Linearizable size

David Agassi
308369677
davidio.agassi@gmail.com

April 23, 2018

Abstract

On many ADTs it is common practice to use a **size** attribute that states the amount of elements stored in the ADT. There are 2 common ways of calculating this amount. The first is to iterate over the ADT and count for each call. The second way would be storing a variable by "the side" of the ADT and updating it with each update of the ADT depending on the operation performed. On concurrent linearizable ADTs, the **size** attribute is always defined, but not always easily accessed. Since the **size** varies in an unknown order and it concerns the whole ADT both methods face a similar challenge regarding linearizability, How can you test the whole ADT without blocking it?.

We will focus on the second method I will call "accumulated size" since it is being accumulated by the operations performed on the ADT. The main challenge is having the **size** constantly updated regarding the multi-threaded operations performed and keeping a single value. In this solution we relax the demands and keep 2 values. We will establish two accumulative **size** attributes, "max size" and "min size" both being altered by the working threads. At each point both values agree we know the **size** value precisely.[1]

The key of the solution is having, for every operation performed, the values in "max size" and "min size" moving further away at the beginning of the operation, and restoring the difference at the end of each operation. For example the **add** operation would increase "max size" at the beginning, and once the **add** operation completes, if it succeeds increment "min size" and if it fails, decrement "max size". Once "max size" and "min size" read the same value[2], it shall return it. I will show that if the counters "max size" and "min size" are linearizable, the **size** attribute also is.

In this paper I will create a Java Abstract Class, and implement a few wrappers to common ADTs[3] extending the class in order to test the idea and evaluate the performance of such operation. I will use the Java LongAdder as the counters, which has a method **sum()** that returns the adders value across threads. This method is a simple sum of differences

[1] Based on "KiWi - OCC Key-Value Map" by Assaf Yifrach, Niv Gabso
[2] in practice this could be optimized.
[3] SkipList LockFreeAVL ConcurrentHMAP

across threads and isn't linearizable. I will create two counters based on the LongAdder, one not linearizable, and the other is[4]linearizable. The second counter should be more prone to starvation.

Contents

[4]using AtomicBollean

Part I
Theoretical Explanation

1 Preface

1.1 Motivation

On many ADTs it is common practice to use a **size** attribute that states the amount of elements stored in the ADT. Typically, in many simple cases, the **size** attribute is easy to define and calculate. If there are no concurrent operations manipulating the ADT, the size is defined as the amount of elements stored in the ADT at each point in the executions history, and can be calculated by iterating over the ADT, therefor blocking it. In concurrent ADT's blocking the whole ADT is bad practice and beats the point. If all operations on the ADT are linearizable, then for all execution the **size** is always well defined, mathematically speaking, but is hard to calculate. As there is a sequential history that matches each run the size is defined as above, as the amount of elements stored in the ADT for each point in the sequential history. But now it is not permitted to block the whole ADT in order to calculate the **size**.

The first insight is that the **size** may depend on all the elements, but the changes in it are easy to calculate from the operations. If we stored a counter and for every operation, as it executed in the linearization point, we would update it with regard to it's success/fail, That counter would store the **size** correctly. This kind of operation looks like some kind of CAS, but current architectures don't support this kind of operation, and this would also create a bottle-neck on the **size**'s memory location, effectively blocking as well the whole ADT.

We will further relax the problem by calculating the **size** on demand instead of saving it in a specific location, making it a linearizable operation. The first implication is that for some stages in the execution, even though the size is theoretically defined, it is unreachable under the execution's timing. On the other hand, we get a non-blocking correct **size** of the ADT. This is very similar to many linearizable operations that have some king of validation and won't return until they succeed.

Thus allowing us to wrap any concurrent ADT and being able to get it's size without blocking it.

1.2 Solution's Outline

An attribute is usually considered as a single memory location that is easily accessed. In this solution we will split the **size** attribute into 2 different attributes, minSize and maxSize and once they agree on the size e.g $minSize = maxSize$[5], we can determine that **size** is equal to them as well. Now each attribute becomes a counter that is shared across threads and is updated locally. On an event of

[5] Practically $minSize \geq maxSize$

Size() being called, the counters are summed across threads and compared until they agree.

