Lock-Free Data-Structure Iterators*

Erez Petrank and Shahar Timnat

Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa, Israel
{erez,stimnat}@cs.technion.ac.il

Abstract. Concurrent data structures are often used with large concurrent software. An *iterator* that traverses the data structure items is a highly desirable interface that often exists for sequential data structures but is missing from (almost all) concurrent data-structure implementations. In this paper we introduce a technique for adding a linearizable wait-free iterator to a wait-free or a lock-free data structure that implements a set. We use this technique to implement an iterator for the wait-free and lock-free linked-lists and for the lock-free skip-list.

Keywords: concurrent data structures, lock-freedom, wait-freedom, linked-list, skiplist, iterator, snapshot.

1 Introduction

The rapid deployment of highly parallel machines resulted in the design and implementation of a variety of lock-free and wait-free linearizable data structures in the last fifteen years. However, almost none of these designs support operations that require global information on the data structure, such as counting the number of elements in the structure or iterating over its nodes. In general, operations such as these will be trivially enabled if snapshot operations are supported because snapshot operations enable a thread to obtain an atomic view of the structure. But creating a "consistent" or linearizable snapshot without blocking simultaneous updates to the data structure is a difficult task. The main focus of this study is to obtain such a view in a wait-free manner.

A common interface in many lock-free and wait-free data structures consists of the INSERT, DELETE and CONTAINS operations. An INSERT operation inserts an integer key (possibly associated with a value) into the data structure, if it is not already present (otherwise it just returns false). A DELETE operation removes a key from the structure, or fails (returning false) if there is no such key, and a CONTAINS operation returns true (and possibly a value associated with this key) if the key is in the list, and false otherwise. Examples of data structures implementing this interface are the lock-free linked-lists [9,8], the wait-free linked-lists [15], the lock-free skiplist [10], and search trees [6,4]. None of these structures implements an iterator.

In this work we present a design which allows the construction of wait-free, highly efficient iterators for concurrent data structures that implement sets. We use this design to implement iterators for the linked-list and skiplist. The iterator is implemented by first obtaining a consistent *snapshot* of the data structure, i.e., an atomic view of all the

* This work was supported by the Israeli Science Foundation grant No. 283/10.

Y. Afek (Ed.): DISC 2013, LNCS 8205, pp. 224–238, 2013.
© Springer-Verlag Berlin Heidelberg 2013

nodes currently in it. Given this snapshot, it is easy to provide an iterator, or to count the number of nodes in the structure.

A well-known related problem is the simpler *atomic snapshot object* of shared memory [1], which has been extensively studied in the literature. An atomic snapshot object supports only two types of operations: UPDATE and SCAN. An UPDATE writes a new value to a register in the shared memory, and a SCAN returns an atomic view of all the registers.

Unfortunately, existing snapshot algorithms cannot support a (practical) data structure iterator. Three problems hinder such use. First, atomic snapshot objects are designed for pre-allocated and well-defined memory registers. Therefore, they are not applicable to concurrent data structures that tend to grow and shrink when nodes are added or removed. Second, the UPDATE operation in the classic snapshot object algorithms [1,3] requires O(n) steps (n is the number of threads), which is too high an overhead to impose on all operations that modify the data structure. Finally, many atomic snapshot objects do not support an efficient READ operation of the shared memory. This lack of support allows linearization arguments that would fail in the presence of a read. But it is hard to imagine a practical data structure that does not employ a read operation, and instead relies on obtaining a full snapshot just to read a single field in the structure.

The first problem is the least bothersome, because one could imagine borrowing ideas from snapshot objects, generalizing them, and building a snapshot algorithm for a memory space that grows and shrinks. But the other two problems are harder to eliminate. The classic algorithms for an atomic snapshot can be easily extended to support a READ operation, but they require O(n) steps for each UPDATE operation, which is too high. Later snapshot algorithms support UPDATE in O(1) steps. Examples are the coordinated collect algorithm of Riany et al. [14], later improved to the interrupting snapshots algorithm [2], and the time optimal snapshot algorithms of Fatourou and Kallimanis [7]. However, these algorithms do not support a READ operation. This lack of support seems inherent as the algorithms employ unusual linearization properties, which sometimes allow the linearization point of an UPDATE to occur before the new value has actually been written to *any* register in the memory. Thus, it is not clear how to add a READ operation that does not require a substantial overhead.

Another wait-free algorithm that supports UPDATE operations in O(1) is the algorithm of Jayanti [11]. Jayanti's algorithm does not support a read operation, and it is not trivial to add an efficient read to it, but our work builds on ideas from this algorithm. An UPDATE operation of Jayanti's algorithm first updates the memory as usual, and then checks whether a SCAN is currently being taken. If so, the update operation registers the update in a designated memory register. In this work we extend this basic idea to provide a snapshot that supports an efficient read as well as the INSERT, DELETE, and CONTAINS operations, which are more complex than the simple UPDATE operation. This facilitates the desirable iterator operation for the data structure. The simplest algorithm of Jayanti, from which we start, is described in Section 2.

