Interrupting Snapshots and the Java™ Size() Method

Yehuda Afek[1], Nir Shavit[1,2], and Moran Tzafrir[1]

[1] Tel-Aviv University, Tel-Aviv, Israel
[2] Sun Microsystems, Burlington, MA

Abstract. The Java™ developers kit requires a `size()` operation for all objects. Unfortunately, the best known solution, available in the Java concurrency package, has a blocking concurrent implementation that does not scale. This paper presents a highly scalable wait-free implementation of a concurrent `size()` operation based on a new lock-free *interrupting snapshots* algorithm for the classical atomic snapshot problem. This is perhaps the first example of the potential benefit from using atomic snapshots in real industrial code (the concurrency package is currently deployed on over 10 million desktops).

The key idea behind the new algorithm is to allow snapshot scans to interrupt each other until they agree on a shared linearization point with respect to updates, rather than trying, as was done in the past, to have them coordinate the collecting of a shared global view. As we show, the new algorithm scales well, significantly outperforming existing implementations.

1 Introduction

The Java developers kit requires a size() operation, counting the number of elements in the data structure, to be made available for all objects. Accordingly, the Java concurrency package (currently deployed on over 10 million desktops), includes a concurrent implementation of the size() operation. Unfortunately, this global-counter based implementation has two problems: (a) it is *blocking*, so non-blocking structures such as the ConcurrentSkipListMap, unfortunately have a blocking size() operation during which other operations on the structure are delayed, and (b) if all modifying operations update the size, the implementation simply does not scale. To allow scalability one must update infrequently and accordingly relax the specification so that size() is only *an approximation* of the actual data structure size.

This paper overcomes the above limitations, presenting a wait-free, linearizable, and highly scalable implementation of the Java size() operation that can be added seamlessly to any concurrent data structure. Our solution is based on a new *interrupting snapshots* algorithm for the classical atomic snapshots problem [1, 2].

1.1 Atomic Snapshots

To implement a scalable size counter, the frequent operations that update size must have a very low overhead, and the burden of the implementation should be placed on the relatively infrequent calls to size(). One solution that immediately pops to mind is to use localized individual counters, one per each of the n threads, to track changes to the structure, and use an atomic snapshot operation [1, 2] to collect an instantaneous coherent view of all the individual counters. An atomic snapshot object thus has two operations, an update() that writes to its given location and a scan() that collects a view of all locations.

The problem is that the classical read-write register-based snapshot algorithms such as [1–3] have an $O(n)$ and higher update complexity even when uncontended (see [4] for a survey). The only practical algorithm with an $O(1)$ common case update complexity is the coordinated collect algorithm of Riany et. al [4]. However, it requires $O(n)$ costly read-modify-write operations (such as compare-and-swap (CAS)) to shared locations to coordinate the collection of a shared snapshot view. To implement a scalable size() operation, we must therefore devise a new type of snapshot algorithm, one that uses a small number of CAS operations to collect the snapshot view.

1.2 Interrupting Snapshots in a Nutshell

Our quest for an algorithm begins with an implementation of a new lock-free snapshot algorithm. As a basis, we start out with the single scanner algorithm of Riany et. al [4] which is in turn based on the single scanner algorithm of [5]. This single scanner algorithm uses an array of $2n$ entries, a recent and a previous entry per thread, and a *scan sequence number* incremented by the single scanner at the start of every scan (the structure is depicted in part (a) of Figure 1). The recent and previous values are tagged with a scan sequence number. Updaters start by reading the scan counter and then update the recent field as long as the field's sequence number is the same as the counter value they read. Upon detecting a newer scan counter value, the updater performs a "copy on write", it copies the recent field to the previous field before writing the new value into recent. To collect a snapshot, the single scanner, after incrementing the scan counter, passes through the array, and for every location collects the recent value if the sequence number is less than the scan's sequence number, and collects the previous value if the recent sequence number is equal to the scan's number (because there is a single scanner it cannot be greater). The collection of values forms a snapshot because the counter increment is a linearization point for the scan: any update that starts after this point will not be collected as we will always collect the latest previous value.

