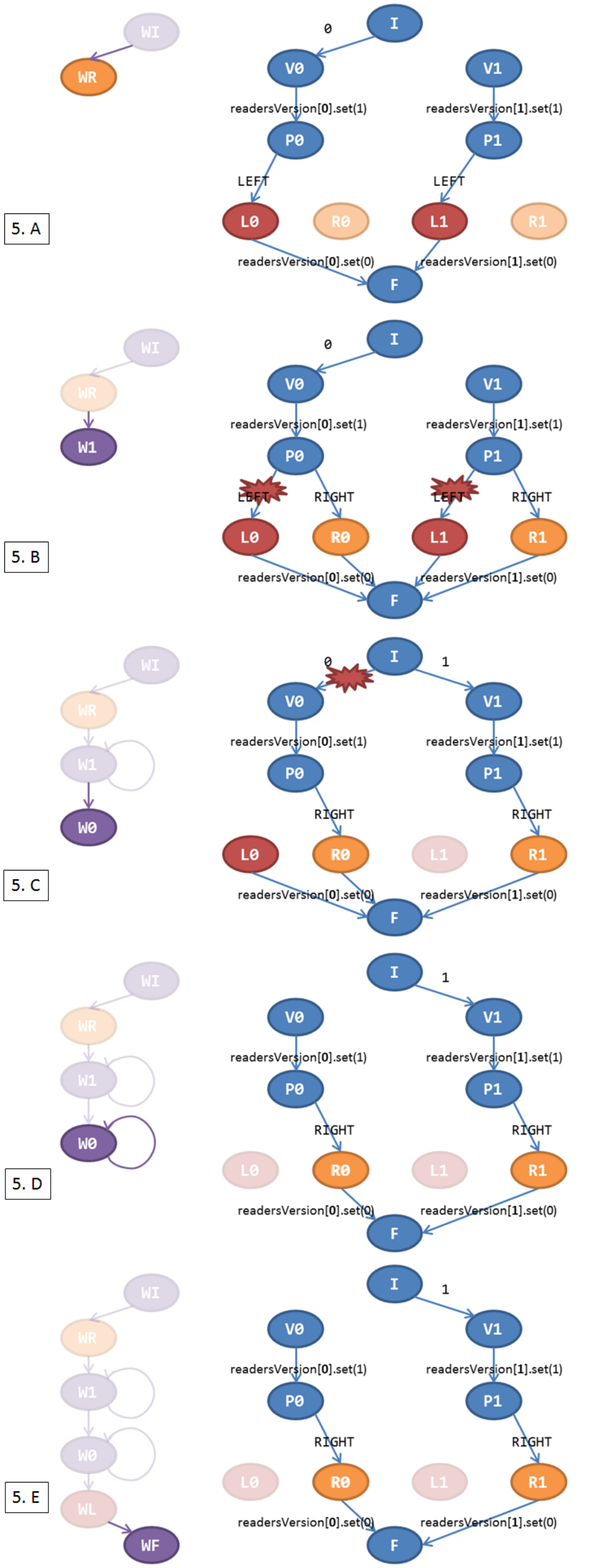

Figure 5. State machine of Writer and Readers when the Writer starts on the `leftInstance`.

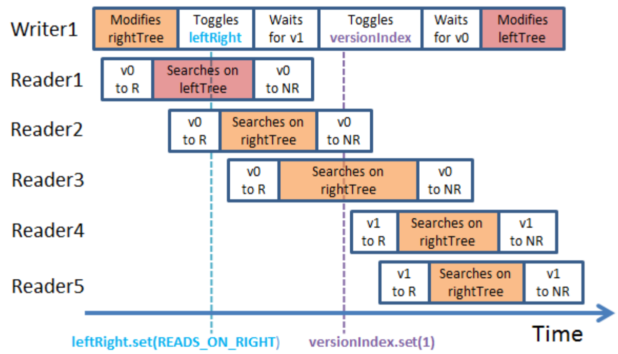

Figure 6. A time diagram of the example described with one Writer and five Readers.