Although most lock-free data structures do not provide iterators, one notable exception is the recent CTrie of Prokopec et al. [13]. This lock-free CTrie efficiently implements the creation of a snapshot, but the performance of updates deteriorates when concurrent snapshots are being taken, because each updated node must be copied,

together with the path from the root to it. Another recent work presenting a concurrent data structure supporting snapshot operations is the practical concurrent binary search tree of Bronson et al. [5]. But their work uses locks, and does not provide a progress guarantee.

In this paper, we present a wait-free snapshot mechanism that implements an $O(1)$ update and read operations. We have implemented a linked-list and skiplist that employ the snapshot and iterator and measured the performance overheads. In our implementation we made an effort to make updates as fast as possible, even if iterations take a bit more time. The rational for this design is that iterations are a lot less frequent than updates in typical data structures use. It turns out that the iterator imposes an overhead of roughly 15% on the INSERT, DELETE, and CONTAINS operations when iterators are active concurrently, and roughly 5% otherwise. When compared to the ad-hoc CTrie iterator of [13], our (general) iterator demonstrates lower overhead on modifications and read operations, whereas the iteration of the data structure is faster with the ad-hoc CTrie iterator.

2 Jayanti's Single Scanner Snapshot

Let us now review Jayanti's snapshot algorithm [11] whose basic idea serves the (more complicated) construction in this paper. This basic algorithm is limited in the sense that each thread has an atomic read/write register associated with it (this variant is sometimes referred to as a single-writer snapshot, in contrast to a snapshot object that allows any thread to write to any of the shared registers). Also, it is a single scanner algorithm, meaning that it assumes only one single scanner acting at any point in time, possibly in parallel to many updaters. In [11], Jayanti extends this basic algorithm into more evolved versions of snapshot objects that support multiple writers and scanners. But it does not deal with the issue of a READ operation, which imposes the greatest difficulty for us. In this section we review the basic algorithm, and later present a snapshot algorithm that implements a read operation (as well as eliminating the single-writer and single-scanner limitations), and combines it with the INSERT, DELETE, and CONTAINS operations.

Jayanti's snapshot object supports two operations: UPDATE and SCAN. An UPDATE operation modifies the value of the specific register associated with the updater, and a SCAN operation returns an atomic view (snapshot) of all the registers. Jayanti uses two arrays of read/write registers, A[n], B[n], initialized to null, and an additional read/write binary field, which we denote *ongoingScan*. This field is initialized to false. The first array may be intuitively considered the main array with all the registers. The second array is used by threads that write during a scan to report the new values they wrote. A third array of n registers, C[n], is never read in the algorithm; it is used to store the snapshot the scanner collects. The algorithm is depicted in figure 1. When thread number k executes an UPDATE, it acts as follows. First, it writes the new value to A[k]. Second, it reads the ongoingScan boolean. If it is set to false, then the thread simply exits. If it is set to true, then the threads *reports* the new value by also writing it to B[k], and then it exits.

When the scanner wants to collect a snapshot, it first sets the ongoingScan binary field to true. Then, in the second step, it sets the value of each register in the array B to

```
A[n], B[n], C[n]:  arrays of read/write
registers initiated to Null
ongoingScan: a binary read/write register
initiated to 0.

Update(tid, newValue)
    1.  A[tid] = newValue
    2.  If (ongoingScan==1)
    3.      B[tid]=newValue
```

```
Scan()
    1.  ongoingScan = 1
    2.  For i in 1..n
    3.      B[i] = NULL
    4.  For i in 1..n
    5.      C[i] = A[i]
    6.  ongoingScan = 0
    7.  For i in 1..n
    8.      If (B[i] != NULL)
    9.          C[i] = B[i]
   10.  Array C now holds the Snapshot
```

Fig. 1. Jayanti's single scanner snapshot algorithm

null (in order to avoid leftovers from previous snapshots). Third, it reads the A registers one by one and copies them into the C array. Fourth, it sets the ongoingScan to false. This (fourth) step is the linearization point for the SCAN. At this point array C might not hold a proper snapshot yet, since the scanner might have missed some updates that happened concurrently with the reading of the A registers. To rectify this, the scanner uses the reports in array B; thus in the final step, it reads the B registers one by one, and copies any non-null value into C. After that, C holds a proper snapshot.

The linearizability correctness argument is relatively simple [11]. The main point is that any UPDATE which completes before the linearization point of the SCAN (line 6) is reflected in the snapshot (either it was read in lines 4-5 or will be read in lines 7-9), while any UPDATE that begins after the linearization point of the SCAN is not reflected in the snapshot. The remaining updates are concurrent with each other and with the scan since they were all active during the linearization point of the SCAN (line 6). This gives full flexibility to reorder them to comply with the semantics of the snapshot object ADT.