These attributes represent the upper and lower bound of the ADT's size. For example, on insertion, the upper bound maxSize is incremented at the beginning, and once it ended, if it succeeded, the lower bound is incremented, else, the upper bound is decremented back to it's original value. By doing this, the upper and lower bounds always pull away at the beginning of each operation and pull closer at the end. If they agree, this also means that at this point no operation is altering the ADT[6].

Depending on the linearizability of the counters, this solution is linearizable.

2 Sizable ADT wrapper

2.1 Counters

At the base for this function lay two counters. A counter in this context is basically a memory location that is updated by adding and subtracting instead of just overriding the value. This allows us to decentralize the actual location of of the counter and have every thread change the counter as it sees fit. In order to read the counter, you need to run across all threads collect the differences it made. The sum of all the differences make up the counter's value.

Summing up a counter could be done in a non linearizable fashion as just running across all threads with no validation, or in a linearizable fashion. For the sake of this section we assume our counter is linearizable.

2.2 Linearizable getSize

Each ADT has a insert/add method that may increase it's size and a delete/remove that may decrease it. Assuming these operation returns true on success and false on fail, we are able to wrap these methods as following.

[6] Not regarding the optimization $\geq$ in previous footer.

Algorithm 1 Size wrapper

```
def add(value){
    maxaddr.inc()
    if(ADT.add(value)):
        minaddr.inc()
    else:
        maxaddr.dec()
}
def remove(value){
    minaddr.dec()
    if(ADT.remove(value)):
        maxaddr.dec()
    else:
        minaddr.inc()
}
```

Notice that at the beginning of each operation the counters move apart and at the end they come closer depending on the result. We will define the first effect as the initial inc/dec of a counter at the beginning of the call, and the second effect as the later "fix" to the counters. Now we can say that the first effects drive the counters apart and the second effects bring them closer. We can think of these counters as the upper and lower bound on the size of the ADT.

Now we will use these counters to evaluate the size of the ADT.

Algorithm 2 getSize

```
def getSize(){
    max = maxaddr.sum()
    while(true){
        min = minaddr.sum()
        if min>=max:
            return max
        max = maxaddr.sum()
        if min>=max:
            return min
    }
}
```

Using the assumption from before we get that once both counters agree and size is returned we there was a point in time where the size **was**, as in actually defined, as the return value. When returning a value the deciding counter is the counter that was read first. For example if we return from the second IF statement the deciding counter is the minaddr counter. Agreeing is stating that the upper bound is smaller or equal to the lower bound, in this in between reading both bound there was an actual point in time where the ADT was at the size of the deciding counter. The key principle here is that although the exact linearization point is not known, it lays between both reads.

It is possible that between reads of both counter the size of the ADT has varied allot, but we know that it did have the value in the deciding counter in the meantime. For example, lets say that the deciding counter was minadder, therefor we know that the at the beginning there where ad least minaddr elements in the ADT. Also, the upper bound has receded below the lower bound, as is there where more **successful** removes then **potentials** adds in the meantime. We can conclude that the size was at one point the size in minadder.

3 Linearizable Counter

3.1 Motivation

The reason the counter need to be linearizable is that without it we can't find the exact sum at each time, and results may very of the correct sum. For example imagine a naive summation of a counter along threads. Now imagine the following sequence:

1. th1: starts add, maxaddr+=1
2. th2: starts remove minaddr-=1
3. th2: successful remove, maxaddr-=1
4. th1: successful add minaddr+=1

Now imagine that the size reads 1 and 4 only. This can happen if after 1 maxaddr is read correctly but minaddr is misread and misses 2. we get a wrong result from size, a read where it is bigger by 1 from any time in the middle.

3.2 Dirty bit method

We will use a atomic dirty bit that is set once the counter is tempered with and on summation it is cleared and tested at the end. This means we now have a single bottle neck for every counter, but it is atomic and is much more common practice.

Algorithm 3 Dirty-Bit Counter Wrapper

```
add(x){
    dirty.set(true);
    counter.add(x);
}
synchronized sum(){
    while (true){
        dirty.set(false);
        s = counter.sum();
        if(!dirty.get()){
            return s;
        }
    }
}
```

Now since before every tempering with the counter in each thread it notifies all the other thread of the change. Now on every sum, we can make sure no changes went undetected during the summing operation making it linearizable, as in we can know the correct sum regarding it being decentralized among threads at the moment it the function returned.