`leftRight`=`LEFT`, `versionIndex`=0, `readersVersion[0]`= `readersVersion[1]` = $\{0, 0, 0, 0, 0\}$,
Writer1 $\rightarrow$ Modifies the `rightTree`
Reader1 $\rightarrow$ Sees `versionIndex` 0 and publishes `readersVersion[0]`= $\{1, 0, 0, 0, 0\}$, does Read on `leftTree`
Writer1 $\rightarrow$ Toggles `leftRight` to `RIGHT`
Reader2 $\rightarrow$ Sees `versionIndex` 0 and publishes `readersVersion[0]`= $\{1, 1, 0, 0, 0\}$, does Read on `rightTree`
Writer1 $\rightarrow$ Yields until `readersVersion[1]` is $\{0, 0, 0, 0, 0\}$
Reader3 $\rightarrow$ Sees `versionIndex` 0 and publishes `readersVersion[0]`= $\{1, 1, 1, 0, 0\}$, does Read on `rightTree`
Writer1 $\rightarrow$ Toggles `versionIndex` to 1
Reader4 $\rightarrow$ Sees `versionIndex` 1, `readersVersion[0]`= $\{1, 1, 1, 0, 0\}$, `readersVersion[1]`= $\{0, 0, 0, 1, 0\}$, does Read on `rightTree`
Write1 $\rightarrow$ Yields until `readersVersion[0]` is $\{0, 0, 0, 0, 0\}$. It will wait until Reader1, Reader2, and Reader3 finish, because it could be possible for Reader3 to be on the `leftTree`. Remember that Write1 doesn't know if Reader3 executed before or after `leftRight` was toggled. In the end we will have `readersVersion[1]`=$\{0, 0, 0, 1, 0\}$
Read5 $\rightarrow$ Sees `versionIndex` 1 and publishes `readersVersion[1]`=$\{0, 0, 0, 1, 1\}$ does Read on `rightTree`
Write1 $\rightarrow$ Modifies the contents of the `leftTree`

A graphical depiction of a time diagram with an example scenario can be seen in Figure 6.

3. Algorithm Variants

With the intent of minimizing the number of synchronization primitives on the read operations, we have developed multiple variants of the original algorithm. All these variants continue to follow the same principle, that Readers have to publish their state, and the Writer is responsible for executing in the instance opposite to the one where the Readers are running. For the *No Version* and *Reader's Version* variants shown below, the `readIndicator` implementation can no longer use counters, instead, each Reader has a dedicated entry where it publishes its state, using the `setState(state)` operation, implying a memory allocation of $O(N_{Readers})$.

3.1 NV - No Version

In this variant of the algorithm, there is no `versionIndex`. Each Reader's state can have four different values: `NOT_READING`, `READING`, `LEFT`, `RIGHT`, where valid transitions are shown in Figure 7.

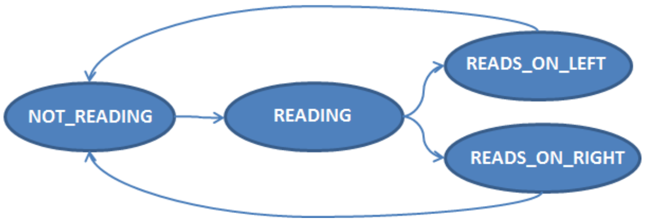

Figure 7. Reader's state machine for the NV algorithm. The transitions between states are atomic.

Algorithm 6: NV Algorithm for read operations

```
Input: Key
Output: Value
readIndicatorNV.setState(READING);
if leftRight.get() == LEFT then
    readIndicatorNV.setState(LEFT);
    Value = leftTree.contains(Key);
else
    readIndicatorNV.setState(RIGHT);
    Value = rightTree.contains(Key);
end
readIndicatorNV.setState(NOT_READING);
return Value;
```

Algorithm 7: NV Algorithm for write operations

```
Input: Key
writersMutex.lock();
localLeftRight = leftRight.get();
if localLeftRight == LEFT then
    rightTree.modify(Key);
else
    leftTree.modify(Key);
end
leftRight.set(-localLeftRight);
readIndicatorNV.waitIfReadingOrArgument(localLeftRight);
if -localLeftRight == LEFT then
    rightTree.modify(Key);
else
    leftTree.modify(Key);
end
writersMutex.unlock();
```

The main difference from the clasical method is that the Writer will wait if there are any Readers in state `READING` or in the state corresponding to the previous value of the `leftRight` variable, either `LEFT` or `RIGHT`. We represent this functionality on Algorithm 7 as `waitIfReadingOrArgument()`.

One possible limitation of this algorithm is that, from a theoretical point of view, it could be possible for a Writer to be stuck indefinitely waiting for a Reader that finishes its operation and starts a new operation immediately afterwards, going temporarily into the `READING` state, but this issue is unlikely to occur for more than a few iterations.

3.2 RV - Reader's Version

In this variant of the algorithm, each Reader has its own version which it increments and publishes.

As shown in Algorithm 8, the sign of the Reader's version is used to represent whether the Reader is in a reading state or not. The version is set before starting the operation, changing the sign from negative to positive and incrementing the version by 1. In the end of the operation, it will change back the sign from positive to negative. For example, when starting with a Reader's version of -1, the next version will be 2, followed by the `contains()` operation, and finally a change to version -2.

On Algorithm 9 we show that the Writer has to *scan* the version of each Reader (`waitUntilIncrementOrNegative()`), waiting for an increment of the version, or for the version to be set to a negative value, before it can proceed with its own operation.