To make this algorithm support multiple scanners, one cannot simply have all scanners increment a shared sequence counter using a CAS operation because different scanners can collect old and new values forming inconsistent views. Riany et. al [4] solved the problem by having threads coordinate the collection

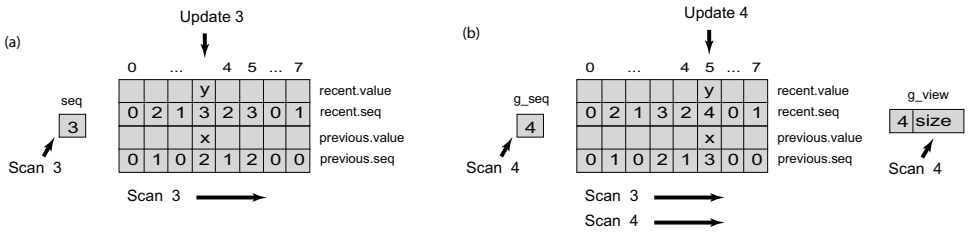

Fig. 1. Sample executions of the algorithm. Part (a) describes an execution of the single scanner algorithm. A scan increments the global sequence number g_seq to 3, then while it collects its view an updater with value y reads this new global sequence number. It must therefor copy its old value x from recent to previous and update its recent field to y with sequence 3. When the scan reads this recent field it will see that its seq is 3, so it started after the scan, and will therefore use the previous value x. In part (b) we see a wait-free algorithm with the seq number now shared among scans. A scan of seq 3 is concurrent with a scan with seq 4, and an updater thread reads 4, updates its recent field to a new value y with sequence 4. When the scan with seq 3 reads this recent field it will see that its seq 4 is greater than 3 and will be forced to retry the scan. Even if it misses this update and completes collecting values, it will be forced to retry when it finally checks that the global sequence number has remained 3. It will perform the retry with a value of 4, unless when it checks the g_view field it finds that the scan with 4 has updated the view, in which case it adopts and returns this new view.

of a single shared snapshot view, a process which unfortunately entails using n CAS operations even in the uncontended case.

The key to our new *interrupting snapshots* algorithm is to avoid collecting a shared view. Instead, we allow scanners to interrupt each other, forcing a retry of the scan. We add a new shared g_view field that contains the sequence number of the last successful scanner. Each scanner begins by incrementing the shared scan counter g_seq using a CAS (this is done per scan, not per update). It next collects its own snapshot view, checking to see that there are no updated fields with a sequence number greater than its own. If there are such sequence numbers, then clearly a later scan has started concurrently (See the example in part (a) of Figure 1). If a scan does complete the collection of values, it checks to see that the shared g_seq counter still contains its own sequence number, and if this number has changed, it again was interrupted. Our idea then is to have the interrupted scan retry collecting its snapshot again. However, this time it *does not* increment the shared g_seq counter. Instead, it uses the sequence number of the scan that interrupted it. This way, if no scan succeeds, then eventually all scans agree on the same sequence number and no longer interrupt each other. The final step in our algorithm is that each scan that successfully collected a view tries to be the next successful scan by increasing the g_view field to its sequence number. If it detects that the g_view field already has a higher value, then it has failed and restarts from the beginning, incrementing the g_seq etc.

How does this help us in collecting a lock-free scan? Well, as we prove, every time a scanner is interrupted it must be because it has seen a sequence number of a later scan, and it adopts this scan's number. After at most $n-1$ threads have all started a scan, if none of them succeed they will all eventually agree on the highest sequence number. They will all collect a view and one of them will succeed in CASing g_view to the new sequence number and complete the scan. Thus, the only way in which new sequence numbers can be continuously generated and prevent a scanner from completing is if at least one scan successfully completed and increased the g_view field, implying that scans are lock-free.

