Lock-free Linearizable 1-Dimensional Range Queries

Bapi Chatterjee

Chalmers University of Technology, Gothenburg, Sweden
{bapic}@chalmers.se

Abstract. In this paper, we present a generic method to perform a linearizable range search in lock-free ordered 1-dimensional data-structures. The algorithm requires single-word atomic compare-and-swap (`CAS`) primitives. We experimentally evaluate the range search algorithm in a lock-free linked-list, skip-list and BST. The experiments demonstrate that the proposed range search algorithm is scalable even in the presence of high percentage of concurrent modify operations and outperforms an existing range search algorithm in lock-free k-ary trees in several scenarios.

1 Introduction

1.1 Background

A number of lock-free data structures exist in the literature to implement an ordered set abstract data type (ADT): singly-linked lists [1, 2], doubly-linked lists [3], skip lists [4, 5], BSTs [6–11], etc. The ADT implemented by these data structures support set operations - ADD, REMOVE and CONTAINS. However, these operations essentially require single point queries and therefore are inherently different from operations such as range search that requires a consistent view of the data structure with respect to multiple points stored in it. The most common consistency framework used in concurrent settings is linearizability [12], which makes a user perceive an operation to take effect instantaneously at a point between the invocation and response of the operation. In effect, linearizability of range search is highly desired because it ensures that the output provides an "aligned view", with respect to a global real-time order, of the targeted entries in the data structure. The wide availability of multi-core processors and a surge in the popularity of in-memory databases have led to significant interest in designing lock-free search data structures that efficiently support linearizable range search.

To our knowledge, the only work existing in the literature with regard to a linearizable range search in a lock-free data structure is by Brown et al. [13], who implemented it in lock-free k-ary search trees. Their method requires range scanning using a depth-first-search followed by validating the scan, and if any node is found outdated, the range scan is retried, often repeatedly. Thus, in the cases of multiple concurrent updates, this method lets the range search starve. Avni et al. [14] used software transactional memory to encapsulate the range search queries in a lock-based concurrent skip list, which they call a Leap list. However, the STM based approaches are known to incur increasing overhead with the growth in the number of target shared-memory words covered by a transaction and thus may not be an efficient approach to perform queries for long ranges in a reasonably populated concurrent data structure. Sagonas et al. [15], proposed to implement range search by way of locking each of the nodes, which contain the points in the target range, in a lock-based data structure.

This paper makes the following contributions:

1. We present a generic method to implement linearizable RANGESEARCH operations in a lock-free ordered search data structure that supports linearizable ADD, REMOVE and CONTAINS operations.
2. Our method can be seamlessly integrated to any lock-free 1-dimensional ordered data structure.
3. We experimentally show that the proposed range search algorithm achieves good scalability and outperforms an existing range search method [13] in high contention scenarios.

1.2 A summary of the work

A naive approach to perform a concurrent linearizable range search, without using locks or transactional memory support, is to collect a lock-free linearizable snapshot of the data structure and output the appropriate subset thereof. However, not many lock-free data structures support linearizable snapshot collection. Exception is - lock-free hash trie by Prokopec et al. [16]. They used a variant of double-compare-single-swap (`DCSS`) primitive, which is implemented using single-word `CAS`. Here, an update with a concurrent snapshot

requires copying each updated node together with the path from the root to it to help the snapshot collection, which results to a performance deterioration of concurrent updates with snapshot. Also, the hash mapping of the keys makes it hard to compute a range search directly from a collected snapshot.

Petrank and Timmat [17] presented another approach for collecting a linearizable snapshot of lock-free linked lists and skip lists based on the wait-free multi-writer multi-scanner *snapshot object* of Jayanti [18]. A snapshot object, which consists of multiple shared-memory words, supports concurrent `scan` and `update` methods. An `update` writes a new value at one of the words, and a `scan` returns an atomic view of all the words. Unlike the approach of [13], [17] does not require (often repeated) restarts, which otherwise causes poor scalability in the cases of high concurrent modifications. Basically, they augment a data structure with a pointer to a special object called *snap-collector* that provides a *platform* for a set operation to *report* a modification to a concurrent snapshot collection operation. Concurrent scanners use a single snap-collector to return the same snapshot. Nevertheless, directly using a "full" snapshot collection for a RangeSearch ultimately is undesirable as it discards the advantage of mostly small size of a range query output.