The Sum method has to be synchronized in order to prevent other sum operations over-ridding a set bit while the sum is performed in order so to keep the bit accessible only to mutations. this will add additional overhead to the function of locking (getting a lock) and queuing.

3.3 Semi-Sizable

In the future implementations we define the naive counter semi-linearizable and I will compare both performances.

Part II
Java implementation and Performance

4 Outline

4.1 Java Implementation

Java has two main ATD types: Collection, Map. Most standard ADT's extend them and so it will be useful to make a generic wrapper for them. I have created an abstract class Sizable that accepts a counter class that performs the getSize regarding to those counters, then extensions of that class that wraps both Java Collection and Java Map interfaces that use the counters in insertion end deletions. For counters anything that extends Java's LongAdder may be used as to create Sizable ADT's.

Also, I have created a linearizable counter as described before and wrapped the basic Java ConcurrentHashMap (Java Map) and Java ConcurrentQueue (Java Collection) for demonstration.

4.2 Test Methodology

I have adapted the given BitBucket repository[7] which contains a collection of concurrent algorithms and a test suit. I have adapted three[8] of them to run under my wrappers and added a timer test for the size operation.

5 Specifics

5.1 Sizable

An abstract class Sizable<C extends LongAdder>[9] that contains:

```
protected C minSize; protected C maxSize;
public long getSize()
```

5.1.1 extensions

Both extension wrap an ADT and implement the insertion and deletion methods by wrapping the ADT's method.

- class SizableCollection<E, A extends Collection<E>, C extends LongAdder> extends Sizable

[7] https://bitbucket.org/trbot86/implementations
[8] SkipList LockFreeAVL ConcurrentHMAP
[9] C stands for a counter class

  - private A collection;
  - public boolean add(E element)
  - public boolean remove(E element)

- class SizableMap<K, V, M extends Map<K,V>, C extends LongAdder> extends Sizable<C>
  - private M map;
  - public V put(K key, V value)
  - public V remove(K key)

5.2 Counters

Java has a naive concurrent counter LongAdder, I have extended it using an AtomicBoolean to create a linearizable counter.

class LinearizableCounter extends LongAdder:

- private AtomicBoolean dirty;
- public void add(long x)
- public long sum()

Now having both Counters and ADT's wrappers I was able to make Sizable and Semi-Sizable wrapper for basic Java ADT's such as ConcurrentHashMap and ConcurrentQueue.

5.3 Wrap For Test

5.3.1 Wrapping the ADT's

I have wrapped the following BitBucket repository [10] and implemented the following adapters using the SizableMap extension:

1. SkipList
   (a) SemiSizableSkipList
   (b) SizableSkipList
2. ConcurrentHashMap
   (a) SemiSizableConcurrentHashMap
   (b) SizableConcurrentHashMap
3. LockFreeAVL
   (a) SemiSizableLockFreeAVL
   (b) SizableLockFreeAVL

[10] https://bitbucket.org/trbot86/implementations (footer 7)

5.3.2 Testing the Size Operation

Also I have changes the running scheme to include size operations by manipulating the trail Ratio and adding a size attribute and call. In order to test the duration and throughput of the size operation I have timed[11] it. In order to create a valid comparison, the original ADT's size function was empty so we can neutralize the timing system calls and evaluate just the duration of the size function. Additionaly, I have run a test with no Size Operations in order to test the overhead of the Sizable extension.

5.3.3 Testing Scheme

I have run trails with the following parameters for each ADT:

1. ADT's: SkipList, ConcurrentHashMap, LockFreeAVL.
2. Wrappers: Original, SemiSizable, SemiSizable.
3. Key Ratios
   (a) Key size: 100, 1000, 1000000
   (b) ops ratios (ins_del_size): 33_33_34, 20_10_20, 45_45_10, 50_50_0
4. Threads: 1, 2, 4, 8, 16, 32.
5. Trails:
   (a) length: 10s.
   (b) repetitions: 8.
6. Test Server:
   (a) name: rack-mad-03
   (b) cores: 2
   (c) threads per core: 22
   (d) total threads: 44 (didn't use hyper-threads)

5.3.4 Linearization Testing

In order to test for the linearization of the solution, I have manipulated the tests getKeySum to evaluate the size of the ADT. In other words, for each operation the keySum added 1 on addition success and subcontracted 1on deletion success, and at the end compared to the getSize [12]. If all operations where detected correctly getSize should return the same size as the test expected.