We can now extend the lock-free interrupting snapshot to implement a wait-free size() implementation in a straightforward fashion. As in Figure 1, we add to g_view a field that contains the snapshot view (in our case an integer representing the collected size) of the last successfully completed scan, and as before, it also contains its sequence number. Successful scans CAS both their sequence number and the view they collected into this location. If a thread is interrupted while performing its lock-free interrupting scan, it checks to see if the g_view field has a sequence greater than its own, that is, it contains a size value collected by some interrupting scan that has started after it did. If it finds such one then it can safely return that size. Otherwise, it continues in the lock-free collection attempt. The algorithm is wait-free because either a scanner is eventually not interrupted and completes a scan, or it must be that there are other successful scans continuously interrupting it, implying that it can return the size collected by one of them.

Our new wait-free size() algorithm thus has a "take the view of some thread that interrupted you" flavor of former algorithms such as [1], but unlike these algorithms, the interruptions are between scans, not between scans and updates, which is the key to the new algorithm's efficiency.

1.3 Benefits of the new algorithm

As we prove, unlike existing solutions, our new size algorithm is wait-free, linearizable, and has an update() operation that in the uncontended case requires only a couple of loads and one store and a scan() operation that requires $O(n)$ loads and two CAS operations. We show that it can be added to any data structure to provide a linearizable size() implementation.

We also note that the wait-free size() implementation we presented can be extended to allow a general scalable wait-free snapshot implementation with a view larger than a single word (in the case of size() we use the fact that the view is contained in a single word to transform the lock-free snapshot scan() to a wait-free size()). [3]

In the performance section we compare our new wait-free size() implementation to the one offered by the Java concurrency package as well as one imple-

[3] The most straightforward practical way to do this is to have a scanner allocate a structure in which its view is collected and use the CASable shared location to store a reference to this structure.

mented using the practical snapshot algorithm of Riany et. al. [4]. Our benchmarking was performed on two state-of-the-art multiprocessor machines: an Azul Vega2 (7200 series) distributed shared memory with up to 768 processors and a Sun Maramba 128-way multicore machine. As can be seen, our algorithm shows impressive scalability while the other algorithms simply do not scale.

In summary, the strong progress and coherence properties combined with high scalability of the new algorithm lead us to suggest that it is a good candidate for replacing the current size operation in the Java concurrency package. It is perhaps satisfying that a theoretically motivated data structure, the atomic snapshot object, introduced in the late 80s [1, 2], can finally find real-world applicability.

2 The Algorithm

Our computation model and specification of atomic snapshots follows [1, 4], with the small exception of replacing load-linked/store conditional operations (LLSC) by compare-and-swap (CAS) operations.

Following [1] an atomic snapshot object provides two operations, update() and scan() with the usual semantics.

Section 1.2 provided a high level view of our new algorithm. Here we provide a more detailed walk through its pseudo-code. As we did earlier, we start by describing our lock-free *interrupting snapshots* algorithm, and then modify it into a wait-free size operation. The main difference between size() and an atomic snapshot is that to store a full snapshot we need an array of values, while for size, one integer word is enough.

The pseudo-code of the lock-free snapshot algorithm is provided in Figure 2. Each updater maintains two values, recent and previous, each with an associated sequence number which is the value the updater observed in the global counter just before updating this value. The two values together are stored in a **struct** called data() (Line 4). The Data of thread i is in Thread[i]. To keep track on which value belongs to which snapshot we maintain a global counter, called g_seq, which each scanner atomically fetches and increments when it starts the scan. Each updater reads before writing a new value. Using these values a scanner can tell which values are associated with its scan. Each scan is identified by the sequence number it obtained in the atomic fetch and increment at the beginning of the scan.

The scanner obtains a snapshot by reading from the updater's locations in Lines 25 to 37. The point in time where it increments g_seq is a "line in the sand". All updates starting after this point are ignored and those before it can be part of the snapshot.