3 From Single Scanner Snapshot to Multiple Iterators

Our goal is to add an iterator to existing lock-free or wait-free data structures. We are interested in data structures that support three standard operations: INSERT, DELETE, and CONTAINS. Similarly to the scanner object, threads executing the INSERT, the DELETE, or the CONTAINS operations cooperate with a potential scanner in the following way.

- Execute the operation as usual.
- Check whether there exists a parallel ongoing scan that has not yet been linearized.
- If the check is answered positively, report the operation.

Two major complications that do not arise with a single scanner snapshot algorithm arise here: the need to report operations of other threads, and the need to support multiple concurrent iterators.

3.1 Reporting the Operations of Other Threads

The first problem stems from dependency of operations. Suppose, for example, that two INSERT operations of the same value (not currently exist in the data structure) are

executed concurrently. One of these operations should succeed and the other should fail. This creates an implicit order between the two INSERTs. The successful INSERT must be linearized before the unsuccessful INSERT. In particular, we cannot let the second operation return before the linearization of the snapshot and still allow the first operation not to be visible in the snapshot. Therefore, we do not have the complete flexibility of linearizing operations according to the time they were reported, as in Section 2.

To solve this problem, we add a mechanism that allows threads, when necessary, to report operations executed by other threads. Namely, in this case, the failing INSERT operation will first report the previous successful INSERT of T_2, and only then exit. This will ensure that the required order dependence is satisfied by the order of reports. In general, threads need to report operations of other threads if: (1) the semantics of the ADT requires that the operation of the other thread be linearized before their own operation, and (2) there is a danger that the iterator will not reflect the operation of the other thread.

3.2 Supporting Multiple Iterators

In the basic snapshot algorithm described in Section 2, only a single simultaneous scanning is allowed. To construct a useful iterator, we need to support multiple simultaneous iterators. A similar extension was also presented in [11], but our extension is more complicated because the construction in [11] does not need to even support a read, whereas we support INSERT, DELETE, and CONTAINS.

In order to support multiple iterators, we can no longer use the same memory for all the snapshots. Instead, the data structure will hold a pointer to a special object denoted the *snap-collector*. The snap-collector object holds the analogue of both arrays B and C in the single scanner snapshot, meaning it will hold the "copied" data structure, and the reports required to "fix" it. The snap-collector will also hold a Boolean equivalent to `ongoingScan`, indicating whether the iteration has already been linearized.

4 The Iterator Algorithm

The pseudo-code for the iterator is depicted in Figure 2. This algorithm applies as is to the wait-free linked-list [15], the lock-free linked-list [9], and the lock-free skiplist [10].

When a thread wishes to execute an iteration over the data structure elements, it will first obtain a snapshot of the data structure. To optimize performance, we allow several concurrent threads executing an iteration to cooperate in constructing the same snapshot. For this purpose, these threads need to communicate with each other. Other threads, which might execute other concurrent operations, also need to communicate with the iterating threads and forward to them reports regarding operations which the iterating threads might have missed. This communication will be coordinated using a snap-collector object.

The snap-collector object is thus a crucial building block of the iterator algorithm. During the presentation of the iterator algorithm, we will gradually present the interface the snap-collector should support. The implementation of the snap-collector object that

supports the required interface is deferred to Section 5. All snap-collector operations are implemented in a wait-free manner so that it can work with wait-free and lock-free iterator algorithms.

To integrate an iterator, the data structure holds a pointer, denoted *PSC*, to a snap-collector object. The PSC is initialized during the initialization of the structure to point to a dummy snap-collector object. When a thread begins to take a (new) snapshot of the data structure, it allocates and initializes a new snap-collector object. Then, it attempts to change the PSC to point to this object using a compare-and-swap (CAS) operation.

4.1 The Reporting Mechanism

A thread executing INSERT, DELETE or CONTAINS operation might need to report its operation to maintain linearizability, if a snapshot is being concurrently taken. It firsts executes the operation as usual. Then it checks the snap-collector object, using the later's `IsActive` method, to see whether a concurrent snapshot is afoot. If so, and in case forwarding a report is needed, it will use the snap-collector `Report` method. The initial dummy snap-collector object should always return false when the IsActive method is invoked.

There are two types of report. An *insert-report* is used to report a node has been inserted into the data structure, and a *delete-report* used to report a removal. A report consists of a pointer to a node, and an indication which type of report it is. Using a pointer to a node, instead of a copy of it, is essential for correctness (and is also space efficient). It allows an iterating thread to tell the difference between a relevant delete-report to a node it observed, and a belated delete-report to a node with the same key which was removed long ago.

Reporting a Delete Operation. It would have been both simple and elegant to allow a thread to completely execute its operation, and only then make a report if necessary. Such is the case in all of Jayanti's snapshot algorithms presented in [11]. Unfortunately, in the case of a DELETE operation, such a complete separation between the "normal" operation and the submission of the report is impossible because of operation dependence. The following example illustrates this point.

