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

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

Andreia Correia
Concurrency Freaks
andreiacraveiroramalhete@gmail.com

Abstract

Concurrency control mechanisms ensure that correct results for concurrent operations are generated, while getting those results as quickly as possible. In this paper, we describe a new concurrency control algorithm with Blocking Starvation-Free write operations and Wait-Free Population Oblivious read operations, which we named the Left-Right algorithm.

We show a new pattern where this algorithm is applied, which requires using two instances of a given resource, and can be used for any data structure, allowing concurrent access to it, similar to a Reader-Writer lock, but in a non-blocking manner for reads. Unlike other non-blocking techniques, like Copy-On-Write, the Left-Right pattern does not need an automatic Garbage Collector (GC), making it viable in languages that don't have an available GC. We present several variations of the Left-Right pattern, with different versioning mechanisms and state machines. In addition, we constructed an optimistic approach that can reduce synchronization for reads.

To demonstrate the usefulness of this algorithm and the new pattern, we applied the Left-Right pattern to the mutable TreeSet implementation of the Java library and compared its performance with other concurrent tree implementations. 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, when compared with the best of the known implementations.

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 without the need of a GC, is to use a Reader-Writer lock. Recent developments have improved throughput and scalability for read operations [4, 5, 18, 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 [15]. 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 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. Other techniques exist which also use multiple instances and have lock-free properties [22, 26].

Another alternative technique is Read-Copy-Update (RCU) [16], which has the advantage of being easy to use, because from a semantic point of view, it can be deployed in a way similar to a reader-writer lock, although with different guarantees of consistency than a reader-writer lock. Unfortunately, the RCU technique requires special runtime support, which currently is provided only for native builds in Linux, and although a User-Space RCU library exists [7] (URCU), there is no support for it in on the JVM (i.e. Java, Scala). Notice also that in order to obtain wait-free or lock-free progress conditions with URCU requires the usage of a lock-free or wait-free algorithm for the desired data structure.

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 provides wait-free guarantees for read operations, while previously known algorithms [27] are blocking or lock-free.

The Left-Right pattern 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, an 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 the Writers are starvation-free [14], and that it has a small overhead for Reader synchronization, makes it ideal for usage in *write-few-read-many* scenarios.

	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.

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

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-side instance (`rightInstance`) while the Readers read the left-side instance (`leftInstance`), and once the Writer completes the modification, 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 modification on the `leftInstance`.

The read operations will run in the `leftInstance` in a non-exclusive mode, while a single Writer modifies the `rightInstance`, or vice-versa. Before exiting, the Writer will apply the same operation to the second data structure (or object), 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.

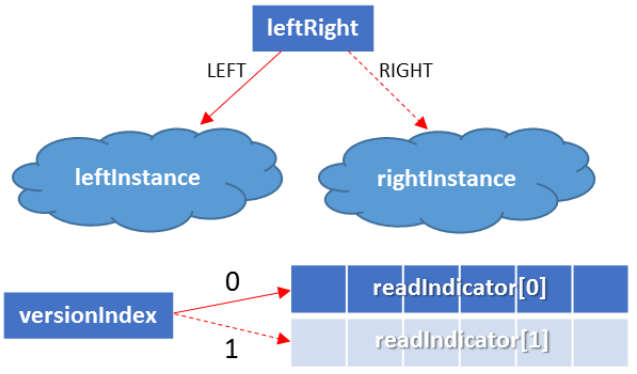

Figure 1. Components of the Left-Right concurrency control.

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 instance the Readers should go into; a `versionIndex` variable, which is modified by the Writer, functioning like a *timestamp*; and a Reader's indicator, `readIndicator`, that allows each Reader to *publish* the `versionIndex` it read.

It is worth mentioning that `versionIndex` is indeed a timestamp, even if in the Left-Right pattern this timestamp can be re-used once the Writer guarantees that there is no Reader using it. The toggle between timestamps is useful because it allows for the usage of counters as `readIndicators`, one counter per timestamp, in this case the timestamp can only be 0 or 1. In section 3 we will show variants of the Left-Right pattern where the `versionIndex` is always incrementing, thus implying that the timestamp is never re-used.

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

A simple implementation for the `readIndicator` is to use two single atomic synchronized counters, one per `versionIndex`. Several implementations of `readIndicator` are known, with either wait-free or lock-free progress conditions [9, 19, 20], and even NUMA-aware [4].

In the presented pseudo-code for the `readIndicator` shown in Algorithm 2, 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 that is unique to each thread. Each entry of `readersVersion[][]` is placed in its own cache line, using padding and alignment, to avoid false sharing [25] 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.

In the next section we will present the concurrency control algorithm that makes it possible for both Writer and Readers to perform concurrently and in isolation on both left and right instances.

2.1 Left-Right Concurrency Control Algorithms

2.1.1 Read operations - `contains()`

In the Left-Right pattern, read-only operations do `arrive()` and `depart()` on one of the `readIndicators` based on the value of `versionIndex`, and then perform their read access on one of the two instances based on the value of `leftRight`, as can be seen in Algorithm 1. This behavior of using the *most recent* of the two instances to do the read-only operations, is similar to the COW technique in which, many instances of the same data structure may exist in memory, but the read-only operation will get the reference to the most recent one (at that point in time) and use it.