One theoretical limitation of this approach is that, the variable for the state of each Reader is continuously incremented and could eventually overflow. If a 64 bit integer is used for this variable, it should take many thousands of years for a current modern CPU to be able to overflow it, thus avoiding this issue in practice.

Algorithm 8: RV Algorithm for read operations

```
Input: Key
Output: Value
tlsEntry = ThreadLocal.get();
readIndicatorRV.setState(1-tlsEntry.localVersion);
if leftRight.get() == LEFT then
    Value = leftTree.contains(Key);
else
    Value = rightTree.contains(Key);
end
readIndicatorRV.setState(tlsEntry.localVersion-1);
return Value;
```

Algorithm 9: RV Algorithm for write operations

```
Input: Key
writersMutex.lock();
localLeftRight = leftRight.get();
if localLeftRight == LEFT then
    rightTree.modify(Key);
else
    leftTree.modify(Key);
end
leftRight.set(-localLeftRight);
readIndicatorRV.waitUntilIncrementOrNegative();
if -localLeftRight == LEFT then
    rightTree.modify(Key);
else
    leftTree.modify(Key);
end
writersMutex.unlock();
```

3.3 Optimistic Read

We will now show an algorithm that has an optimistic approach to the Left-Right pattern. The idea is that the Reader *assumes* that no Writer is changing the tree where the Reader is executing, which will be the case if the `versionIndex` hasn't changed.

Notice that to implement the Optimistic variant of the algorithm, it requires a small modification of the algorithm for the Writer. As shown in bold on algorithm 11, instead of being a two-state variable (0 or 1), the `versionIndex` is now an incremental counter. We also add a release-barrier to prevent code in the read operation from being re-ordered [2] and placed after reading `versionIndex` in line 8.

Mechanisms such as this one are not new [12], and have been used before for Reader-Writer locks [16] and directly on data structures [3]. Similarly to these mechanisms, this variant is not as

Algorithm 10: Optimistic algorithm for read operations

```
Input: Key
Output: Value
localVersionIndex = versionIndex.get();
if leftRight.get() == LEFT then
    Value = leftTree.contains(Key);
else
    Value = rightTree.contains(Key);
end
releaseBarrier();
newLocalVersionIndex = versionIndex.get();
if newLocalVersionIndex == localVersionIndex then
    return Value;
else
    containsKeyAlgorithm1();
end
```

Algorithm 11: Algorithm for write operations when using optimistic read operations

```
Input: Key
writersMutex.lock();
localLeftRight = leftRight.get();
if localLeftRight == LEFT then
    rightTree.modify(Key);
else
    leftTree.modify(Key);
end
leftRight.set(-localLeftRight);
prevVersionIndex = versionIndex.get();
nextVersionIndex = prevVersionIndex + 1;
readIndicator.waitUntilEmpty(nextVersionIndex % 2);
versionIndex.set(nextVersionIndex);
readIndicator.waitUntilEmpty(prevVersionIndex % 2);
if -localLeftRight == LEFT then
    rightTree.modify(Key);
else
    leftTree.modify(Key);
end
writersMutex.unlock();
```

generic as the previously described variants of the Left-Right technique, because it allows a Reader and a Writer to run on the same instance at the same time. In addition, this kind of approach requires automatic GC and the underlying object or data structure must have atomicity guarantees on their members. For example, on C11/C++1x, using an optimistic approach for a tree requires the underlying tree to have a node traversal functionality with atomic proprieties, which can be achieved with relaxed atomics [7]. On Java, there is no need for the nodes of the tree to be `volatile` because the JVM guarantees atomicity for references [17], thus allowing this technique to be used in Java, with several single-threaded data structure without any modifications.

3.4 Read synchronized operations

Depending on the variant of the algorithm, we get a different finite number of atomic sequentially consistent operations when doing a Read:

- Classic Left-Right: 2 get() + 2 set()
- NV - No Version: 1 get() + 3 set()
- RV - Reader's Version: 1 get() + 2 set()
- Optimistic: minimum 3 get(), maximum 5 get() + 2 set()

Apart from the Optimistic method, the three variants have similarly little contention and provide equal performance as can be seen on section 4.

Recently discovered Reader-Writer locks [4] have shown that it is possible to have good scalability proprieties for read operations with a number of synchronized calls of one atomic `get()` and two atomic `set()` at best, a performance that the RV variant of the Left-Right technique will always guarantee.

3.5 Left-Right technique as a Reader-Writer Lock

Although the Left-Right technique is not a Reader-Writer lock, it can be implemented and used in a way very similar to one, the main difference being that a Reader-Writer lock protects a block of code, while the Left-Right protects a specific object without shared attributes.