[11] with System.nanoTime()
[12] in steady state.

6 Evaluation

6.1 expectation

6.1.1 Linearizability

We are expecting the size reported form getSize in steady state to be the sum of operations across all threads. I it fails at that, I might as well just not hand in this paper...

6.1.2 Performance

Wrapper overhead: The wrapper should slow down the Ops performed, but not by much as they are a constant overhead, $O(1)$. The Linearizable counter has a bottle-neck on the dirty bit that will slow the system down even further, as the number of threads grow, the overhead on the linearizable counter should grow.

Size Operation: The size operation should slow with the increase in the thread count[13], as it would fail more. The Linearizable counter should slow down significantly as it has to sum 2 Linearizable counters at a high rate, with the dirty-bit bottle-neck.

Key Range We are not expecting the key range to effect the rate of mutations to and therefor effecting the throughput.

6.2 results

6.2.1 Linearizability

All check-sums where valid.[14]

6.2.2 Performance

I have evaluated 3 ADT's, all graphs came out similar with different values, in order to make this section more fluent I will only bring one ADT for each section. All plot could be found in the projects repository[15] as well with the generating script.

[13] more precisely, with more manipulations to the ADT.
[14] See run.log file for tests
[15] \Evaluation\plots

Wrapper overhead:

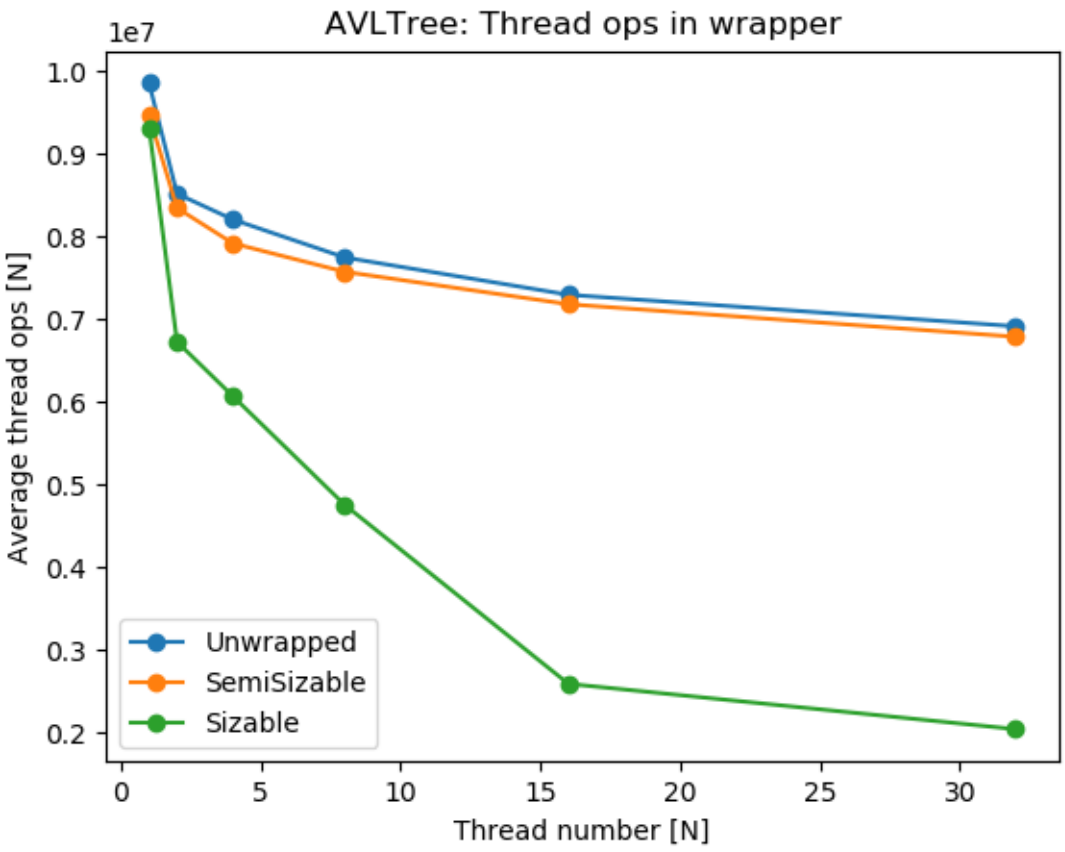

Figure 1: Wrapper Overhead

This plot shows the throughput of a thread in respect to the number of threads running[16]. Under the current load all we do is test how much the wrapper slows the ADT down.
As predicted it seems like the SemiSizable wrapper has a minor effect, but the Sizable deteriorates the throughput noticeably as the thread number grows.

[16] key-range=10^6, load = 50% inserts 50% removes

Size Operation:

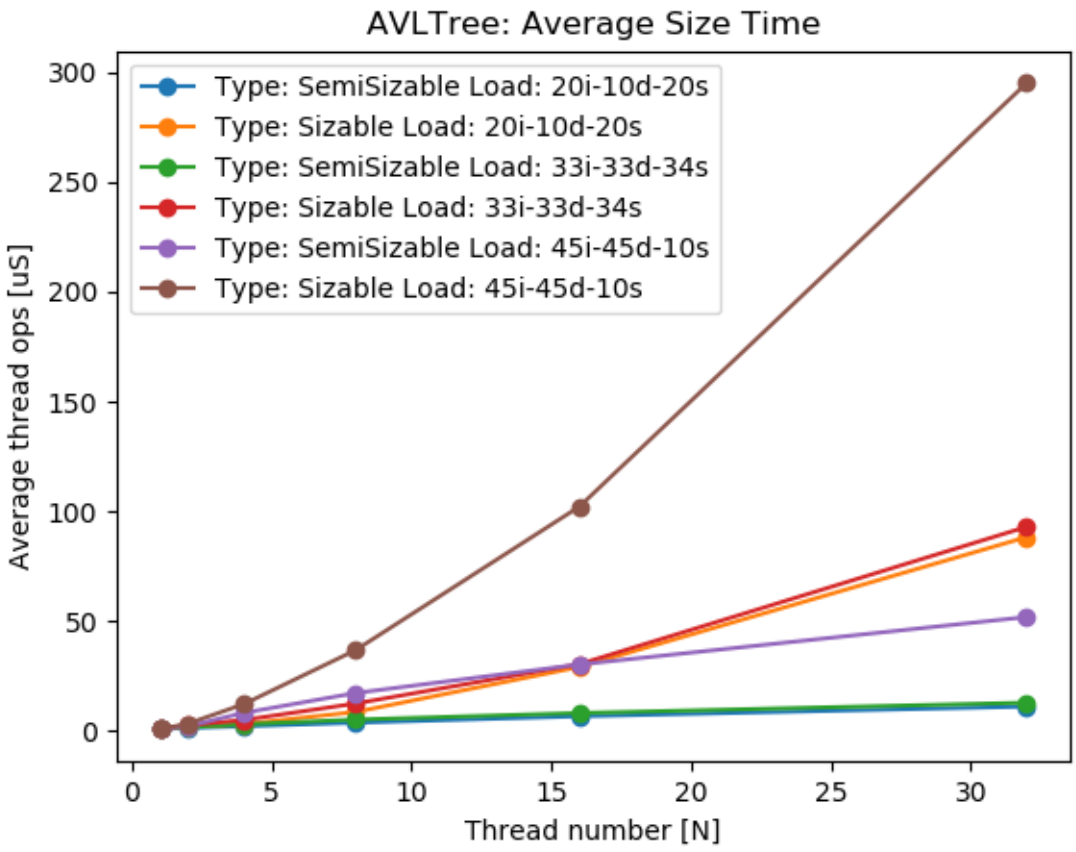

Figure 2: getSize Duration

This plot shows the duration of the get Size time with regards to the load on the ADT and the wrapper type. As predicted the SizeDuration increases the more threads there are and the bigger the load is. Slightly surprisingly, the SemiSizable is effected only in high loads[17]. As well as predicted once the load gets bigger, more mutations happen and the Sizable getSize operation grows at a high rate and starves.