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()`. As shown in Algorithm 3, mutable operations start by acquiring the lock on `writersMutex` to guarantee mutual exclusivity of Writers, and then modify the instance opposite to the one currently referenced by `leftRight`. The Writer then toggles the `leftRight`, waits for unfinished Readers on the new `versionIndex`, then toggles the `versionIndex`, and waits again for Readers on the previous value of `versionIndex` before modifying the other instance.

Algorithm 1 Algorithm for read operations - contains()

```
1: function CONTAINS(Key)
2:     localVersionIndex = versionIndex.get();
3:     readIndicator.arrive(localVersionIndex);
4:     if leftRight.get() == LEFT then
5:         Value = leftInstance.contains(Key);
6:     else
7:         Value = rightInstance.contains(Key);
8:     end if
9:     readIndicator.depart(localVersionIndex);
10:    return Value;
11: end function
```

Algorithm 2 An implementation of readIndicator

```
1: function READINDICATOR.ARRIVE(X)
2:     tlsEntry = ThreadLocal.get();
3:     readersVersion[X][tlsEntry.threadIndex].set(READING);
4: end function

5: function READINDICATOR.DEPART(X)
6:     tlsEntry = ThreadLocal.get();
7:     readersVersion[X][tlsEntry.threadIndex].set(NOT_READING);
8: end function

9: function READINDICATOR.ISEMPTY(X)
10:    for i in readersVersion[X].size do
11:        if readersVersion[X][i].get() == READING then
12:            return false;
13:        end if
14:    end for
15:    return true;
16: end function
```

Algorithm 3 Algorithm for write operations - modify()

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

2.1.3 Progress Conditions

We show that a given method is wait-free population oblivious [14] by showing that every call to that method finishes its execution in a finite number of steps, independently of the number of threads.

The read operation shown in Algorithm 1 has no loops, and always executes in a constant number of steps, thus ensuring 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, although 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, making the Writer **starvation-free** with respect to Readers. The Writer has to acquire a mutually exclusive lock `writersMutex`, which means its progress condition is **blocking**. If the mutex used in `writersMutex` provides starvation-free properties, then the write operation will be starvation-free because the Left-Right algorithm guarantees that a Writer will not be starved by Readers, and a starvation-free mutex guarantees that a Writer will not be starved by other Writers.

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 Java JVM or the default on C11/C++1x/D/Scala/Go. This means that `Atomic.get()` (loads) are atomic and executed with an acquire-barrier, and `Atomic.set()` (stores) are atomic and imply a release-barrier [6].

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 2, and then load the value of `leftRight` in line 4, whilst in Algorithm 3, the Writer will first set `leftRight` to the new value in line 9, and then set `versionIndex` to the new version in line 15.

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.

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 *isolation*. 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 explaining figures 4 and 5, keeping in mind that the first write operation is a special case of figure 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.

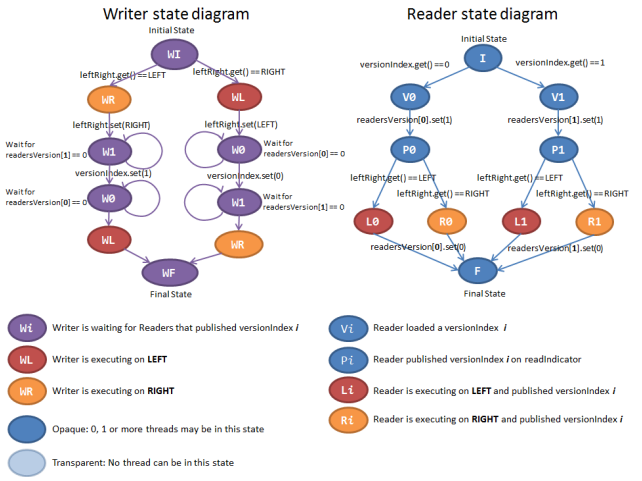

Figure 2. Full state machines for the Writer and Readers. Some of the transitions on the Reader's diagram are not allowed when the Writer is in certain states.

It is interesting to notice that every time a write operation modifies the second instance, it does so with the guarantee that it performs in isolation, and this guarantee is still valid when the next Writer starts, which allows it to perform its modification immediately on its first instance without having to do any waiting for Readers.

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 Writer.