Algorithm 12: Algorithm for readerLock()

```
Input: leftInstance,rightInstance
Output: instance
tlsEntry = ThreadLocal.get();
tlsEntry.localVersionIndex = versionIndex.get();
readIndicator.setState(tlsEntry.localVersionIndex, READING);
if leftRight.get() == LEFT then
    return leftInstance;
else
    return rightInstance;
end
```

Algorithm 13: Algorithm for readerUnlock()

```
tlsEntry = ThreadLocal.get();
readIndicator.setState(tlsEntry.localVersionIndex, NOT_READING);
```

Algorithms 12 and 13 describe how to transform Algorithm 1 to obtain a funcionality similar to a `readLock()` and `readUnlock()`. A thread-local-storage variable is used to pass the value read for the `versionIndex` between the `readLock()` and the `readUnlock()`, but other techniques are possible. Notice that although the names have the word "lock" associated, these two algorithms are non-blocking.

For the write operations, a third method must be implemented, and it is the one responsible for *swapping* the two instances. Algorithms 14, 15 and 16 describe how to transform algorithm 4 to provide a functionality similar to a `writeLock()` and `writeUnlock()`.

Compared to Reader-Writer locks, the main limitation of the Left-Right technique using this approach, is that it requires two identical instances (`leftInstance` and `rightInstance`) of the single object or data structure that is meant to be accessed in a thread-safe way.

Algorithm 14: Algorithm for writerLock()

```
Input: leftInstance,rightInstance
Output: instance
writersMutex.lock();
if leftRight.get() == LEFT then
    return rightInstance;
else
    return leftInstance;
end
```

Algorithm 15: Algorithm for writerToggle()

```
Input: leftInstance,rightInstance
Output: instance
localLeftRight = leftRight.get();
leftRight.set(-localLeftRight);
prevVersionIndex = versionIndex.get();
nextVersionIndex = (prevVersionIndex + 1)%2;
readIndicator.waitUntilEmpty(nextVersionIndex);
versionIndex.set(nextVersionIndex);
readIndicator.waitUntilEmpty(prevVersionIndex);
if -localLeftRight == LEFT then
    return rightInstance;
else
    return leftInstance;
end
```

Algorithm 16: Algorithm for writerUnlock()

```
writersMutex.unlock();
```

Algorithm 17: Example of using a Left-Right pattern instead of a Reader-Writer lock to protect an object

```
Input: leftInstance,rightInstance
instance = readerLock(leftInstance, rightInstance);
instance.someReadOnlyOperation();
readerUnlock();
...
firstInstance = writerLock(leftInstance, rightInstance);
firstInstance.someWriteModifyOperation();
secondInstance = writerToggle(leftInstance, rightInstance);
secondInstance.someWriteModifyOperation();
writerUnlock();
```

4. Performance Evaluation

A set of performance tests were conducted on a dual Opteron 6272 with a total of 32 cores, running Windows 7 with JDK 8 (b100). We executed 7 individual runs for each of the data structures presented below, and plotted the median of the operations per milisecond, where the value of operations per milisecond is an average over a period of 30 seconds, which is presented on Figures 8, 9, 10 and 11.

Each of the seven runs was done twice, once with a TreeSet that contained one thousand elements and once with one million elements, totalling 14 runs. On each run, there were always 2 threads doing solely write operations, where each thread did one `remove()` folowed by one `add()` operation. This was done in a sequential way over an array with 4 times the number of elements in the set, such that the `remove()` is done on the ith-element and the `add()` for the ith-element plus `numElements`, where `numElements` may be 10^3 or 10^6. This way we ensure that the tree is constantly mutating, and rebalaced often.

- **RWLockTreeSet**: java.util.TreeSet protected with a Reader-Writer lock ScalableStampedRWLock [20]. The Scalable-StampedRWLock is a freely available lock that combines the C-RW-WP lock described in [4] with the StampedLock provided in Java JDK8 [16]. The `add()` and `remove()` are protected with `exclusiveLock()`, and the `contains()` with the `sharedLock()` and, therefore, all operations are blocking.
- **LRScalableTreeSet**: java.util.TreeSet with the classic Left-Right technique described in Algorithms 1 and 4.

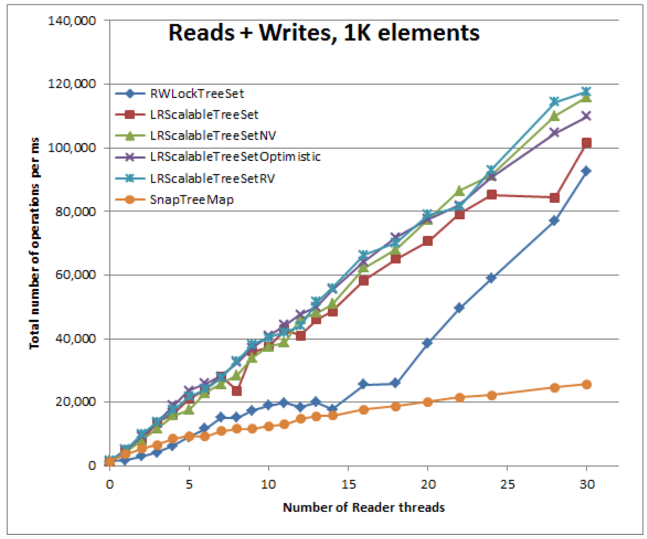

Figure 8. Total operations per millisecond as function of the number of Readers when 2 Writers are running on a set of 10^3 elements