Suppose a thread S starts taking a snapshot while a certain key x, is in the data structure. Now, another thread T_1 starts the operation DELETE(x) and a third thread T_2 concurrently starts the operation CONTAINS(x). Suppose T_1 completes the operation and removes x, but the scanner missed this development because it already traversed x, and suppose that now T_1 is stalled and does not get to reporting the deletion. Now T_2 sees that there is no x in the list, and is about to return false and complete the CONTAINS(x) operation. Note that the CONTAINS operation must linearize before it completes, whereas the snapshot has not yet linearized, so the snapshot must reflect the fact that x is not in the data structure anymore. Therefore, to make the algorithm linearizable, we must let T_2 first report the deletion of x (this is similarly to the scenario discussed in Section 3.1.). However, it cannot do so: to report that a node has been deleted, a pointer to that node is required, but such a pointer is no longer available, since x has been removed.

We solve this problem by exploiting the delete mechanism of the linked-list and skiplist (and other lock-free data structures as well). As first suggested by Harris in [9], a node is deleted in two steps. First, the node is marked. A marked node is physically in the data structure, and still enables traversing threads to use it in order to traverse the list, but it is considered *logically deleted*. Second, the node is physically removed from the list. The linearization of the DELETE operation is in the first step. We will exploit this mechanism by reporting the deletion between these two steps (lines 11-13 in Figure 2).

Any thread that is about to physically remove a marked node will first report a deletion of that node (given a snapshot is concurrently being taken). This way, the report is appropriately executed *after* the linearization of the DELETE operation. Yet, if a node is no longer physically in the list, it is guaranteed to have been reported as deleted (if necessary). Turning back to the previous scenario, if T_2 sees the marked node of x, it will be able to report it. If it doesn't, then it can safely return. The deletion of x has already been reported.

Reporting an Insert Operation. After inserting a node, the thread that inserted it will report it. To deal with operation dependence, a CONTAINS method that finds a node will report it as inserted before returning, to make sure it did not return prior to the linearization of the corresponding insertion. Furthermore, an INSERT operation that fails because there is already a node N with the same key in the list will also report the insertion of node N before returning, for similar reasons.

However, there is one additional potential problem: an unnecessary report might cause the iterator to see a node that has already been deleted. Consider the following scenario. Thread T_1 starts INSERT(3). It successfully inserts the node, but get stalled before checking whether it should report it (between lines 22 and 23). Now thread T_2 starts a DELETE(3) operation. It marks the node, checks to see whether there is an ongoing iteration, and since there isn't, continues without reporting and physically removes the node. Now thread S starts an ITERATION, announces it is scanning the structure, and starts scanning it. T_1 regains control, checks to see whether a report is necessary, and reports the insertion of the 3. The report is of course unnecessary, since the node was inserted before S started scanning the structure, but T_1 does not know that. T_2 did see in time that no report is necessary, and that is why it did not report the deletion. The trouble is that, since the deletion is not reported, reporting the insertion is not only unnecessary, but also harmful.

We solve this problem by exploiting again the fact that a node is marked prior to its deletion. An insertion will be reported in the following manner (lines 31-35).

- Read PSC, and record a private pointer to the snap-collector object, SC.
- Check whether there is an ongoing iteration, by calling SC.IsActive().
- If not, return. If there is, check whether the node you are about to report is marked.
- If it is, return without reporting. If it is not marked, then report it.

The above scheme solves the problem of harmfully reporting an insertion. If the node was unmarked after the relevant ITERATION has already started, then a later delete operation that still takes place before the linearization of the iteration will see that it must report the node as deleted. There is, however, no danger of omitting a necessary

report; if a node has been deleted, there is no need to report its insertion. If the delete occurred before the linearization of the iteration, then the iteration does not include the node. If the delete occurred after the linearization of the iteration, then the insert must be present after the linearization of the iteration as well (since it had a chance to see the node is marked), and therefore it is possible to set the linearization of the insertion after the iteration as well.

4.2 Performing an Iteration

A thread that desires to perform an ITERATION first reads the PSC pointer and checks whether the previous iteration has already been linearized by calling the IsActive method (line 53). If the previous iteration has already been linearized, then it cannot use the same snapshot, and it will allocate a new snap-collector. After allocating it, it will attempt to make the global PSC pointer point to it using a CAS (line 56). Even if the CAS fails, the thread can continue by taking the new value pointed by the PSC pointer, because the linearization point of the new snap-collector is known not to have occurred before the thread started its ITERATION operation. Therefore, this CAS doesn't interfere with wait-freedom, because the thread can continue even if the CAS fails.

A snapshot of the data structure is essentially the set of nodes present in it. The iterating thread scans the data structure, and uses the snap-collector to add a pointer to each node it sees along the way (lines 62-68), as long as this node is not marked as logically deleted. The iterating thread calls the `AddNode` method of the snap-collector for this purpose.