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 there is a finite number of Readers that can publish `versionIndex` 0. 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 have finished, the Writer can proceed to toggle `versionIndex` from 1 to 0, the timestamp 0 can now be re-used. Figure 4.C shows that all new read operations will load `versionIndex` 0 and `leftRight` LEFT. But there are still *old* Readers 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 Readers will load `versionIndex` 0 which means there is a finite number of read operations that can publish on the `readIndicator` `readersVersion[1]` thus guaranteeing that the Writer will not

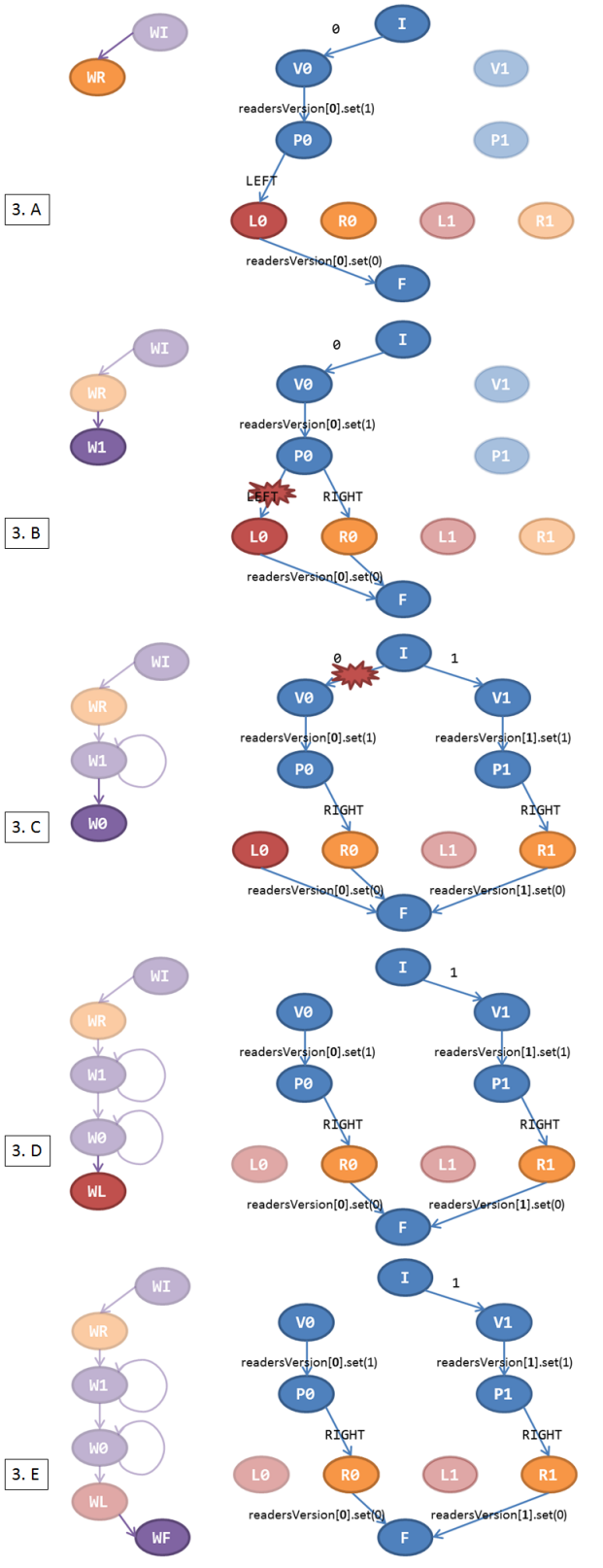

Figure 3. For the first Writer, the `leftRight` starts at LEFT and `versionIndex` at 0.

starve. Again, the procedure 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. As shown in the state machine on figure 4.D, there can be Readers that are on state **V1** and as such, loaded `versionIndex 1` but did not yet publish on the `readIndicator`, and can transition to **P1** and **L1**, and are not seen by the Writer. Finally the Writer can proceed to modify the second instance because both states **R0** and **R1** are empty, so all Readers 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.

2.3 Linearizability

As far as the Writers are concerned, during the `modify()` operation described in Algorithm 3, 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.

3. Algorithm Variants

We have developed multiple variants of the original algorithm, each with a different number of synchronization primitives. 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, implying a memory usage 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 6. The main difference from the basic 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`. Readers start by setting their state to `READING`, then read the current value of `leftRight` and then they set their state accordingly to the `leftRight` they have read, as shown in Algorithm 4.

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. Also, in this variant, the Writer is no longer starvation-free with respect to Readers.

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 5. To help distinguish between active and inactive Readers, 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