[17] This growth function should be tested in the future

Key Range

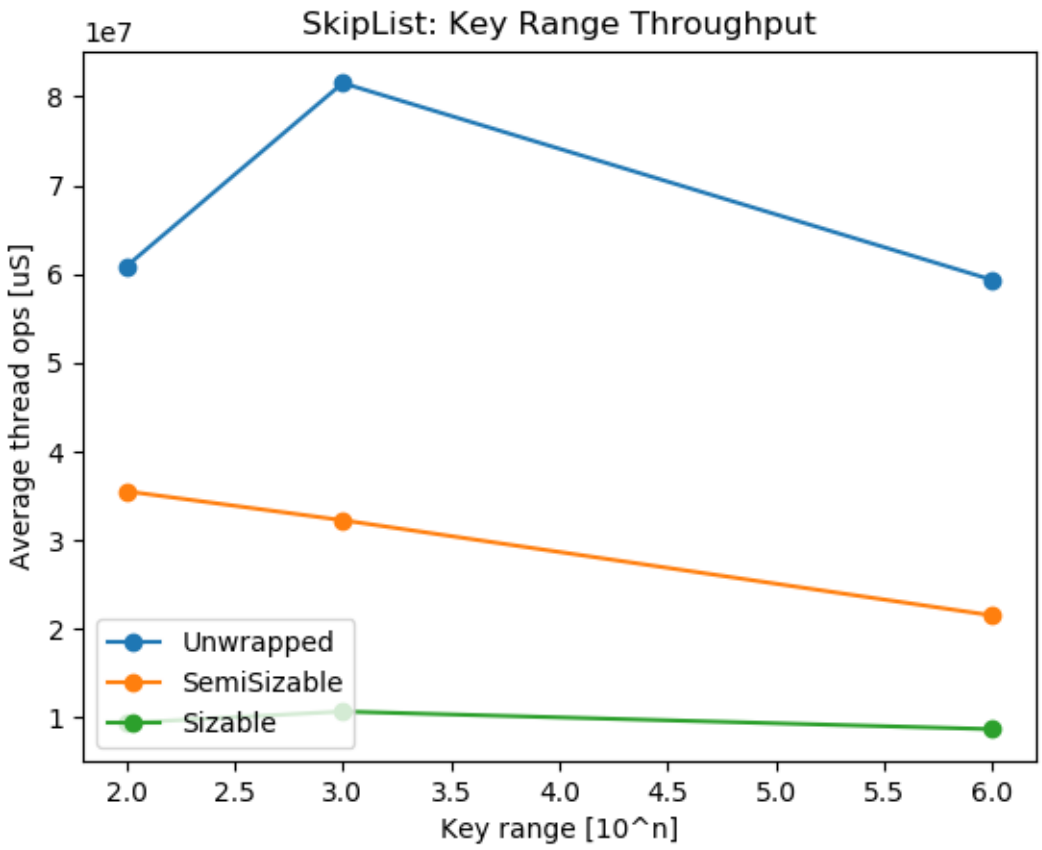

Figure 3: Key Range effect on the throuput

In this plot we see the effect on the key range for a run with a load of 33%insert 33% deletions 34%size on 16 threads. It's hard to make to much conclusions from a 3 dot line, but as predicted they seem stable. an unexpected burst is seen in the regular ADT at a key range of 10^4, which appears in all graphs and I can not explain.

7 Conclusion and expansions

7.1 Conclusion

It is possible to get a Non blocking size call, but it becomes prune to starvation and is non scale-able even without even calling the size function once. On the other hand the SemiSizable extension an ADT might prove useful in order to get a good approximation of the size in a non blocking manner. We saw that the greatest impact on the returning time occurred the bigger the load is.

7.2 further expansions

- Make the sizable extensions actually implement the Java Collection and Java Map interfaces to make them easy to use.
- Test the growth function of the SemiSizable function under even bigger loads to see if it blows as well.

8 References

1. inspired by "KiWi - OCC Key-Value Map" by Assaf Yifrach, Niv Gabso
2. Git repository: `https://github.com/FruitOwl/LinearizableSize`
3. Git test repository: `https://github.com/FruitOwl/LinearizableSizeTest`