Consider a read from the j-th location. If thread j's recent sequence number is smaller than the scanner's sequence number then clearly the value j had in its recent field (associated with the read sequence number) was its valid value at the time that the scanner performed its increment of g_seq and the scanner then takes this value (Line 29). If j's recent number was equal the scanner's

```
1  data structures
2  ------------------------------
3   struct     view  { view[1], view[2], ... view[n] } //  Holds a snapshot
4   struct     Data  { int     _value, _seq;  };      // a pair <value; seq_number>
5   struct     Thread {
6              Data _recent;
7              Data _prev; };
8   int        g_view_seq  ;
9   int        g_seq;  //global snapshot sequence(counter)
10  Thread* g_mem; //array of Thread struct, with an entry for each thread
11  g_view     g_view; //holds the latest global view with its associated seq#
12 ------------------------------
13  method wait_free_update(i, new_value)
14 ------------------------------
15  if (g_seq != g_mem[i]._recent._seq) { g_mem[i]._prev := g_mem[i]._recent; }
16  g_mem[i]._recent := {new_value, g_seq};
17 ------------------------------
18  method lock_free_scan()
19 ------------------------------
20   start_view_seq  := g_view_seq  ;
21  scan_seq :=AtomicIncAndFetch(g_seq);
22  do
23      n_scan := view[1,...n]  ;
24      scan_ok := true;
25      for (i =:0; i < N; ++i) {
26          prev := g_mem[i]._prev;
27          recent := g_mem[i]._recent;
28          if (recent._seq < scan_seq) {
29              n_scan[i] := recent._value;
30          } else if (recent._seq == scan_seq) [4] {
31              n_scan[i] := prev._value;
32          } else {scan_ok := false;
33                   start_view_seq  := g_view_seq;
34                  scan_seq := g_mem[i]._recent._seq;
35                  if (scan_seq == start_view_seq){CAS(g_seq, scan_seq, scan_seq+1); scan_seq=g_seq;}
36                  break;
37      } }                        // end for
38      if (scan_ok) {
39          if (CAS (g_view_seq, start_view_seq , scan_seq)) {
40              return n_scan;
41          } start_view_seq  := g_view_seq;
42       } endif
43  while (true);
```

Fig. 2. The lock-free *interrupting snapshots* algorithm.

sequence number, then the scanner takes j's previous value into the snapshot

view. If however, j's recent sequence number is greater than the scan's sequence number then the scanner has been interrupted by a later thread. In this case the scanner starts a new attempt to collect a snapshot, this time adopting the higher number of the interrupting scanner (Line 34). Finally, to keep linerization the a scanner must check that no scan operation completed during its interval. And this is what the CAS is for in Line 39.

Clearly if no scanner succeeds and the scanners run to infinity then consider the highest sequence number scanner, this scanner must eventually succeed or a new scanner joins in. But the number of scanner is bounded so the algorithm is lock-free.

A scanner that succeeds cannot just return the scan it obtained, because it may not be linearized with other successful and concurrent scans. To this end we add a global view location g_view_seq storing the sequence number of the most recent scan that successfully collected a view. In effect we guarantee that at any point of time there is (in retrospect) only one scan that is going to succeed. A scan will complete and return if it can successfully CAS its sequence number into the g_view_seq (Line 39) replacing the global view sequence number it has seen just before starting its most recent scan attempt. This is also the motivation of the CAS, to ensure the linearizability by the invariant that at any point of time there is in effect only one uninterrupted scan that is going to succeed.

The pseudo-code of the wait-free size() described in *Figure* 3 is a modification of the lock-free code presented above. The key to turning the above lock-free snapshot algorithm into a wait-free size() algorithm, is to add a global view (in our case consisting of a size integer fitting in a single word) that a scanner may check if it is interrupted by another. Thus, either a scanner succeeds in the lock-free scan attempt, or it is interrupted by another scan with a higher sequence number. Since such repeated failures can only happen if some thread continuously succeeds, then if each interrupted scanner checks the g_view, it will eventually observe a value with a larger sequence number than its own in g_view. It can safely adopt the size stored in this view, because the execution interval of the scan of the interrupting thread started after its own.

Due to space limitations the proofs of the algorithms are omitted from this abstract.