When the iterating thread finishes going over all the nodes, it is time to linearize the snapshot (and iteration). It calls the `Deactivate` method in the snap-collector for this purpose (this is similar to setting ongoingScan to zero in Jayanti's algorithm). Afterwards, further calls to the IsActive method will return false. An INSERT, DELETE, or CONTAINS operation that will start after the deactivation will not report to this snap-collector object. If a new ITERATION starts, it is no longer able to use this snap-collector, and so it allocates a new one.

To ensure proper linearization in the presence of multiple iterating threads, some further synchronization is required between them. A subtle implied constraint is that all threads that iterate concurrently and use the same snap collector object must decide on the same snapshot view. This is needed, because the linearization point of operations that occur concurrently with the closure of the snapshot picture is determined by whether they appear in the snapshot or not. So if an operation appears in the snapshot view of one thread but not in a snapshot view of another, then the linearization argument fails.

To assure the snapshot is consistent for all threads we enforce the following. First, before a thread calls the Deactivate method, it calls the `BlockFurtherNodes` (line 66). The snap-collector ensures that after a call of BlockFurtherNodes returns, further invocations of AddNode cannot install a new pointer, or have any other effect. Second, before the first iterating thread starts putting together the snapshot according to the collected nodes and reports, it blocks any further reports from being added to the snap-collector. This is achieved by invoking the `BlockFurtherReports` method (line 69). From this point on, the snap-collector is in a read-only mode.

Next, the iterating thread assembles the snapshot from the nodes and reports stored in the snap-collector. It reads them using the `ReadPointers` and `ReadReports` methods. A node is in the snapshot iff: 1) it is among the nodes added to the snap-collector OR there is a report indicating its insertion AND 2) there is no report indicating its deletion.

Calculating the snapshot according to these rules can be done efficiently if the nodes and reports in the snap-collector are sorted first. As explained in Section 5.1, the snap-collector is optimized so that it holds the nodes sorted throughout the execution, and thus sorting them requires no additional cost. The reports, however, still need to be sorted. Finally, once the iterating thread assembled the snapshot, it can trivially perform an iteration, by simply going over the nodes present in the snapshot one after the other. Thus, the overall complexity of an ITERATION is O(#nodes + #reports*log(#reports)).

5 The Snap-Collector Object

One can think of the snap-collector object as holding a list of node pointers and a list of reports. The term *install* refers to the act of adding something to these lists. Thus, the snap-collector enables the iterating threads to install pointers, and the modifying threads to install reports. It supports concurrent operations, and it must be wait-free since it is designed as a building block for wait-free and lock-free algorithms. The semantics and interface of the snapshot object follow. To relate the new algorithm to the basic one, we also mention for each method (*in italics*), its analogue in the single scanner snapshot. *Tid* is short for Thread Identifier.

- NewSnapCollector(). *No equivalent*. Allocates a new snap-collector object.
- AddNode(Node* node, int tid). *Analogue to copying a register into array C*. Installs a pointer to the given node. May fail to install the pointer if the BlockFurtherPointers() method (see below) has previously been invoked.
- Report(Report* report, int tid). *Analogue to reporting a new value in array B*. Installs the given report. May fail to install the report if the BlockFurtherReports() method (see below) has previously been invoked.
- IsActive(). *Analogue to reading the ongoingScan binary field*. Returns true if the Deactivate() method has not yet been called, and false otherwise. (True means the iteration is still ongoing and further pointers might still be installed in the snapshot object.)
- BlockFurtherPointers(). *No analogue. Required to synchronize between multiple iterators*. After this method is completed, any further calls to AddNode will do nothing. Calls to AddNode concurrent with BlockFurtherPointers may fail or succeed arbitrarily.
- Deactivate(). *Analogue to setting ongoingScan to false*. After this method is complete, any call to IsActive returns false, whereas before this method is invoked for the first time, IsActive returns true.
- BlockFurtherReports(). *No analogue. Required to synchronize between multiple iterators*. After this method is completed, any further calls to Report will do nothing. Calls to Report concurrent with BlockFurtherReports may succeed or fail arbitrarily.

```
General:  The data structure will hold an
additional field, PSC, which is a pointer to a
snap-collector object.
1.  Initialize()
2.     Initialize the data structure as usual
3.     PSC = (address of) NewSnapCollector()
4.     PSC->Deactivate()
5.
6.  Delete(int key)
7.    search for a node with required key
8.    if not found
9.       return false
10.  else   // found a victim node with the key
11.     mark the victim node
12.     ReportDelete(pointer to victim)
13.     physically remove the victim node
14.     return true
15.
16. Insert(Node n)
17.    search for the place to insert the node
         n as usual, but before removing
         a marked node, first call ReportDelete()
18.    If n.key is already present in the data
          data structure on a different node h
19.       ReportInsert(pointer to h)
20.       return false
21.    else
22.       Insert n into the data structure
23.       ReportInsert(pointer to n)
24.       return true
25.
26. ReportDelete(Node *victim)
27.   SC = (dereference) PSC
28.   If (SC.IsActive())
29.      SC.Report(victim, DELETED)
30.
31. ReportInsert(Node* newNode)
32.    SC = (dereference) PSC
33.    if (SC.IsActive())
34.       if (newNode is not marked)
35.          Report(newNode, INSERTED)
36.
37. Contains(int key)
38.    search for a node n with the key
39.    if not found  then return false
41.    else if n is marked
42.       ReportDelete(pointer to n)
43.       return false
44.    else   ReportInsert(pointer to n)
45.            return true

46. TakeSnapshot()
47.    SC = AcquireSnapCollector()
48.    CollectSnapshot(SC)
49.    ReconstructUsingReports(SC)
50.
51. AcquireSnapCollector()
52.    SC = (dereference) PSC
53.    if (SC.IsActive())
54.       return SC
55.    newSC = NewSnapCollector()
56.    CAS(PSC, SC, newSC)
57.    newSC = (dereference) PSC
58.    return newSC
59.
60. CollectSnapshot(SC)
61.    Node curr = head of structure
62.    While (SC.IsActive())
63.       if (curr is not marked)
64.          SC.AddNode(pointer to curr)
65.       if (curr.next is null) // curr is the last
66.          SC.BlockFurtherNodes()
67.          SC.Deactivate()
68.       curr = curr.next
69.    SC.BlockFurtherReports()
70.
71. ReconstructUsingReports(SC)
72.    nodes = SC.ReadPointers()
73.    reports = SC.ReadReports()
74.    a node N belong to the snapshot iff:
75.     ((N has a reference in nodes
           OR  N has an INSERTED report)
          AND
          (N does not have a DELETED report)
```