Attiya et al. [19] introduced the notion of *partial snapshot objects*. A partial snapshot collection is essentially a generalization of the snapshot collection, in which the target is restricted to only a part of the multi-word object. Their fundamental idea is based on multiple concurrent announcements of *partial* `scan` operations. To handle the announcements, they used the idea of an *active set object* of Afek et al. [20] to construct a set of concurrent processes that collect partial snapshots. An active set provides the methods - `join`, `leave` and `get_set`. A `join` method is called to join the set of "active" processes, a `leave` is called to leave such a set, and `get_set` outputs the list of processes which are part of the active set. Thus, an active set implementation keeps track of the processes that "currently" perform a partial `scan`. Imbs et al. [21] optimized this work with respect to the number of helping steps.

The work in [19] underlines the motivation that a partial snapshot collection must be cheaper than a (full) snapshot collection. Hence, to implement a lineaeizable range search in a lock-free data structure, without facing a poor scalability in the presence of concurrent modifications, our work proposes a solution drawing from [19] and utilizing the snap-collector design of [17]. We build an active set of processes that perform range search and equip them with *range-collector* objects. A range-collector is essentially a snap-collector as described in [17] with a fixed *associated* range.

To realize an active set, we use a lock-free linked-list of range-collectors. We augment a data structure with a pointer to one of the ends of this list, say the *head*. A new range-collector is allowed to be added only at the other end, say the *tail*. Thus, the addition to, removal from and traversal through this lock-free list delegate the active set methods `join`, `leave` and `get_set`, respectively. A RangeSearch operation starts with traversing through the list and checks whether there is an *active* range-collector with a coincident target range. If in the list such an active range-collector is found, it is used for a concurrent *coordinated* range scan, which is analogous to using same snap-collector by multiple concurrent scanners in [17]. A range-collector is removed from the lock-free linked-list as soon as the range scan at it gets over.

Now, unlike the concurrent snapshot collections, where they output the same snapshot by anchoring their linearization points at the same atomic step, there are issues with concurrent range scans with overlapping but non-coincident associated ranges. Suppose op_1 and op_2 are two such concurrent RangeSearch operations. First, if op_1 has its linearization point before that of op_2, and it includes a data-point k in its output, and there is no linearization point of a Remove(k) between the linearization points of op_1 and op_2, op_2 must also include k in its output for consistency, if k happened to belong to the intersection of the associated ranges. That effectively means that op_1 and op_2 must synchronize before their returns. Second, because a range-collector has a fixed associated range, and the associated ranges of op_1 and op_2 are non-coincident, we can not use a single range-collector for both op_1 and op_2. It implies that, if for concurrent and independent progress we use different range-collectors for op_1 and op_2, a concurrent set operation, say op_3, will have to report to both the range-collectors, if its return point happens to belong to the intersection of the two associated ranges. Naturally, it increases the overhead of reporting, which is undesirable. To solve these issues, if op_1 successfully joins the active set before op_2 by adding its range-collector to the lock-free list, we first make op_2 help op_1 to finish, and then let it restart. Thus at any time-point, the augmented list contains active range-collectors with disjoint ranges only.

Please note that, similar to [17], our approach to implement range search in a lock-free data-structure is orthogonal to the actual data structure design, and thus is generic. Lastly, because the range search in data structures storing multidimensional points is not similar to that in 1-dimensional data structures, for detail see Samet's book [22], we do not claim that our method can be directly adapted to multidimensional range search problem in a concurrent setting, which in its own merit is very important but yet largely unexplored.

2 The Lock-Free Range Search

Fundamentally, our algorithm presents a generalization of the lock-free iterator algorithm of [17]. For the sake of reference, we have presented a brief recap of the algorithm of [17] in the Appendix A.

The lock-free data structures that employ the presented range search algorithm use a lazy procedure, which is now well known [1–11], to implement a REMOVE operation; that is, there is an atomic `CAS` step, which on successful execution, indicates a *logical removal* of the target node, and following that the logically removed node is detached from the data structure. Thereby a REMOVE operation that logically removes a node, first reports it to an appropriate range-collector if required, and then goes to take steps to detach the node. The CONTAINS operations that return `true`, first report the node to an appropriate range-collector, and then return. The ADD operations, if returning `false` on finding a node containing the query key present in the data structure, behave similar to a successful CONTAINS operation, whereas if it successfully adds a new node to the data structure, reports that before returning `true`.

The implementation of the lock-free linked-list of range-collectors has absolutely similar semantics as the lock-free linked-list of [23], except the fact that no addition happens anywhere in the middle of the list. Removal of a range-collector, say c, takes two successful `CAS` steps: first we inject a `mark` descriptor at the link connecting c to its successor using a `CAS` and then modify the link emanating from the predecessor of c and incoming to it, to connect to the successor with a `CAS`. An addition of a range-collector to this list requires a single successful `CAS` execution. The detail algorithm with pseudo-code is presented in Appendix B.