3 Performance Evaluation

The JDK's *ConcurrentHashMap* size() makes two attempts at a successful "double collects" in the style of [1] and if it does not succeed it locks all segments and counts the size. To use our *wait-free* size() solution, we simply had to add the *wait-free-size* object to the JDK's ConcurrentHashMap, and initialize its number of threads to the number of threads in the system (i.e., the concurrency level in JDK). Then we replace the JDK's size() method with a call to $wait\_free\_size()$. The *add* & *remove* methods of *segment* i, just call $update(i, new\_value)$, and no other changed were required (each thread maintains locally the total number of keys it is responsible for, which the new_value).

```
1  data structures
2  ---------------------------------
3   struct     Data { int _value, _seq ; };// a pair <value; seq_number>
4   struct     Thread {
5              Data _recent;
6              Data _prev; };
7   int        g_seq; //global snapshot sequence(counter)
8   Thread*    g_mem; //array of Thread struct, with an entry for each thread
9   Data       g_view; //holds the latest size result with its associated seq#
10 ---------------------------------
11  method wait_free_update(i, new) // Threads count locally and update with the new value.
12 ---------------------------------
13  if(g_seq != g_mem[i]._recent._seq) { g_mem[i]._prev := g_mem[i]._recent;}
14  g_mem[i]._recent := { new, g_seq};
15 ---------------------------------
16  method wait_free_size ()
17 ---------------------------------
18   start_view := g_view;
19  scan_seq :=AtomicIncAndFetch(g_seq);
20   first_seq := scan_seq;
21  do
22      size := 0 ;
23     scan_ok := true;
24     for (i =:0; i < N; ++i) {
25        prev := g_mem[i]._prev ;
26        recent := g_mem[i]._recent;
27        if (recent._seq < scan_seq) {
28             size += recent._value;
29        } else if (recent._seq == scan_seq) [5] {
30             size += prev._value;
31        } else {scan_ok := false;
32                  start_view := g_view;
33                  scan_seq := g_mem[i]._recent._seq;
34                  break;
35        } }                    // end if and end for
36     if ( first_seq ≤ (g_view._seq)) {
37        return (g_view._view); }
38     if (scan_ok) {
39        if (CAS (g_view, start_view, [scan_seq, size])) {
40             return size;
41        } start_view := g_view;
42      } endif
43  while (true);
```

Fig. 3. The wait-free size() algorithm.

3.1 Benchmarks Layout

Two comparisons are carried out in this section. First we compare the JDK's *ConcurrentHashMap* size(), to the new *wait-free* size(). Second, we compare

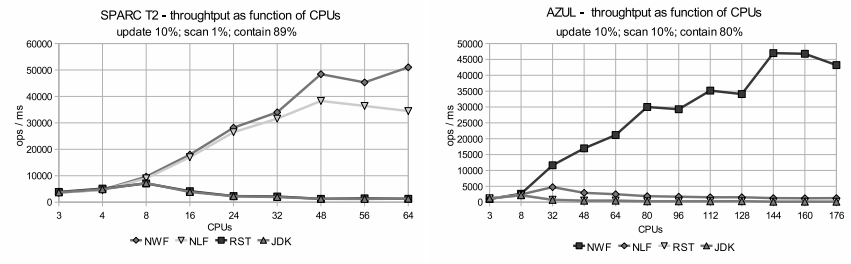

Fig. 4. *Scheme I*: how the *number* of threads affects the performances

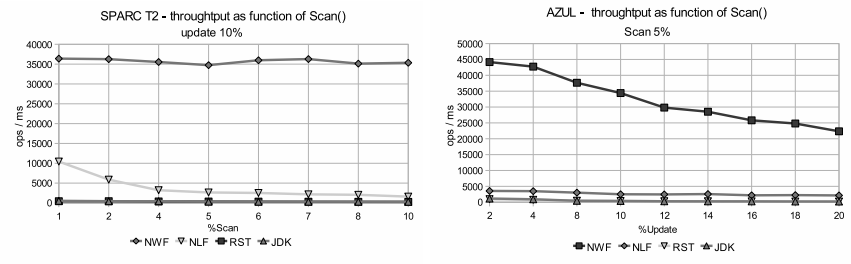

Fig. 5. *Scheme I*: how the the scan() & update() operations affects the performance

our *lock-free-snapshot* algorithm, to the previous *lock-free* snapshot algorithm of Riany, Shavit, and Touitou [4].

Evaluations are performed using micro-benchmarks similar to those used by [4]. The update() operation is equivalent to add() (add a key) and remove() (remove a key). The scan() operation is equivalent to the size() (get the number of keys) operation. And the contain() (check if a key exists), does not effects the *snapshot*.