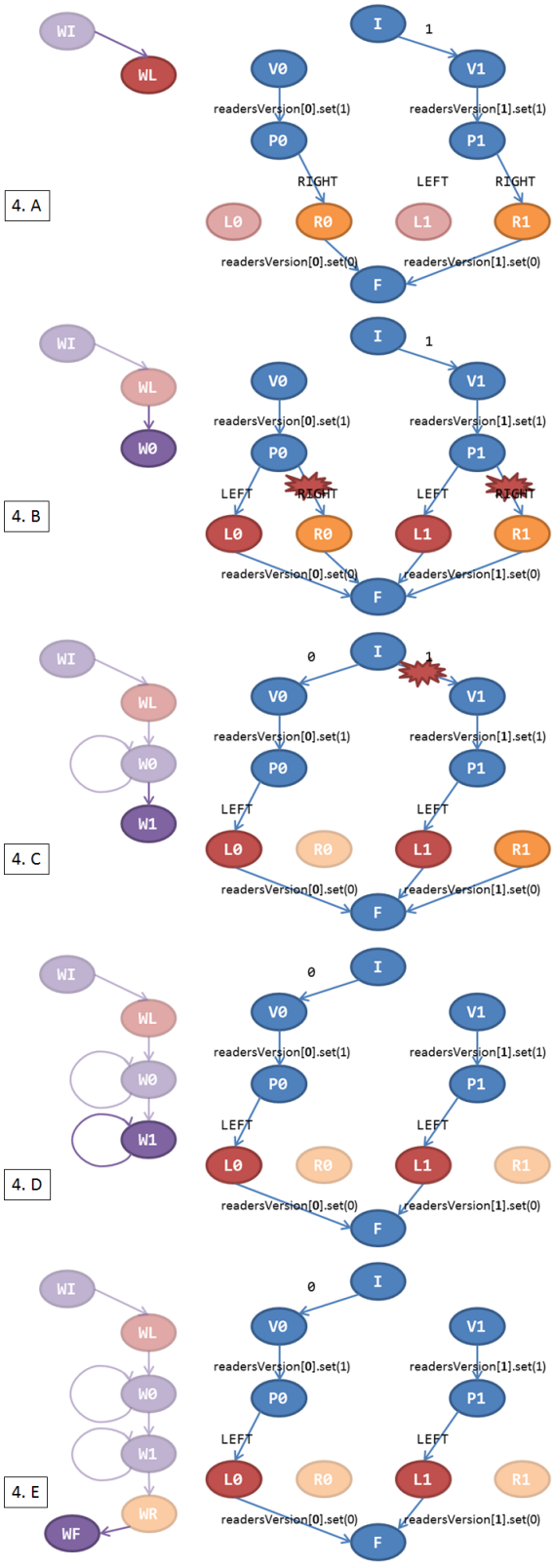

Figure 4. State machine of Writer and Readers when the Writer starts on the `rightInstance`.

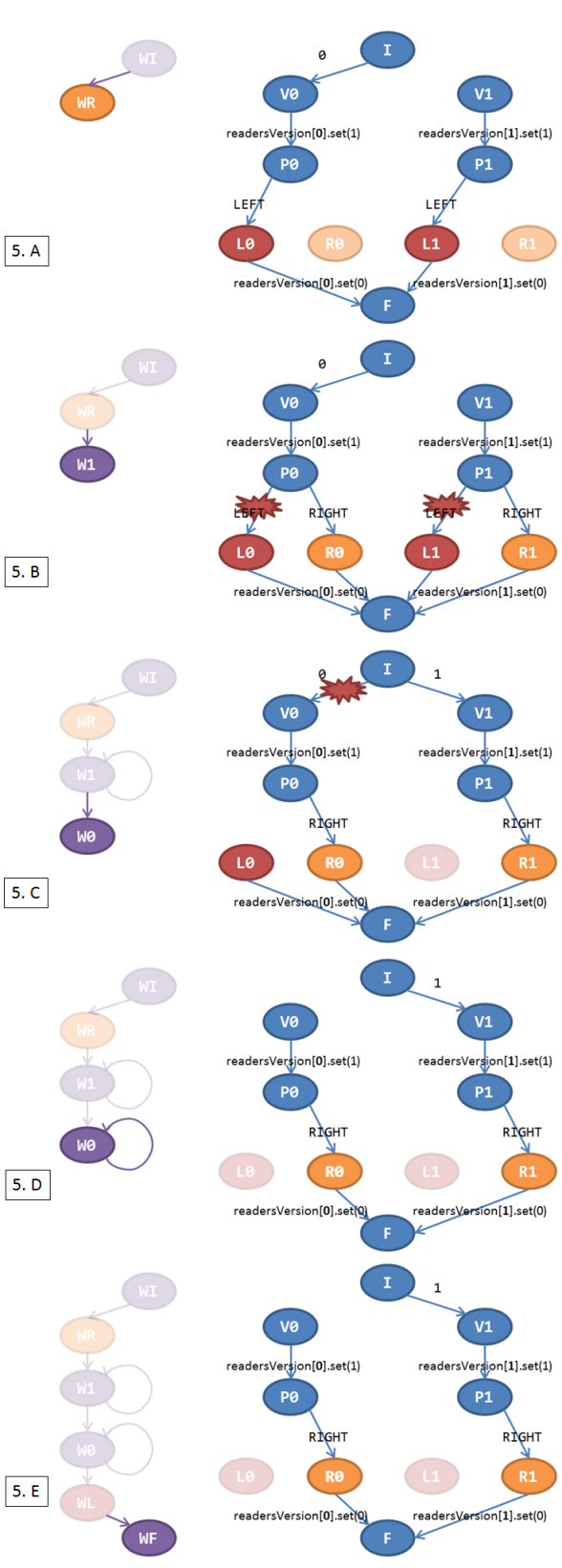

Figure 5. State machine of Writer and Readers when the Writer starts on the `leftInstance`.

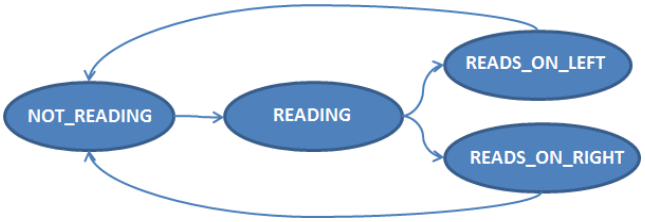

Figure 6. Reader's state machine for the NV algorithm. The transitions between states are atomic.