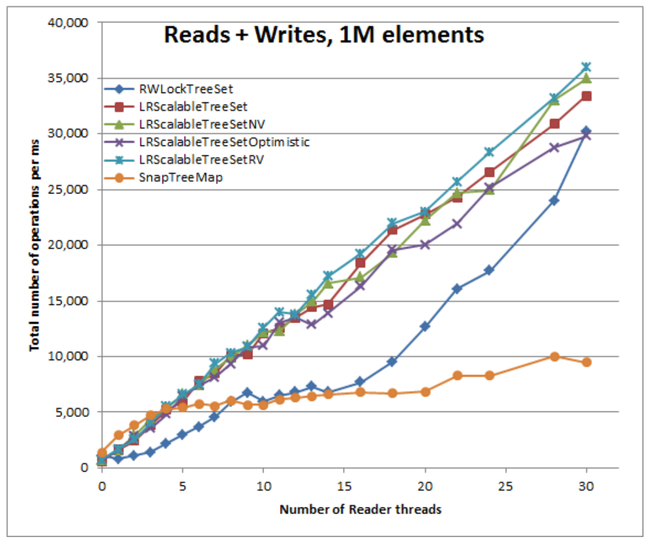

Figure 9. Total operations per millisecond as function of the number of Readers when 2 Writers are running on a set of 10^6 elements

- **LRScalableTreeSetNV**: java.util.TreeSet with the Left-Right technique (No Version) without using a `versionIndex`, as described in Algorithms 6 and 7.
- **LRScalableTreeSetRV**: java.util.TreeSet with the Left-Right technique (Reader's Version) where each Reader updates its own version that replaces the state, as described in Algorithms 8 and 9.
- **LRScalableTreeSetOptimistic**: java.util.TreeSet with the optimistic approach described in Algorithms 10 and 11.
- **SnapTreeMap**: edu.stanford.ppl.concurrent.SnapTreeMap with hand-over-hand optimistic validation and a relaxed balance tree. All operations of the SnapTreeMap are blocking.

Initially, we tried to compare with an implementation using the COW pattern, based on an immutable TreeMap [10], but tests with

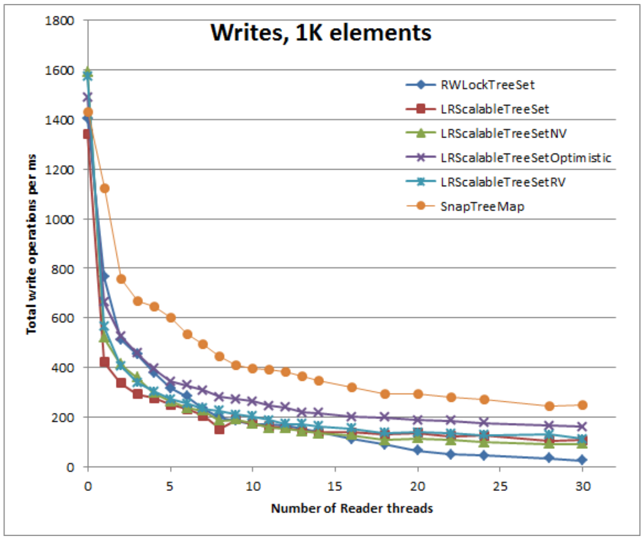

Figure 10. Write operations per millisecond as function of the number of Readers on a set of 10^3 elements

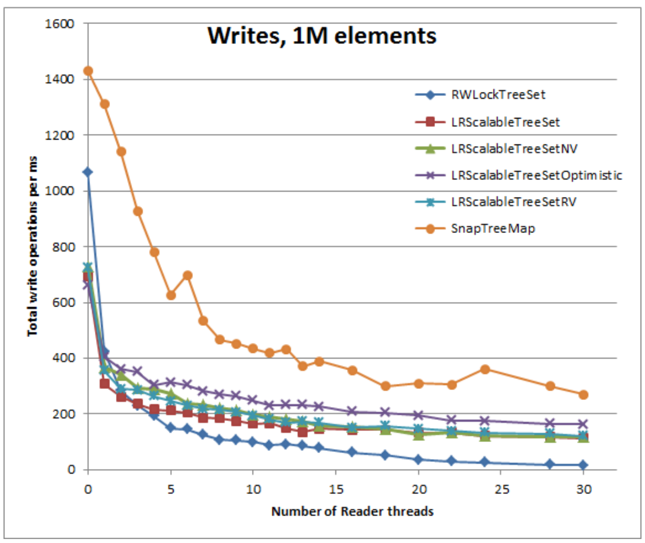

Figure 11. Write operations per millisecond as function of the number of Readers on a set of 10^6 elements

10^3 elements gave low performance, and tests with 10^6 elements were so slow to fill the initial tree as to make the technique impractical, so we chose not to include this technique in our benchmarks.