The linearizability arguments of operations in an execution of our algorithm are absolutely similar to those in [17]. The concurrent RANGESEARCH operations that use same range-collector, share the atomic step that *deactivates* the range-collector, to determine their linearization points. To obtain a total order, we can arrange such concurrent RANGESEARCH operations in the order of their invocation points. The linearization points of other set operations remain unaltered.

3 Experimental Evaluation

We implemented the presented algorithms in Java using RTTI. The experimental setup is given in Appendix C. We evaluated the introduced range search algorithm in three lock-free data-structures - (a) H_LinkedList: linked list of [1] , (b) Im_Ex_BST: a lock-free external BST in which a non-recursive traversal is facilitated using parent-links; the code is available at [24] (c) ConcSkipList: the skip list of [5].

We thankfully obtained the basic code of [17] from the authors in a personal communication. They used `AtomicMarkedReference` objects of java.util.concurrent.atomic library in the snap-collector implementation as well as in the linked list of [1], whereas the java library code of [5] was used for the skip list. To optimize the code, we aligned all the implementations to RTTI by replacing every instance of `AtomicMarkedReference` object with a volatile variable and used `AtomicReferenceFieldUpdater` on top of it for `CAS`.

We compared the algorithm with the range search implementation of [13] (BA_KST64), which is based on Java RTTI. We used the author's code available at their home-page. In the experiments in [13], $k = 64$ produced the best results among the k-ary search trees. Therefore, we chose to compare our implementation with the one with $k = 64$.

To simulate the variation due to the contention, we selected the combination of key-range, percentage of operations and the number of threads as following: (a) the % of (ADD, REMOVE, CONTAINS, RANGESEARCH) $\in \{(05, 05, 89, 01), (05, 05, 88, 02), (25, 25, 89, 01), (25, 25, 88, 02)\}$, (b) $|\{key \in K\}| \in \{10^2, 10^3\}$ and (c) the number of threads $\in \{2, 4, 8, 16, 28, 32\}$.

In the experiments, the keys are selected at random from the chosen key-ranges following a uniform distribution. Additionally, all the threads in the experiment perform all the operations selecting next operation at random with a probability expressed by the distribution percentage. In RANGESEARCH experiments, we recorded the throughput of the method SIZE which computes the size of the node-set returned by a call of RANGESEARCH. To call a RANGESEARCH([lo, hi]), we randomly selected two keys from the chosen key-range

and passed their min as lo and their max as hi. For a linearizable SIZE method in BA_KST64, we used the length of the return of the `subset()` method thereof.

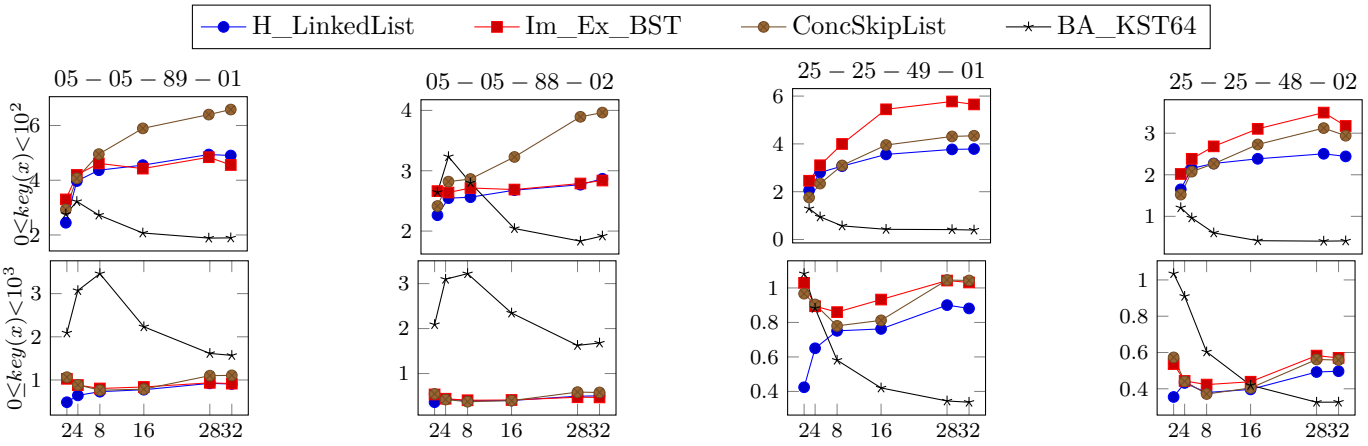

Fig. 1: Performance in terms of the throughput ($\frac{1000\times\#Ops}{ms}$) vs. $\#threads$, varying the range of the keys and the distribution of operations in terms of ADD%-REMOVE%-CONTAINS%-RANGESEARCH %.