Fig. 2. The Iterator

- ReadPointers(). *No analogue*. Returns a list of all the pointers installed in the snapshot object. Should be called only after BlockFurtherPointers is completed by some thread.
- ReadReports(). *No analogue*. Returns a list of all the reports installed in the snapshot object. Should be called only after BlockFurtherReports is called by some thread.

5.1 The Snap-Collector Implementation

The implementation of the snap-collector object is orthogonal to the iterator algorithm, but different implementations can affect its performance dramatically. This section briefly explains the particulars of the implementation used in this work.

The proposed implementation of the snap-collector object maintains a separate linked-list of reports for each thread. It also maintains a single linked-list of pointers to the nodes of the data structure, and one boolean field indicating whether it is currently active (not yet linearized).

IsActive, Deactivate. The IsActive method is implemented simply by reading the boolean field. The Deactivate method simply writes false to this field.

AddReport. When a thread needs to add a report using the AddReport method, it adds it to the end of its local linked-list dedicated to this thread's reports. Due to the locality of this list its implementation is fast, which is important since it is used also by threads that are not attempting to iterate over the data structure. Thus, it facilitates low overhead for threads that only update the data structure.

Although no other thread may add a report to the thread local linked- list, a report is still added via a CAS, and not a simple write. This is to allow the iterating threads to block further reports in the BlockFurtherReports method. However, when a thread adds a report, it does not need to check whether the CAS succeeded. Each thread might only fail once in adding a report for every new iteration. After failing such a CAS, it will hold that the IsActive method will already return false for this iteration and therefore the thread will not even try to add another report.

BlockFurtherReports. This method goes over all the threads local linked-lists of reports, and attempts by a CAS to add a special dummy report at the end of each to block further addition of reports. This method should only be invoked after the execution of the Deactivate method is completed. The success of this CAS need not be checked. If the CAS succeeds, no further reports can be added to this list, because a thread will never add a report after a dummy. If the CAS fails, then either another iterating thread has added a dummy, or a report has just been added. The first case guarantees blocking further repots, but even in the latter case, no further reports can now be added to this list, because the thread that just added this report will see that the snap-collector is inactive and will not attempt to add another report.

AddNode. The basic idea in the implementation of AddNode is to use the lock-free queue of Michael and Scott [12]. To install a pointer to a node, a thread reads the tail

pointer. If the tail node is last, it attempts to add its node after the last node and then fix the tail to point to the newly added node. If the tail node is not the last node, i.e., its next field hold to a non-null node, then the thread tries by a CAS to change the tail to point to the next node (similarly to [12]), and retries adding its node again.

Clearly, this implementation is not wait-free as the thread may repeatedly fail to add its node and make progress. We therefore use a simple optimization that slightly alters the semantics of the AddNode method. To this end, we note that nodes should be added to the snapshot view in an ascending order of keys. The AddNode method will (intentionally) fail to add any node whose key is smaller than or equal to the key of the last node added to the snap-collector. When such a failure happens, AddNode returns a pointer to the data structure node that was last added to the snap-collector view of the snapshot. This way, an iterating thread that joins in after a lot of pointers have already been installed, simply jumps to the current location. This also reduces the number of pointers in the snap-collector object to reflect only the view of a single sequential traverse, avoiding unnecessary duplications. But most importantly, it allows wait-freedom.

