Linearizable size - project proposal

David Agassi
308369677
davidio.agassi@gmail.com

February 18, 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

[1]Based on "KiWi - OCC Key-Value Map" by Assaf Yifrach, Niv Gabso

[2]in practice this could be optimized.

[3]CuncurentBST - (copy from internet?), Java ConcurrentHashMap, Java ConcurrentLinkedQueue

adders value across threads. This method is a simple sum of differences across threads and isn't linearizable. I will create 2 counters based on the LongAdder, one not linearizable, and the other is[4]linearizable. The second counter should be more prone to starvation.

Part I

Proof

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

Part II

Java implementation

[4]using AtomicBollean