3.1 Observations and Discussion

Figure 1 demonstrates the experimental observations. We observed a completely different behavior of our implementation compared to that of BA_KST64 with regards the performance and overhead.

- With the growth in the number of modify operations (compare the plots of columns 1 and 2 together to 3 and 4), our method substantially outperforms BA_KST64 (even linked list performs better than KST64 in smaller data-structure with high modification percentage). This is expected from the growing number of validations required in BA_KST64 with the growth in modify operations.
- In all the cases, as the number of threads increases, our method exhibits good scalability whereas it is opposite in BA_KST64. This again can be understood in terms of increasing number of validations required in high contention scenarios in BA_KST64.
- Among the data-structures considered for evaluating our generic method, in high contention cases BST outperforms the skip list, whereas in cases of low contention and smaller size of the data structure, skip list wins over the BST. This can be explained in terms of higher number of steps required to find the successor of a node in an external BST.
- On increasing the percentage of RANGESEARCH operations (compare the plots of column 1 with 2 and 3 with 4), the throughput of our algorithm decreases across the data-structure types, whereas we do not see similar throughput change in BA_KST64. It indicates that our method has marginally higher overhead compared to BA_KST64, specifically when the number of concurrent threads and percentage of modify operations is low. It can be explained in terms of the fundamental difference in the snapshot collection strategies ([25] vs. [18]). We make CONTAINS operations report to the concurrent RANGESEARCH, whereas in BA_KST64, CONTAINS do not bother about RANGESEARCH. The experimental evaluations in [17] also showed somewhat similar behaviour with respect to the comparison between throughput and overhead of [17] and [16].

4 Conclusion

In this paper we presented a generic method to perform linearizable range search in lock-free 1-dimensional ordered data structures. Our experiments showed that the proposed range search method scales well in high contention scenarios.

The k-ary tree by Brown et al. [13] used dirty bits in nodes. We observed that this method achieves better throughput in low contention scenarios compared to our method. We can design a hybrid method that uses similar strategy to achieve scalability of RANGESEARCH together with lower overhead in low contention scenarios, whereas falls back to the presented method when percentage of concurrent modification is high. Also, we can explore the design of a lock-free BST which facilitates faster computation of successor of a given node, for instance, by connecting the leaves.

References

1. T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *15th DISC*, 2001, pp. 300–314.
2. M. Fomitchev and E. Ruppert, "Lock-free linked lists and skip lists," in *23rd PODC*, 2004, pp. 50–59.
3. H. Sundell and P. Tsigas, "Lock-free and practical doubly linked list-based deques using single-word compare-and-swap," in *9th OPODIS*. Springer, 2005, pp. 240–255.
4. ——, "Fast and lock-free concurrent priority queues for multi-thread systems," *J. Parallel Distrib. Comput.*, vol. 65, no. 5, pp. 609–627, 2005.
5. D. Lea, "`ConcurrentSkipListMap`," in *`java.util.concurrent`*.
6. F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, "Non-blocking binary search trees," in *29th PODC*, 2010, pp. 131–140.
7. S. V. Howley and J. Jones, "A non-blocking internal binary search tree," in *24th SPAA*, 2012, pp. 161–171.
8. A. Natarajan and N. Mittal, "Fast concurrent lock-free binary search trees," in *19th PPoPP*, 2014, pp. 317–328.
9. T. Brown, F. Ellen, and E. Ruppert, "A general technique for non-blocking trees," in *19th PPoPP*, 2014, pp. 329–342.
10. B. Chatterjee, N. Nguyen, and P. Tsigas, "Efficient lock-free binary search trees," in *33rd PODC*, 2014, pp. 322–331.
11. F. Ellen, P. Fatourou, J. Helga, and E. Rupert, "The amortized complexity of non-blocking binary search trees," in *33rd PODC*, 2014, pp. 332–341.
12. M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.
13. T. Brown and H. Avni, "Range queries in non-blocking k-ary search trees," in *16th OPODIS*. Springer, 2012, pp. 31–45.
14. H. Avni, N. Shavit, and A. Suissa, "Leaplist: lessons learned in designing tm-supported range queries," in *32nd PODC*. ACM, 2013, pp. 299–308.
15. K. F. Sagonas and K. Winblad, "Efficient support for range queries and range updates using contention adapting search trees," in *Languages and Compilers for Parallel Computing - 28th International Workshop, LCPC 2015, Raleigh, NC, USA, September 9-11, 2015, Revised Selected Papers*, 2015, pp. 37–53.
16. A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky, "Concurrent tries with efficient non-blocking snapshots," in *Acm Sigplan Notices*, vol. 47, no. 8. ACM, 2012, pp. 151–160.
17. E. Petrank and S. Timnat, "Lock-free data-structure iterators," in *Distributed Computing*. Springer, 2013, pp. 224–238.
18. P. Jayanti, "An optimal multi-writer snapshot algorithm," in *37th STOC*, 2005, pp. 723–732.
19. H. Attiya, R. Guerraoui, and E. Ruppert, "Partial snapshot objects," in *20th SPAA*, 2008, pp. 336–343.
20. Y. Afek, G. Stupp, and D. Touitou, "Long-lived adaptive collect with applications," in *40th FOCS*, 1999, pp. 262–272.
21. D. Imbs and M. Raynal, "Help when needed, but no more: efficient read/write partial snapshot," *Journal of Parallel and Distributed Computing*, vol. 72, no. 1, pp. 1–12, 2012.
22. H. Samet, *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.
23. T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *DISC*, vol. 1. Springer, 2001, pp. 300–314.
24. B. Chatterjee, "`ConcurrentSet`," in *https://github.com/bapi/ConcurrentSet*.
25. Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic snapshots of shared memory," *Journal of the ACM (JACM)*, vol. 40, no. 4, pp. 873–890, 1993.
26. M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *15th PODC*, 1996, pp. 267–275.