Algorithm 4 NV Algorithm

```
1:  function CONTAINS(Key)
2:      readIndicatorNV.setState(READING);
3:      if leftRight.get() == LEFT then
4:          readIndicatorNV.setState(LEFT);
5:          Value = leftTree.contains(Key);
6:      else
7:          readIndicatorNV.setState(RIGHT);
8:          Value = rightTree.contains(Key);
9:      end if
10:     readIndicatorNV.setState(NOT_READING);
11:     return Value;
12: end function

13: function MODIFY(Key)
14:     writersMutex.lock();
15:     localLeftRight = leftRight.get();
16:     if localLeftRight == LEFT then
17:         rightTree.modify(Key);
18:     else
19:         leftTree.modify(Key);
20:     end if
21:     leftRight.set(-localLeftRight);
22:     readIndicatorNV.waitIfReadingOrArgument(localLeftRight);
23:     if -localLeftRight == LEFT then
24:         rightTree.modify(Key);
25:     else
26:         leftTree.modify(Key);
27:     end if
28:     writersMutex.unlock();
29: end function
```

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.

The Writer has to *scan* the version of each Reader, 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, a method we named `waitIfReadingOrArgument()`.

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.

3.3 Optimistic Read

We will now show an optimistic approach to the Left-Right pattern in Algorithm 6. The idea is that the Reader *assumes* that no Writer is changing the instance where the Reader is executing, which will be the case if the `versionIndex` has not changed.

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

Mechanisms such as this one are not new [12], and have been used before for Reader-Writer locks [17] and directly on data struc-

Algorithm 5 RV Algorithm

```
1: function CONTAINS(Key)
2:     tlsEntry = ThreadLocal.get();
3:     readIndicatorRV.setState(1-tlsEntry.localVersion);
4:     if leftRight.get() == LEFT then
5:         Value = leftInstance.contains(Key);
6:     else
7:         Value = rightInstance.contains(Key);
8:     end if
9:     readIndicatorRV.setState(tlsEntry.localVersion-1)
10:    return Value;
11: end function

12: function MODIFY(Key)
13:    writersMutex.lock();
14:    localLeftRight = leftRight.get();
15:    if localLeftRight == LEFT then
16:        rightInstance.modify(Key);
17:    else
18:        leftInstance.modify(Key);
19:    end if
20:    leftRight.set(-localLeftRight)
21:    readIndicatorRV.waitUntilIncrementOrNegative()
22:    if -localLeftRight == LEFT then
23:        rightInstance.modify(Key);
24:    else
25:        leftInstance.modify(Key);
26:    end if
27:    writersMutex.unlock();
28: end function
```

tures [3]. Similarly to those mechanisms, this variant is not as generic as the previously described variants of the Left-Right pattern, because it allows a Reader and a Writer to run on the same instance at the same time, which may impact the invariants of the data structure [24]. In addition, this kind of approach requires automatic GC and the underlying object or data structure must have atomicity guarantees on its members.

3.4 Read synchronized operations

Depending on the variant of the algorithm, we get a different finite number of atomic synchronized operations when doing a read:

- Basic Left-Right: 2 get() + 2 set()
- NV - No Version: 1 get() + 3 set()
- RV - Reader's Version: 1 get() + 2 set()
- Optimistic: min 3 get(), max 5 get() + 2 set()

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

3.5 Left-Right technique as a Reader-Writer Lock

Although the Left-Right pattern 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.

On a functional language, a lambda with the `modify()` function (write operation) can be passed and applied twice, once on `leftInstance` and once on `rightInstance`. On non-functional languages, we can separate the Algorithm 3 into three methods: the first method implements the code from line 2 to 4 and returns the instance to be modified first; the second method implements the code from lines 9 to 19 and returns the second instance to be modified; the third method unlocks the `writersMutex`. Algorithm 1 can be separated into two methods, where the first one implements the

Algorithm 6 Optimistic algorithm

```
1: function CONTAINS(Key)
2:     localVersionIndex = versionIndex.get();
3:     if leftRight.get() == LEFT then
4:         Value = leftInstance.contains(Key);
5:     else
6:         Value = rightInstance.contains(Key);
7:     end if
8:     releaseBarrier();
9:     newLocalVersionIndex = versionIndex.get();
10:    if newLocalVersionIndex == localVersionIndex then
11:        return Value;
12:    else
13:        return containsKeyAlgorithm1();
14:    end if
15: end function

16: function MODIFY(Key)
17:    writersMutex.lock();
18:    localLeftRight = leftRight.get();
19:    if localLeftRight == LEFT then
20:        rightTree.modify(Key);
21:    else
22:        leftTree.modify(Key);
23:    end if
24:    leftRight.set(-localLeftRight);
25:    prevVersionIndex = versionIndex.get();
26:    nextVersionIndex = prevVersionIndex + 1;
27:    readIndicator.waitUntilEmpty(nextVersionIndex % 2);
28:    versionIndex.set(nextVersionIndex);
29:    readIndicator.waitUntilEmpty(prevVersionIndex % 2);
30:    if -localLeftRight == LEFT then
31:        rightTree.modify(Key);
32:    else
33:        leftTree.modify(Key);
34:    end if
35:    writersMutex.unlock();
36: end function
```

code from lines 2 to 4, while the second implements the lines 9 and 10. If needed, a thread-local-storage variable can be used to pass the value read for the `versionIndex` between the two functions of the read operation. The end result of this transformation can be seen in Algorithm 7, and a code snippet with an example usage in Algorithm 8.