The snap-collector object still holds a tail pointer to the queue (which might at times point to the node before last). To enqueue a pointer to a node that holds the key k, a thread reads the tail pointer. If the tail node holds a key greater than or equal to k, it doesn't add the node and simply returns the tail node. If the tail node is not the last node, i.e., its next field hold to a non-null node, then this means that there is another thread that has just inserted a new node to the snapshot view. In this case, this new node is either the same node we are trying to add or a larger one. So in this case the thread tries by a CAS to change the tail to point to the next node (similarly to [12]), and then it returns the new tail, again without adding the new node.

This optimization serves three purposes: it allows new iterating threads to jumps to the current location; It makes the AddNode method fast and wait-free; and it keeps the list of pointers to nodes sorted by their keys, which then allows a simple iteration over the keys in the snapshot.

BlockFurtherNodes. This method sets the tail pointer of the nodes list to point to a special dummy with a key set to the maximum value and the node set to null. Combined with our special implementation of AddNode, further calls to AddNode will then read the tail's special maximum value and will not be able to add additional nodes.

ReadPointers, ReadReports. These methods simply return a list with the pointers / reports stored in the snap-collector. They are normally called only after the BlockFurtherNodes, Deactivate, and BlockFurtherReports methods have all been completed, thus the lists of pointers and reports in the snap-collector are immutable at this point.

5.2 Some Simple Optimizations

The implementation used for the performance measurements also includes the following two simple optimizations.

Elimination of many of the reports. An additional binary field was added to each node, initialized to zero. When a thread successfully inserts a node, and after reporting it if necessary, this binary field is set to 1. Future INSERT operations that fail due to this node, and future CONTAINS operations that successfully find this node, first check to see if this bit is set. If so, then they know that this node has been reported, and therefore, there is no need to report the node's insertion.

If a large portion of the operations are CONTAINS operations, as is the case in typical data structure usage, this optimization avoids a significant portion of the reports. This is because in such cases most of the reports are the result of successful CONTAINS operations. However, note that this optimization is not always recommended, as it adds overhead to the INSERT operations even if ITERATION is never actually called.

Avoidance of repeated sorting. After a single thread has finished sorting the reports, it posts a pointer to a sorted list of the reports, and saves the time it would take other threads to sort them as well, if they haven't yet started to do so.

6 Performance

In this section we report the performance of the proposed iterator, integrated with the lock-free linked-list and skiplist in Java. We used the linked-list implementation as included in the book "The Art of Multiprocessor Programming" by Herlihy and Shavit [10], and added to it the iterator mechanism described in this paper. For the skiplist, we used the Java code of ConcurrentSkipListMap by Doug Lea, and added our mechanism. We also measured the performance of the CTrie, which is the only other lock-free data structure with comparable semantics that supports ITERATION. The CTrie is included in the Scala 2.10.0 distribution, and we used this implementation to measure its performance.

All the tests were run on SUN's Java SE Runtime, version 1.6.0, on a system that features 4 AMD Opteron(TM) 6272 2.1GHz processors. Each processor has 8 cores (32 cores overall), and each core runs 2 hyper-threads (i.e., 64 concurrent threads overall). The system employs a memory of 128GB and an L2 cache of 2MB per processor.

The algorithms were tested on a micro-benchmark in which one thread repeatedly executes ITERATION operations, going over the nodes one by one continually. For the other threads, 50% of the operations are CONTAINS, 25% are INSERT, and 25% are DELETE, with the number of threads varying between 1-31. In each test the keys for each operation were randomly and uniformly chosen in the ranges $[1, 32]$, $[1, 128]$, or $[1, 1024]$. In each test, all the threads were run concurrently for 2 seconds. All the tests were run in one long execution. The different data structures were run alternately: for a specific test-case parameters (i.e., the number of threads and the key range) first the linked-list was run for a 2 seconds interval, then the CTrie, and then the skiplist. After a single 2 seconds interval run of each data structure, the next test-case was run for all the three. After all the test-cases were completed once, a second iteration of the tests was initiated. The execution consisted of overall 16 such iterations; however, the first iteration was omitted from the results, and only served to allow the JVM the time to warm up. The averages of the other 15 iterations are reported in the figures.