A Snap-collector implementation

The lock-free data-structures considered in [17] are - linked-list [1] and skip-list [5], in which each node x has a unique key $key(x)$ and it is straightforward to find $next(x)$. The data-structures implement an ADT that provides the operations ADD(x), REMOVE(x) and CONTAINS(x). Without any ambiguity, we denote ADD($key(x)$) by ADD(x). In these data-structures, the operation REMOVE(x) takes more than one atomic `CAS` steps. One of the `CAS` steps, whose success ensures that x will be eventually removed, is generally known as *logical remove*, and is considered the linearization point of REMOVE(x).

To implement the snapshot collection, a pointer PSC to an object called *snap-collector* is maintained in the data-structure. A snap-collector basically holds a boolean variable *isActive* and two separate groups of lists: (a) a single sorted list L of node pointers and (b) two arrays A and R of lists of *reports*. A report comprises a node pointer and a *report-type* (ADD or REMOVE). The size of A and R is equal to the number of processes in the system. A snap-collector with false *isActive* field is called *inactive*, otherwise *active*. If there is no ongoing snapshot collection, the pointer PSC holds a reference to an inactive snap-collector. Initially PSC points to a dummy snap-collector which is always inactive.

A snapshot collection starts with reading the pointer PSC. If PSC is found pointing to an inactive snap-collector, a new active snap-collector is allocated. The snap collector attempts to update PSC to point to the new snap-collector using a `CAS`. If the `CAS` fails then it helps the concurrent operation that successfully injects its own snap-collector. In both the situations (success in injecting own snap-collector or helping), the process scans the data-structure and collects the address of every node x that it discovers which is not logically removed. The list L (which has very similar semantics as the lock-free queue of Micheal et al. [26]) is used for the purpose in which the node $tail(L)$ satisfies $key(l) \leq key(tail(L))$ for every node $l \in L$. To optimize the scanning by multiple concurrent processes, a process is allowed to insert a node-pointer $^*x$ in L only if $key(x) > key(tail(L))$. An attempt to insert $^*x$ in L returns $tail(L)$ if $key(x) \leq key(tail(L))$ (without making any changes in L) and $^*x$ is returned if it was successfully added to L. The *isActive* variable is checked before every attempt to read a node from the data-structure and if it is found false then that finishes the snapshot collection. On completion of scanning the data-structure, *isActive* is set false. This is the linearization point of the snapshot. Further, in order to ensure that all the processes have the same view of L, before attempting to set false at *isActive*, a null is inserted at L.

For a linearizable snapshot collection, a process performing ADD/ REMOVE/ CONTAINS requires to report the operation. A REMOVE(x), on finding x, performs steps up to the logical remove of x, then before taking further steps, adds a report (report-type: REMOVE) consisting $^*x$ to the report-list pointed by $R[tid]$, where tid denotes process-id. An ADD(x) after adding x with a successful `CAS`, reports $^*x$ to the $A[tid]$ list of a concurrent active snap-collector. Before every ADD report, the logically remove status of $^*x$ is checked to avoid an unnecessary reporting. A CONTAINS(x) on finding a node reports (ADD or REMOVE) to an active snap-collector. To ensure a consistent view of A and R by all the processes in the system, before the *isActive* field of a snap-collector is set false, a null report is added to each of the $A[tid]$ and $R[tid]$ report-lists. After the linearization of the snapshot, the reports are processed by sorting them and then merging them to L. A processed snapshot contains address to the nodes present in the tree during the execution interval of the snapshot collection. The concurrent processes working at the same snap-collector return same snapshot.