As expected, as the number of Reader threads increases, all four variants of the Left-Right technique scale almost linearly. Regarding the total number of operations, they have a throughput of up to five times higher when compared with the SnapTreeMap, if we consider a TreeSet with 1000 elements. The throughput of the SnapTreeMap increases slowly with the number of Readers and it seems to have reached a plateau on Figures 8 and 9. The RWLockTreeSet also scales well as the number of Reader threads increases, but the number of write operations decreases signifcantly as seen on Figures 10 and 11.

Regarding write operations, the algorithm with the highest performance is the SnapTreeMap, which can be explained by: the SnapTreeMap uses a relaxed balance tree and multiple Writers can execute at the same time; the Left-Right pattern has to write on two distinct trees and serializes writes.

Notice that the SnapTreeMap algorithm does $O(ln\ n)$ atomic sequentially consistent loads on each read operation, and our benchmark was done on a machine with x86 architecture, that does not incur a performance hit when executing these atomic operations, which gives the SnapTreeMap an advantage. The same benchmark on other architectures may yield better results for the Left-Right technique because while traversing the TreeSet it does not execute any atomic sequentially consistent loads, both for the read and write operations.

4.1 Workload Tests

As mentioned before, the Left-Right technique benefits from dedicated Reader threads, but for the sake of comparison, we also compared the RWLockTreeSet, LRTreeSetOptimistic, and SnapTreeMap, using threads that perform both read and write operations. We experimented with three different workload configurations, 10%, 1% and 0.1% writes, where each percentage value represents the probability that a write operation will be done, using a random number generator to determine whether it is a read or write operation. For example, the plot on Figure 7 with 10% Writes means that, on average, for every write operation there were nine read operations. Similarly to the performance tests on the previous section, each write operation consists of a `remove()` done on the ith-element and an `add()` on the ith-element plus `numElements`.

On this setting, the SnapTreeMap is the overall winner, benefiting from the fact that Writes can execute simultaneously, while the Left-Right techniques and the RWLockTreeSet serialize writes. The only scenario where the LRTreeSetOptimistc performs better or equal to the SnapTreeMap is for 0.1% writes.

These kind of mixed task tests, where reads are dependent on a previous write finishing, cause an artificial serialization that prevents reads from being scalable and from taking advantage of the Wait-Free progress condition provided by the Left-Right technique.

4.2 Latency measurements

In order to estimate latency, we used the scenario described in section 4, with two dedicated Writer threads and two dedicated Reader threads, and measured the time it took for the `contains()` method to complete using `System.nanotime()`. The test ran for 10^4 seconds for each data structure, executing more than 10^{10} function calls per data structure. We chose to compare the latencies of the `contains()` operation for the RWLockTreeSet, LRScalableTreeSet and SnapTreeMap, and the results are show in Table 2.

	RWLockTreeSet	LRScalableTreeSet	SnapTreeMap
99%	35	< 1	3
99.9%	43	< 1	6
99.99%	111	2	16

Table 2. Latency measurements in microseconds for the `contains()` method on a tree with 10^3 elements.

Using the RWLockTreeSet as an example, the table can be read as follows: 99% of the calls to the contains() method take 35 microseconds or less to complete. Table 2 shows a good latency performance for `contains()` operations on the LRScalableTreeSet, where according to our measurements, 99.99% of the operations take 2 microseconds or less to complete.

5. Advantages/Disadvantages

The Left-Right technique has some disadvantages when compared with other concurrency techniques:

- Consumes twice the memory for the data structure itself, but not the data it contains.
- Has a higher synchronization cost than the Copy-On-Write with CompareAndSet technique.
- Write operations are not concurrent, and must be performed twice, once on each tree.

Some of the advantages of using this pattern are:

- This algorithm can be implemented on top of any single-threaded data structure (not just trees), or even a single object.
- The SnapTreeMap and COW data structures all require a GC, but at the exception of the Optimistic, none of the variants of the Left-Right technique need a GC or an extra memory management system.
- It is Wait-Free Population Oblivious on read operations.
- Unlike other techniques such as RCU [15], that require native API support, the Left-Right can be used in any system that has support for languages with a sequentially consistent memory model (i.e. C++1x, C11, Java, Scala).
- Write operations (`add()/remove()`) have to wait only for the read operations that started *before* the `versionIndex` was modified. Moreover, it allows the existence of a dedicated Writer thread without any impact on the Readers.