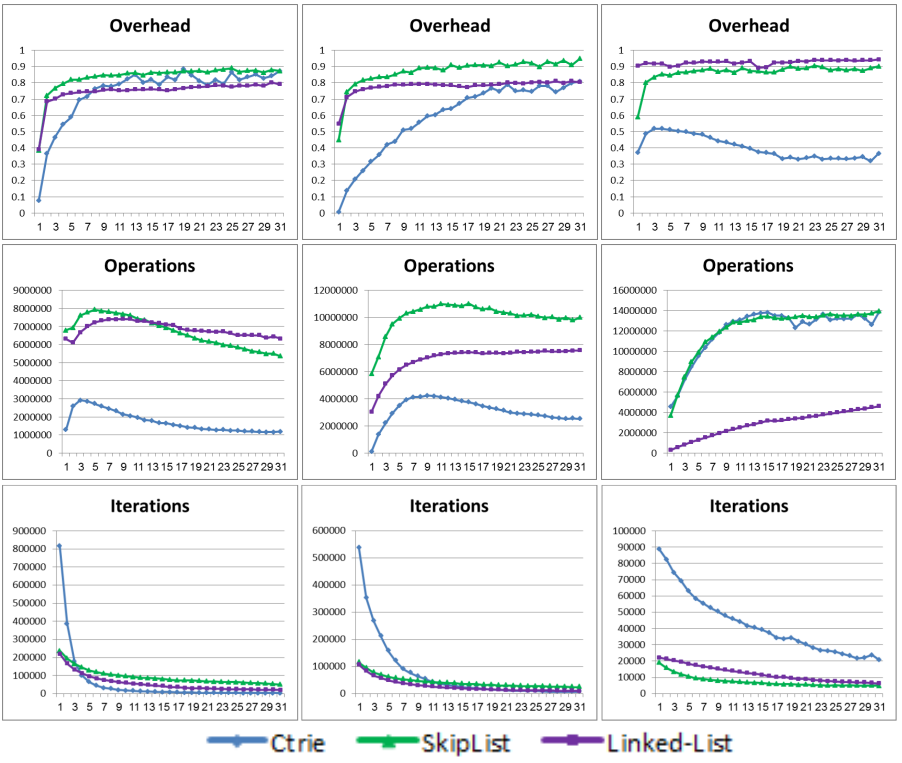

Fig. 3. Results for 32 possible keys (left) 128 possible keys (middle) 1024 possible keys (right)

For each key range, we present three different graphs. In the first graph, we measure the number of operations executed as a fraction of the number of operations executed without the additional iterating thread. For example, for a range of keys $[1, 32]$, for 16 threads, the number of operations executed while an additional thread is continually iterating the nodes is 86% of the number of operations executed by 16 threads in the skiplist data structure that *does not support iteration* at all. Thus, this graph presents the cost of adding the support for an iterator, and having a single thread continually iterate over the structure. For the CTrie, there is no available lock-free implementation that does not support iteration at all, so we simply report the number of operations as a fraction of the number of operations executed when there is no additional concurrent thread iterating over the structure. In the second graph, we report the absolute number of INSERT, DELETE, and CONTAINS operations executed in the different data structures while a single thread was iterating, and in the third graph we report the number of ITERATION operations that the single thread completed. This last measure stands for the efficiency of the iterator itself.

The results appear in Figure 3. In general, the results show that the iterator proposed in this paper has a small overhead on the other threads (which execute INSERT, DELETE and CONTAINS), and in particular, much smaller than the overhead imposed by the

CTrie iterator. The overhead of the proposed iterator for other threads is usually lower than 20%, except when the overall number of threads is very small. This means that the proposed iterator does relatively little damage to the scalability of the data structure. As for overall performance, we believe it is less indicative of the contribution of our work, as it reflects mainly the performance of the original data structures regardless of the iterator. Having said that, the linked-list performs best for 32 keys, the skiplist for 128 keys, and the CTrie and skiplist performs roughly the same for 1024 keys.

References

1. Afek, Y., Dolev, D., Attiya, H., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. In: PODC, pp. 1–13 (1990)
2. Afek, Y., Shavit, N., Tzafrir, M.: Interrupting snapshots and the java™ size() method. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 78–92. Springer, Heidelberg (2009)
3. Anderson, J.H.: Multi-writer composite registers, pp. 175–195 (1994)
4. Braginsky, A., Petrank, E.: A lock-free b+tree. In: SPAA, pp. 58–67 (2012)
5. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. In: PPOPP, pp. 257–268 (2010)
6. Ellen, F., Fatourou, P., Ruppert, E., van Breugel, F.: Non-blocking binary search trees. In: PODC, pp. 131–140 (2010)
7. Fatourou, P., Kallimanis, N.D.: Time-optimal, space-efficient single-scanner snapshots & multi-scanner snapshots using cas. In: PODC, pp. 33–42 (2007)
8. Fomitchev, M., Ruppert, E.: Lock-free linked lists and skip lists. In: Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, pp. 50–59. ACM, New York (2004)
9. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 300–314. Springer, Heidelberg (2001)
10. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
11. Jayanti, P.: An optimal multi-writer snapshot algorithm. In: STOC, pp. 723–732 (2005)
12. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. ACM Symposium on Principles of Distributed Computing (PODC), pp. 267–275 (1996)
13. Prokopec, A., Bronson, N.G., Bagwell, P., Odersky, M.: Concurrent tries with efficient non-blocking snapshots. In: PPOPP, pp. 151–160 (2012)
14. Riany, Y., Shavit, N., Touitou, D., Touitou, D.: Towards a practical snapshot algorithm. In: ISTCS, pp. 121–129 (1995)
15. Timnat, S., Braginsky, A., Kogan, A., Petrank, E.: Wait-free linked-lists. In: PPOPP, pp. 309–310 (2012)