B Lock-free linearizable range search algorithm

The pseudo-code of the algorithm is given in Figures 2 and 3. We have sufficiently commented the pseudo-code to describe the functionality of the methods. We consider an ordered lock-free data-structure with nodes of type DSNode which are indexed by the keys k uniquely selected from a partially ordered universe. The data-structure is provided with a lock-free unordered list of objects of type RCNode (range-collector node), line 6 to line 10. An RCNode, in addition to containing the lists L, A and R similar to a snap-collector object, contains a pointer *next* to connect to another node in the list. It also records the range in terms of the lower limit *lo* and the upper limit *hi*. An important difference between an RCNode and a snap-collector node of [17] is the list L: we store additional info in a node (of type **Window**, line 5) of L that can facilitate traversal in the data-structures (like BST) in which, unlike linked list and skip list, computing $next(x)$ is not straightforward.

```
▷ data-structure-node.
1  struct DSNode {K k; ···;} *DSNPtr;
▷ report consisting a DSNPtr and report-type.
2  struct Report {▷ RptType: ADD, REM.
3    RptType rt; DSNPtr node;
4  } *RptPtr;
▷ packet of a DSNPtr and auxilliary info to find next node.
5  struct Window {DSNPtr cur;···;};
6  struct RCNode {▷ range-collector-node.
7    K lo, hi; Window * L;
8    RptPtr * A, R;▷ Array-Size=#Threads.
     ▷ mark bit of next is used.
9    (RCNPtr ref, bool mark) next;
10 } *RCNPtr;
▷ Global shared variables to initialize.
11 RTail = RCNode(∞, ∞, null, null, (null, 1));
12 RHead = RCNode(-∞, -∞, null, null, (*RTail, 1));
13 ···;▷ Other global variables.
▷ Returns the Window of DSNPtr with key just ≥ lo; returns null if no such node present.
14 FindMinNode(K lo)
▷Returns a Window of DSNPtr with key ≥ key(x).
15 Next(Window x)
▷ Returns true if x is not logically removed.
16 IsPresent(DSNPtr x)
▷ Returns true if range-collection at rn is not linearized i.e. its next field's mark is 0.
17 IsActive(RCNode rn)
▷ Returns the Window of DSNPtr with largest key in rn·L after attempting to add x to it.
18 AddNode(RCNode rn, Window x)
▷ Adds a Window of DSNPtr with max-value key to rn·L and sets 1 at the mark of rn's next.
19 Deactivate(RCNode rn)
20   AddNode(rn, Window(*DSNode(∞),···));
21   BTS(rn·next·mark, 1);
▷ Atomically adds a dummy report to each of A[tid] and R[tid] lists of rn.
22 BlockFurtherReports(RCNode rn)
▷ Returns whether the interval [lo,hi] isConatined/contains/overlaps/isDisjoint with the interval of the range-collector-node rn.
23 ChkOvrlap(K lo, K hi, RCNode rn)
▷ Returns a list of keys in interval [lo,hi] from the range-collector-node rn.
24 ProcessRange(K lo, K hi, RCNode rn)
▷ Returns true if range-collection at rn is not linearized i.e.its next field's mark is 0.
25 Report(RCNode rn, Report r, int tid)
▷ Collects the nodes in [lo,hi] to add to rn·L.
26 SnapRange(K lo, K hi, RCNode rn)
27   w = FindMinNode(lo);
28   if w·cur≠null then
29     while IsActive(rn) and w·cur.key≤hi do
30       if IsPresent(w·cur) then
31         w = Next(AddNode(rn, w));
32   Deactivate(rn); BlockFurtherReports(rn);
▷ Finds appropriate active concurrent range-collection to report node x.
33 SyncWithRangeQuery(DSNPtr x, RptType report, int tid)
34   (ref, *)=RHead→next;
35   while ref≠RTail do
36     if IsActive(ref) and ref.lo≤x.key≤ref.lo then
37       if report == ADD and IsPresent(x) then
38         Report(ref, Report(x, ADD), tid);
39       else Report(ref, Report(x, REM), tid) ;
40       break;
41     else (ref, *)=ref→next ;
▷ Returns a list of keys in interval [lo,hi].
42 RangeSearch(K lo, K hi)
43   pre=RHead; cur=pre→next; mode=INIT;
44   retry:
45   while true do
46     (ref, m)=cur→next;
47     while ref ≠ null and m==1 do
48       if !CAS(pre→next, (cur, 0), (ref, 0)) then
49         goto retry;
50       (ref, m)=cur→next; pre=cur; cur=ref;
51     if ref==null then
52       if mode==CLEAN then
53         return ProcessRange(lo, hi, R);
54       R=RCNode(lo, hi, null, null, (*RTail, 0));
55       if CAS(pre→next, (cur, 0), (*R, 0)) then
56         SnapRange(lo, hi, R);
57         (ref, *)=R→next; mode=CLEAN;
58         if CAS(pre→next, (R, 0), (ref, 0)) then
59           return ProcessRange(lo, hi, R);
60       else goto retry;
61     else if ChkOvrlap(lo, hi, cur)==isConatined then
62       SnapRange(cur·lo, cur·hi, cur);
63       R=cur; mode=CLEAN; (ref, *)=R→next;
64       if CAS(pre→next, (R, 0), (ref, 0)) then
65         return ProcessRange(lo, hi, R);
66     else if ChkOvrlap(lo, hi, cur)==isDisjoint then
67       (ref, m)=cur→next; pre=cur; cur=ref;
68     else
69       SnapRange(cur·lo, cur·hi, cur);
70       (ref, *)=cur→next;
71       CAS(pre→next, (cur, 0), (ref, 0));
```