4. Performance Evaluation

As a purely illustrative example, we applied the Left-Right pattern to a java.util.TreeSet, measured its performance and latency, and compared with other well known solutions. A set of performance tests were conducted on a dual Opteron 6272 with a total of 32 cores, running Windows 7 with JDK 8 (u20). We executed 7 individual runs for each of the data structures presented below, and plotted the median of the operations per millisecond, where the value of operations per millisecond is an average over a period of 30 seconds, which is presented on Figure 7.

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()` followed 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 rebalanced often.

- **RWLockTreeSet**: java.util.TreeSet protected with a Reader-Writer lock ScalableRWLock [23]. The ScalableRWLock is a freely available lock that uses the C-RW-WP lock described in [4] with a new `readIndicator` that combines a Concur-

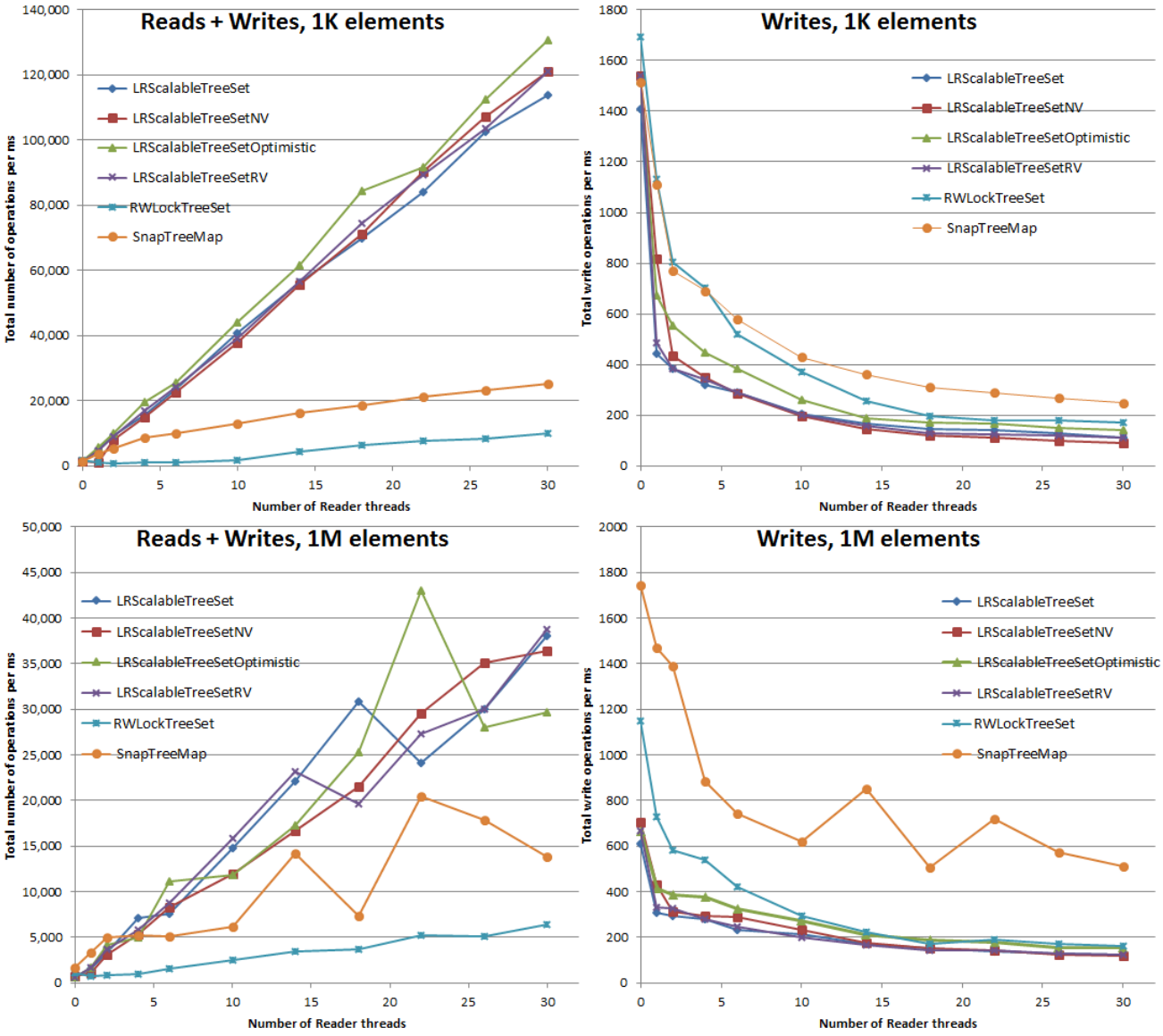

Figure 7. Total operations per millisecond as function of the number of Readers when 2 Writers are running on a set of 10^3 elements