6. Discussion

Depending on specific application requirements, there are still improvements that can be done on this algorithm. The following paragraphs describe some of them.

Single Writer Some multi-threaded applications have, by design, a single dedicated Writer thread and multiple Reader threads. For those kind of applications, the `writersMutex lock()/unlock()` calls can be removed, which will result in a performance improvement.

Asynchronous Writes In case asynchronous writes are an acceptable approach, a reserved thread can be dedicated to the write operations and a lock-free queue used to delegate `add()/remove()` operations from the other threads to this one. This has the disadvantage that a before-happens sequence between read and write operations is lost, but if the application is insensitive to it, then this technique will behave as a single Writer.

Bulk Writes It is simple to provide extra functionality to perform several writes in one shot, through `addBulk()/removeBulk()` functions. This improves the performance because it reduces the overall number of synchronized operations per write.

7. Conclusion

We have shown in this article a generic concurrency technique that provides Wait-Free Population Oblivious guarantees for the read operations and does not require automatic Garbage Collection. Its two main innovations consist of, the usage of two instances of the underlying object or data structure, which allow a Writer and multiple Readers to work simultaneously, and the development of a new concurrency control algorithm that gives the read operations a Wait-Free progress guarantee. A practical implementation for a concurrent tree using the Left-Right technique was presented, that when compared with other concurrent implementations can underperform when it comes to write operations, but when using dedicated Reader threads, it provides a scalability for read operations, that the others can not match.

Due to the recent trend of increased multi-core systems, concurrency researchers are, more than ever, being pressured to find practical mechanisms that allow systems to scale. Until now, most of the focus was on enabling concurrency through *serialization*. The Left-Right technique is a mechanism that by using two instances, reduces the contention on a resource, thus increasing *parallelization*. We believe that due to its performance, latency, and flexibility of usage, in practice, this pattern can be used to wrap any *single* data structure or object, thus avoiding the employment of other synchronization techniques, such as Reader-Writer locks. Moreover, when compared with Reader-Writer locks, the Left-Right pattern has the advantage that it is non-blocking for the read operations, thus providing strong latency guarantees that no Reader-Writer lock is able to provide.

Acknowledgments

We wish to thank *anonymous reviewer 1* on the Scala2013 conference for his encouragement and important contribution to the linearization. And a thanks to Davide Cuda for his helpful comments on the paper's structure.

References

[1] M. AdelsonVelskii and E. M. Landis. An algorithm for the organization of information. Technical report, DTIC Document, 1963.

[2] H. J. Boehm. Can Seqlocks get along with Programming Language Memory Models? `http://safari.ece.cmu.edu/MSPC2012/slides_posters/boehm-slides.pdf`, 2012.

[3] N. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. *PPoPP 2010*, 2010.

[4] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. NUMA-aware reader-writer locks. *PPoPP 2013*, 2013.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein". *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.

[6] A. Correia and P. Ramalhete. Scalable RW Lock with a single LongAdder. `http://concurrencyfreaks.com/2013/09/scalable-rw-lock-with-single-longadder.html`, 2013.

[7] CPP-ISO-committee". C++ Memory Order. `http://en.cppreference.com/w/c/atomic/memory_order`, 2013.

[8] A. Dragojevic, M. Herlihy, Y. Lev, and M. Moir. On the power of hardware transactional memory to simplify memory management. *PODC*, pages 99–108, 2011.

[9] V. L. Faith Ellen, Yossi Lev and M. Moir. Snzi: Scalable nonzero indicators. *PODC 07*, 2007.

[10] F. J. Group. fj.data.TreeMap. `http://functionaljava.googlecode.com/svn/artifacts/3.0/javadoc/fj/data/TreeMap.html`, 2013.

[11] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Foundations of Computer Science, 1978., 19th Annual Symposium on*, pages 8–21. IEEE, 1978.

[12] M. Herlihy. Optimistic concurrency control for abstract data types. *Operating Systems Review*, 21(2):33–44, 1987.

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

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

[15] P. E. McKenney. What is Read Copy Update. `https://www.kernel.org/doc/Documentation/RCU/whatisRCU.txt`, 2013.

[16] OpenJDK. StampedLock. `http://cr.openjdk.java.net/~chegar/8005697/ver.00/javadoc/StampedLock.html`, 2013.

[17] Oracle. Atomic Access. `http://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html`, 2013.

[18] P. Ramalhete and A. Correia. Distributed Cache Line Counter. http://concurrencyfreaks.com/2013/08/concurrency-pattern-distributed-cache.html, 2013.