Fig. 2: The Range-collector Algorithm

The unordered list has similar semantics as the lock-free linked-list of Harris [1], except that here a new node can be added only at one end. The unordered list is represented by two sentinel RCNodes *RHead* and *RTail* with ranges [-∞ -∞] and [∞ ∞], respectively, s.t. they are disjoint to any RCNode (-∞ and ∞ are the minimum and the maximum elements, respectively, in the partially ordered universe of keys). At the initialization of the data-structure, the *next* of *RHead* points to *RTail* and L, A and R of both these RCNodes are null. Additionally, the *next* of *RTail* is always null. A new RCNode is added using a `CAS` such that its *next* pointer points to *RTail*. On `CAS` failure we restart from *RHead*. To remove an RCNode from the unordered list, first its *next* pointer is marked (one unused bit is set using a `CAS`) and then the *next* pointer of the previous RCNode is updated. Any traversal through the list always helps a pending remove in the path. The method RangeSearch of the data-structure (see line 42 to line 65) intrinsically uses add and remove of an RCNode in the unordered list together with the traversal through the list.

A process intending to perform RangeSearch([*lo hi*]) starts scanning the unordered list starting from *RHead*. A variable *mode* is assigned value `INIT` to indicate that the process is yet to start the collection of desired data-structure nodes, line 43. On reading an RCNode *cur* with range [x y], one of the following is done depending on how [*lo hi*] relates to [x y]:

1. If the *next* of *cur* is marked and not null (line 47), it indicates a pending but linearized RangeSearch operation. We help to clean the RCNode *cur* from the unordered list.
2. If the *next* of *cur* is null (line 51), i.e. *cur* is the node *RTail*, a new RCNode R is allocated and added. After that, the method SnapRange (line 26 to line 32) is called to collect the snapshot of the desired subset of the data-structure. In an execution of SnapRange, the collection of nodes from the data-structure and storing them in the list L works exactly the same way as in a snap-collector object. However, unlike a snap-collector, in which a separate boolean variable is used to maintain the active status, we use the mark-bit of the *next* of the RCNode for the purpose. The method Deactivate (line 19 to line 21) is called which consists of two steps - adding a node with key ∞ to the list L and a test-and-set (`BTS`) (`CAS` is equally applicable) to set the mark-bit of *next*. The `BTS` works as the linearization point of the RangeSearch operation. After this step any call to IsActive(R) will return false. Clearly, the terminal node of L with key ∞ causes termination of the collection of any further node. Additionally, the SnapRange method also calls the method BlockFurtherReports which works along the same lines as in a snap-collector and thus ensures that no further concurrent modify operation can be reported to R. On completion of SnapRange, *mode* is changed to `CLEAN` and the RCNode R is attempted to be detached from the unordered list. If the `CAS` to detach the node fails, the traversal is restarted. If *mode* is set to `CLEAN` then reaching the node *RTail* indicates that the targeted RCNode has been detached. Finally, the method ProcessRange is called to return the desired data-structure-nodes. ProcessRange includes steps of sorting the added reports and combining the nodes from L to produce a set of nodes with keys in the range [*lo hi*].
3. If the *next* of *cur* is neither marked nor null, it indicates an active RangeSearch, therefore we check the relation between ([*lo hi*]) and ([x y]) by calling ChkOvrlap and
   (a) if the relation is isConatined (line 61), we help the undergoing SnapRange at *cur* and ProcessRange is used to return only those nodes which have the keys ∈ [*lo hi*].
   (b) if the relation is isDisjoint (line 66), we simply move to the next RSNode as *cur* does not cover any data-structure-node with key in the range [*lo hi*].
   (c) and finally, if the relation is contains or overlaps (line 68), we help the undergoing SnapRange at *cur* and the traversal is restarted from *RHead*.