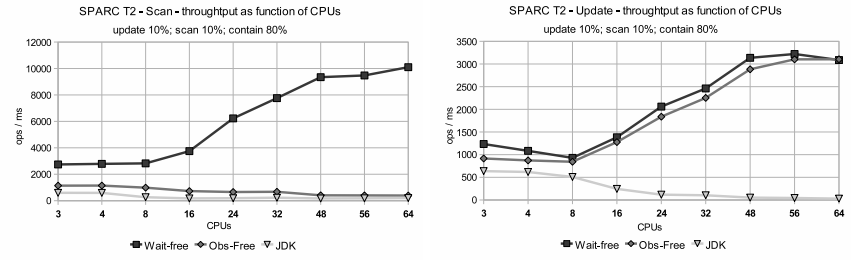

Fig. 6. *Scheme II*: Each thread performs one type of an operation

Two schemes are used in the evaluation, in benchmark-scheme *Scheme I*, we assign to each *thread* a predefined mixture of operations, the percent of update(), scan(), and contain() is set and is equal for all threads (see Figure 6). We consider this scheme a bit more realistic, and not restricting, from the user point-of-view.

In benchmark-scheme *Scheme II*, each *thread* performs only one type of operation, e.g. update(), or scan(), or contain(). And then we measure the *throughput* of each operation (see Figure 6).

We implemented the size() using four different algorithms. The first is *JDK*, its the original JDK's ConcurrentHashMap with the blocking size(). The second is *NWF* it is our new implementation of size() using the *wait-free-size*. Third implementation, *NLF*, is our implementation of size() based on the *lock-free* snapshot. And the fourth implementation, *RST*, is the *coordinated collect* algorithm from [4].

The benchmarks were done on two architectures to make sure the algorithm is not architecture specific. The first architecture is *SPARC II* UltraSPARC T2 Plus processor, it has eight-core 1.2_{GHz}, with two processors per system, maximum 128 threads, and up to 256_{GB} of memory. The second architecture is *AZUL* Vega2 Java-machine, the 7200 series contains up to 768 processing cores on 16 processor chips with 768_{GB} of memory, each processor chip has 48 core.

3.2 *Benchmark Scheme I:*

In figure *Figure* 4 we test how the *number* of threads affects the performance of the scan() & update()operations. We measure the throughput, the number of operation per millisecond. Our *NWF* implementation almost not affect by the number of threads, while the *JDK* & *RST* do not scale at all, this is because *JDK* beehives like a *global-lock*, and the complexity of *RST* give it the same behavior.

In figure *Figure* 5 we test how the *percent* of the scan() operations affects the performance of the scan() & update()operations. Increasing Our *NWF* implementation almost not affect by the number of threads, while the *JDK* & *RST* do not scale at all, this is because *JDK* beehives like a *global-lock*, and the complexity of *RST* give it the same behavior.

3.3 *Benchmark Scheme II:*

Figure 6 is perhaps the more interesting benchmark as it shows the benefit of adding the mechanism to "adopt the view of interrupting scans" that we added to our size() implementation to make it wait-free over the lock-free scan() on which it is based.

4 Acknowledgments

We thank Dave Dice and Doug Lea for their help during the writing of this paper.

References

1. Afek, Y., Dolev, D., Attiya, H., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. In: PODC. (1990) 1–13
2. Anderson, J.H.: Multi-writer composite registers. Distrib. Comput. **7**(4) (1994) 175–195
3. Attiya, H., Rachman, O.: Atomic snapshots in o (n log n) operations. SIAM J. Comput. **27**(2) (1998) 319–340
4. Riany, Y., Shavit, N., Touitou, D.: Towards a practical snapshot algorithm. In: ISTCS. (1995) 121–129
5. Kirousis, L.M., Spirakis, P.G., Tsigas, P.: Simple atomic snapshots: A linear complexity solution with unbounded time-stamps. In: ICCI '91: Proceedings of the International Conference on Computing and Information, London, UK, Springer-Verlag (1991) 582–587