rentLinkedQueue with an array. 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 basic Left-Right technique described in Algorithms 1 and 3.
- **LRScalableTreeSetNV**: java.util.TreeSet with the Left-Right technique (No Version) without using a `versionIndex`, as described in section 3.1.
- **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 section 3.2.
- **LRScalableTreeSetOptimistic**: java.util.TreeSet with the optimistic approach described in section 3.3.
- **SnapTreeMap**: SnapTreeMap from the edu.stanford.ppl.concurrent package with hand-over-hand optimistic validation and a relaxed balanced tree. 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 of a relaxed balanced tree, and partially external trees for deletion, which minimizes the tree's rebalance frequency. All operations of the SnapTreeMap are blocking.

All our implementations of the Left-Right algorithms use the java `synchronized` keyword to guarantee exclusivity between write operations, represented on the algorithms 3, 4, 5, 6 as `writersMutex.lock()` and `writersMutex.unlock()`. Also, the `readIndicator` uses a ConcurrentLinkedQueue with an array, the same as the one used in ScalableRWLock [23].

Initially, we tried to compare with an implementation using the COW pattern, based on an immutable TreeMap [10], but runs with 10^3 elements gave low performance, and runs with 10^6 elements

Algorithm 7 Making a *kind of* RW-Lock from Left-Right

```
1: function READERLOCK(leftInstance, rightInstance)
2:     tlsEntry = ThreadLocal.get();
3:     tlsEntry.localVersionIndex = versionIndex.get();
4:     readIndicator.setState(tlsEntry.localVersionIndex, READING);
5:     if leftRight.get() == LEFT then
6:         return leftInstance;
7:     else
8:         return rightInstance;
9:     end if
10: end function

11: function READERUNLOCK( )
12:     tlsEntry = ThreadLocal.get();
13:     readIndicator.setState(tlsEntry.localVersionIndex, NOT_READING);
14: end function

15: function WRITERLOCK(leftInstance, rightInstance)
16:     writersMutex.lock();
17:     if leftRight.get() == LEFT then
18:         return rightInstance;
19:     else
20:         return leftInstance;
21:     end if
22: end function

23: function WRITERTOGGLE( )
24:     localLeftRight = leftRight.get();
25:     leftRight.set(-localLeftRight);
26:     prevVersionIndex = versionIndex.get();
27:     nextVersionIndex = (prevVersionIndex + 1)%2;
28:     readIndicator.waitUntilEmpty(nextVersionIndex);
29:     versionIndex.set(nextVersionIndex);
30:     readIndicator.waitUntilEmpty(prevVersionIndex);
31:     if -localLeftRight == LEFT then
32:         return rightInstance;
33:     else
34:         return leftInstance;
35:     end if
36: end function

37: function WRITERUNLOCK( )
38:     writersMutex.unlock();
39: end function
```

Algorithm 8 Example of using a Left-Right pattern instead of a Reader-Writer lock to protect an object

```
1: instance = READERLOCK(leftInstance, rightInstance);
2: instance.someReadOnlyOperation();
3: READERUNLOCK( );
   ...
4: firstInstance = WRITERLOCK(leftInstance, rightInstance);
5: firstInstance.someWriteModifyOperation();
6: secondInstance = WRITTERTOGGLE(leftInstance, rightInstance);
7: secondInstance.someWriteModifyOperation();
8: WRITERUNLOCK( );
```

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 as can be seen on the leftmost plots of Figure 7.

Regarding write operations, the algorithm with the highest performance is the SnapTreeMap, which can be explained by: the SnapTreeMap uses a relaxed balanced 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 or write operations.

4.1 Workload Benchmark

As mentioned before, although the Left-Right technique benefits from dedicated Reader threads, 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 two reads or two write operations are done, where the two write operations consist of a `remove()` done on the ith-element followed by an `add()` on the ith-element plus `numElements`. For example, the plot on Figure 4.1 with 10% Writes means that, on average, for every write operation there were nine read operations. Similarly to the performance benchmarks on the previous section, each data point in Figure 4.1 is the median of 7 runs.

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.

Notice that these kind of mixed task benchmark, 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

Low latency is an important characteristic when choosing techniques to use in real-time systems. 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 started with a 20 second warmup period and ran for 10^4 seconds each time, 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%	38	< 1	2
99.9%	67	< 1	5
99.99%	106	8	11

Table 2. Latency measurements in microseconds for the `contains()` method on a tree with 10^6 elements.

Using the RWLockTreeSet as an example, the table can be read as follows: 99% of the calls to the contains() method take 38 microseconds or less to complete. Table 2 shows a good latency performance for `contains()` operations on the LRScalableTreeSet, where according to our measurements, 99.9% of the operations take less than 1 microsecond to complete.