A Contains, Add or Remove operation, which is concurrent to a RangeSearch, reports the data-structure-nodes in a similar way as it does in [17]. An object **Report** consists of the address of the node to be reported and the type of report (`ADD` or `REM`). To report a node x, the method SyncWithRangeQuery is called. SyncWithRangeQuery traverses through the unordered list to locate an RCNode which is active and has the range ∋ *key*(x). However, during the traversal no RangeSearch is helped. Before reporting an `ADD`, the method IsPresent is called to check whether X is logically removed. If no relevant RCNode is found then nothing is reported. A Contains (line 87 to line 92) operation on finding the target node reports it as `ADD`

```
    ▷ Returns the DSNode with key equal key;
    returns null if no such node present.
72  Find(K key)
    ▷ Adds DSNode (key) to return true if no such
    node present else returns false.
73  Add(K key, int tid)
74    while true do
75      if (x=Find(key))≠null then
76        if IsPresent(x) then
77          SyncWithRangeQuery(x, REM, tid);
78          return false;
79        else
80          SyncWithRangeQuery(x, REM, tid);
81          ···;▷ Complete Remove.
82          continue;
83      else
84        ···;▷ Complete Add.
85        SyncWithRangeQuery(x, ADD, tid);
86        return true;
```

```
    ▷ Returns true if there exists a DSNode with
    key equal key else false.
87  Contains(K key, int tid)
88    if (x=Find(key)) == null then return false;
89    if IsPresent(x) then
90      SyncWithRangeQuery(x, ADD, tid); return true;
91    else
92      SyncWithRangeQuery(x, REM, tid); return false;
    ▷ Removes the DSNode (key) to return true;
    returns false if no such node present.
93  Remove(K key, int tid)
94    if (x=Find(key)) == null then return false;
95    if IsPresent(x) then
96      ···;▷ Logically remove DSNode (key).
97      SyncWithRangeQuery(x, REM, tid);
98      ···;▷ Complete Remove.return true;
99    else
100     SyncWithRangeQuery(x, REM, tid);
101     ···;▷ Complete Remove.return false;
```

Fig. 3: A Lock-free data structure algorithm that employs the linearizable range search algorithm

if it is not logically removed else REM is reported. If the node is not found, there is nothing to report. An Add operation (line 73 to line 86) first adds the desired node, then calls the method SyncWithRangeQuery to report ADD. If a node with query key is found in the data-structure, which is not logically removed, then it behaves as a Contains operations. On finding a node with the query key but logically removed, a REM is first reported, after that the pending Remove operation is helped and then the Add is reattempted. A Remove (line 93 to line 101) on finding a node with the query key, attempts to logically remove the node, reports REM of the node, then completes the remaining steps to clean the node.

Similar to [17], our RangeSearch implementation in a lock-free data-structure does not change the usual linearization points of the Add/Remove/Contains operations. The unordered-linked list guarantees that no two RangeSearch, whose range overlap, can collect their snapshots simultaneously. The proof of lock-freedom is lengthy but the intuition is straightforward as in [17].

C Experimental Setup

We used a machine with a dual chip Intel(R) Xeon(R) E5-2695 v3 processor with 14 hardware processes per chip (28 hardware processes in total with hyper-processing) running at 2.30 GHz. The machine has 64 GB of RAM and runs over CentOS Linux 7.1.1503 (Kernel version: 3.10.0-229.el7.x86_64) with Java HotSpot(TM) 64-Bit Server VM (1.8.0_51) with 1 GB initial heap size and 15.6 GB maximum heap size. All the implementations were compiled using `javac` version 1.8.0_51 and the runtime flags -d64 -server were used. We performed 10 repetitions of 5 seconds runs for each combination of the parameters shown in the graphs. The average over the 10 trials are recorded. The keys in all the data-structures are taken of Integer type.