[19] P. Ramalhete and A. Correia. Distributed Cache-Line Counter Scalable RW-Lock. http://concurrencyfreaks.com/2013/09/distributed-cache-line-counter-scalable.htm, 2013.

[20] P. Ramalhete and A. Correia. ScalableStampedlock. http://sourceforge.net/projects/ccfreaks/files/java/src/com/concurrencyfreaks/locks/ScalableStampedRWLock.java/download, 2013.

[21] P. Ramalhete and A. Correia. Combining the Stampedlock and LongAdder to make a new RW-Lock. http://concurrencyfreaks.com/2013/09/combining-stampedlock-and-longadder-to.html, 2013.

[22] J. Torrellas, H. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *Computers, IEEE Transactions on*, 43(6):651–663, 1994.

[23] Wikipedia. Concurrency control algorithms. http://en.wikipedia.org/wiki/Category:Concurrency_control_algorithms, 2013.

A. Appendix

Source code in Java and C++11 is available on Sourceforge as part of the Concurrency Freaks Library

`http://sourceforge.net/projects/ccfreaks/`

The classes used in this paper can be found under the folder `papers/LeftRight`: com.concurrencyfreaks.papers.LeftRight:

RWLockTreeSet.java
LRScalableTreeSet.java
LRScalableTreeSetNV.java
LRScalableTreeSetRV.java
LRScalableTreeSetOptimistic.java
BenchmarkTreeSetFullRebalance.java
BenchmarkTreeSetLatency.java

Figure 13. On the electronic version of this document, this figure shows an animation of the Writer and Reader's state machine and their interaction.

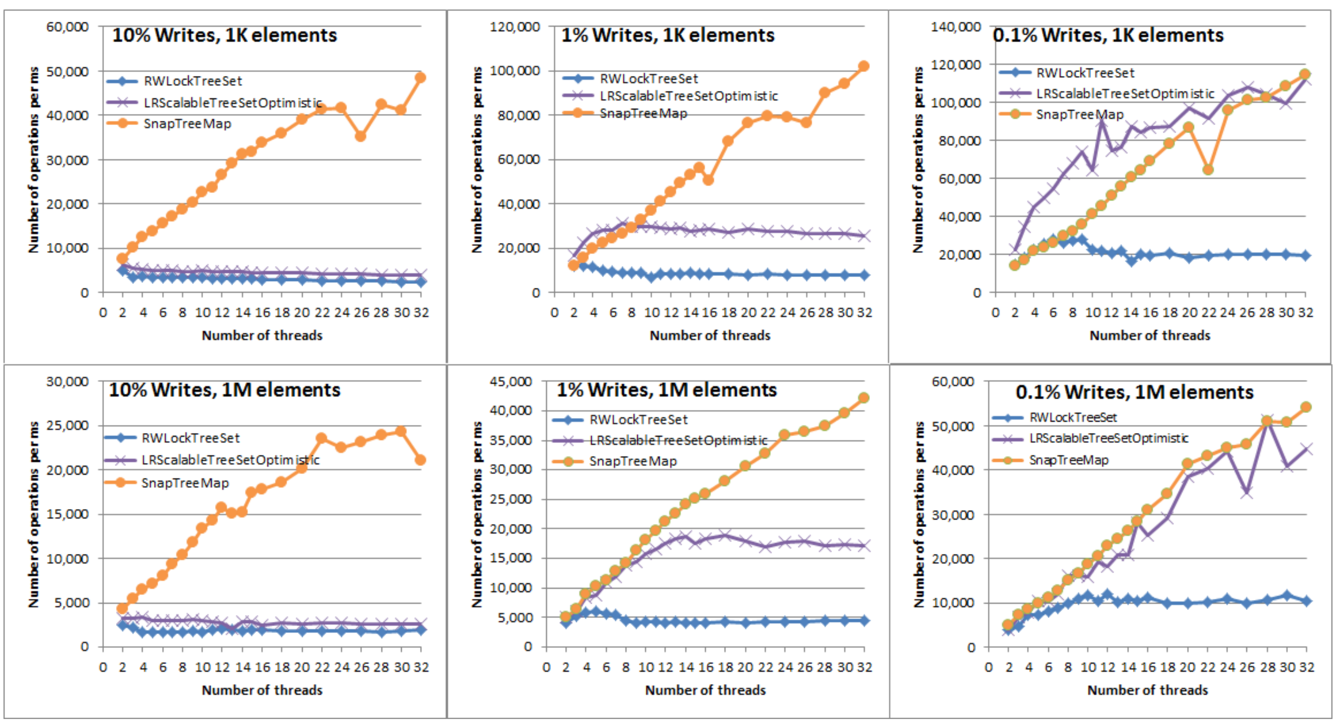

Figure 12. Each plot shows the throughput of the different techniques with 10^3 or 10^6 elements for 10%, 1%, and 0.1% Writes.