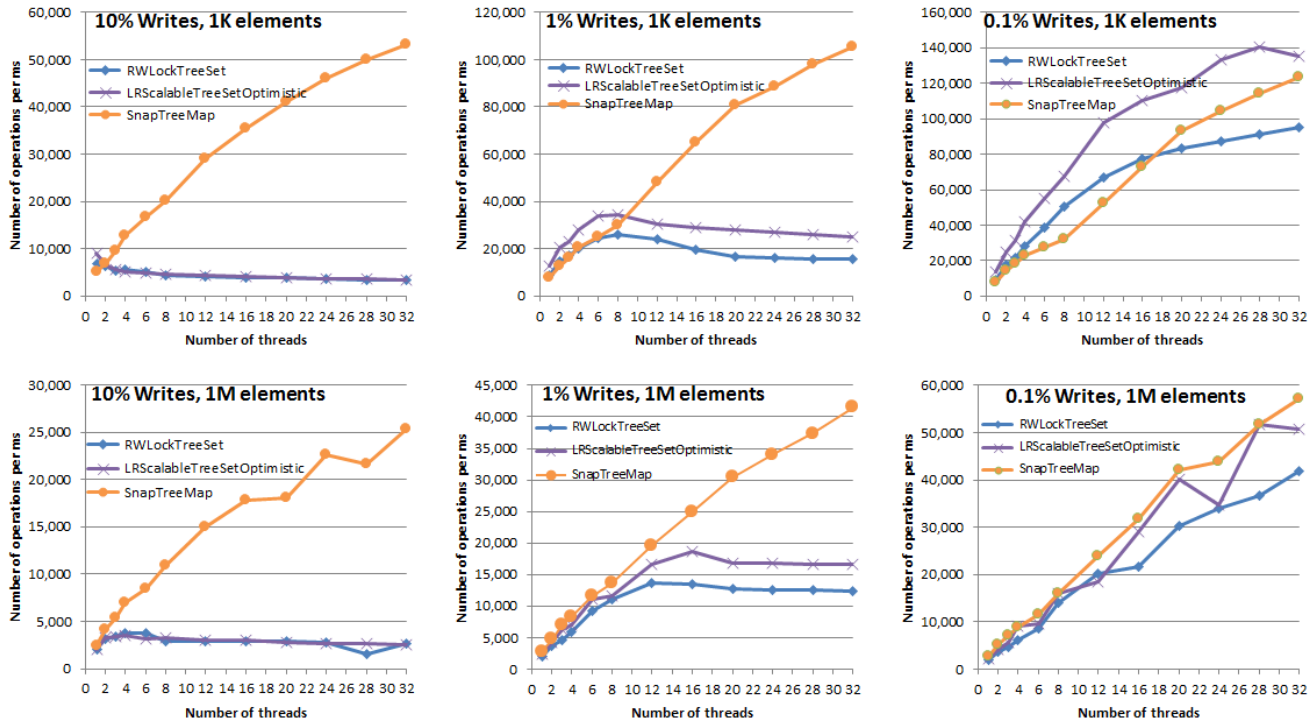

Figure 8. Each plot shows the throughput of the different techniques with 10^3 or 10^6 elements for 10%, 1%, and 0.1% Writes.

5. 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, which is, as far as we know, the only technique with such characteristics and performance. 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, and Starvation-Freedom to write operations. A practical implementation for a concurrent tree using the Left-Right pattern 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.

Acknowledgements

We wish to thank *anonymous reviewer 1* on the Scala conference for his encouragement and contribution to the linearization section, and to thank Davide Cuda and Maurice Herlihy for helpful comments on a preliminary version of this paper.

Source code in Java is available on GitHub as part of the Concurrency Freaks Library
`https://github.com/pramalhe/ConcurrencyFreaks/`
The classes used in this paper can be found under the folder `Java/com/concurrencyfreaks/papers/LeftRight`:
com.concurrencyfreaks.papers.LeftRight:
ScalableRWLockTreeSet.java
LRScalableTreeSet.java
LRScalableTreeSetNV.java
LRScalableTreeSetRV.java
LRScalableTreeSetOptimistic.java
BenchmarkTreeSetFullRebalance.java
BenchmarkTreeSetLatency.java

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] 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.

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

[7] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of read-copy update. *Parallel and Distributed Systems, IEEE Transactions on*, 23(2):375–382, 2012.

[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] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 30 Corporate Drive, Suite 400, Burlington, MA 01803, USA, 2008.

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

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

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

[18] 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.

[19] 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.

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

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

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

[23] P. Ramalhete and A. Correia. ScalableRWLock. http://sourceforge.net/projects/ccfreaks/files/java/src/com/concurrencyfreaks/locks/ScalableRWLock.java, 2013.

[24] P. Ramalhete and A. Correia. StampedLock.tryOptimisticRead() and Invariants. http://concurrencyfreaks.com/2013/11/stampedlocktryoptimisticread-and.html, 2013.

[25] H. Sutter. Eliminate False Sharing. http://www.drdobbs.com/parallel/eliminate-false-sharing/217500206, 2009.

[26] D. Vyukov. Improved SeqLock. http://www.1024cores.net/home/lock-free-algorithms/reader-writer-problem/improved-lock-free-seqlock, 2013.

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