Lock-free Contention Adapting Search Trees

Kjell Winblad
Uppsala Univeristy
kjell.winblad@it.uu.se

Konstantinos Sagonas
Uppsala Univeristy
konstantinos.sagonas@it.uu.se

Bengt Jonsson
Uppsala Univeristy
bengt.jonsson@it.uu.se

ABSTRACT

Concurrent key-value stores with range query support are crucial for the scalability and performance of many applications. Existing lock-free data structures of this kind use a fixed synchronization granularity. Using a fixed synchronization granularity in a concurrent key-value store with range query support is problematic as the best performing synchronization granularity depends on a number of factors that are difficult to predict, such as the level of contention and the number of items that are accessed by range queries. We present the first lock-free key-value store with linearizable range query support that dynamically adapts its synchronization granularity. This data structure is called the lock-free contention adapting search tree (LFCA tree). An LFCA tree does local adaptations of its synchronization granularity based on heuristics that take contention and the performance of range queries into account. We show that the operations of LFCA trees are linearizable, that the lookup operation is wait-free, and that the remaining operations (insert, remove and range query) are lock-free. Our experimental evaluation shows that LFCA trees achieve more than twice the throughput of related lock-free data structures in many scenarios. Furthermore, LFCA trees are able to perform substantially better than data structures with a fixed synchronization granularity over a wide range of scenarios due to their ability to adapt to the scenario at hand.

CCS CONCEPTS

• **Software and its engineering → Concurrent programming structures**; • **Information systems** → *Unidimensional range search*; *Key-value stores*;

KEYWORDS

concurrent data structures, range queries, lock-freedom, adaptivity, linearizability

ACM Reference Format:
Kjell Winblad, Konstantinos Sagonas, and Bengt Jonsson. 2018. Lock-free Contention Adapting Search Trees. In *SPAA '18: 30th ACM Symposium on Parallelism in Algorithms and Architectures, July 16–18, 2018, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3210377.3210413

SPAA '18, July 16–18, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5799-9/18/07…$15.00
https://doi.org/10.1145/3210377.3210413

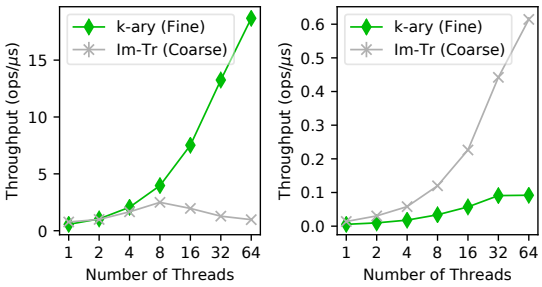
1 INTRODUCTION

On multicore machines, concurrent key-value stores (maps) with range query support are crucial for the scalability of applications such as big scale data processing and in-memory databases (e.g., Google's F1 [21] and Yahoo's Flurry [24]). It is thus of no surprise that the multicore revolution has motivated a number of data structures of this type, e.g., [1, 2, 4, 17]. A key-value store represents a set of items (keys), each with an associated value. Sets can be seen as a simplification of key-value stores that do not have any values associated with the items. From here on, we will discuss sets but we note that what applies to sets also applies to key-value stores as sets can trivially be modified to become key-value stores.

Concurrent sets that support both single-item operations (insert, remove and lookup[1]) and multi-item operations (e.g., range query and clone[2]) face the following dilemma: Single-item operations usually benefit from as fine-grained synchronization as possible, as this leads to few conflicts. In contrast, multi-item operations usually benefit from more coarse-grained synchronization, as this leads to less time spent on synchronization-related overhead (e.g., fewer locks need to be acquired). We say that the *conflict time* for an operation is the amount of the time during which the operation can conflict with other concurrent operations. More coarse-grained synchronization can also lead to short conflict times for operations in concurrent sets that internally use and exploit immutable data in a way that we will now explain. We will use the lock-free k-ary search tree [4] to illustrate how immutable data can be used to make the conflict time for range queries short. Lock-free k-ary search trees store all items inside immutable leaf nodes that can contain k items each. The insert and remove operations of such trees work by replacing leaf nodes. A range query Q in a k-ary search tree first collects all the immutable leaf nodes that Q needs. The range query Q ends its conflict time and linearizes once this collection phase finishes. The items that Q needs to return are scanned from the collected leaf nodes after Q's conflict time. This removal of this scanning from the conflict time is possible due to the immutability of the leaf nodes.

An even more extreme way to exploit immutability in range queries is to store all items in a single immutable data structure. Such a data structure, that we call *Im-Tr*, is constructed from a single mutable reference pointing to an immutable balanced search tree T. A new instance of T reflecting an update (insert or remove) can be constructed in $O(\log n)$ time (where n is the number of items before the update) as one only needs to "copy" nodes on a path from the root to a leaf to create the new instance [12]. The insert and remove operations of *Im-Tr* change the mutable reference using an atomic

[1] An insert operation inserts an item (replacing an existing item if one with an equal key already exists), the remove operation removes an item with the given key if such an item exists and the lookup operation returns an item with the given key if such an item exists.
[2] A range query operation returns all items with keys within the given range (specified by two keys) and clone makes a clone of the data structure.



(a) Using small range queries (b) Using large range queries

Figure 1: Throughput of coarse- vs fine-grained synchronization.

compare-and-swap (CAS) operation so the reference points to a new immutable instance reflecting the update. Using this scheme, which is also described by Herlihy [9], it is trivial to perform range queries with constant conflict time as they only need to get a snapshot by reading the mutable reference and then perform the range query in the snapshot.

Figure 1[3] illustrates the scalability of a data structure that uses a fixed relatively fine-grained synchronization (the lock-free k-ary search tree [4] with the k parameter set to 64) and one that uses coarse-grained synchronization (*Im-Tr*) in two scenarios that only differ in the sizes of the ranges involved in the range queries. As shown in the graphs, fine-grained synchronization achieves superior scalability when the size of ranges in range queries is small (Fig. 1a) as the mutable reference of *Im-Tr* gets heavily contended, while *Im-Tr* with coarse-grained synchronization has superior scalability when range queries are large (Fig. 1b) as the k-ary search tree's range queries have long conflict times in this scenario.

As the above example illustrates, having concurrent sets with a fixed synchronization granularity is far from ideal. A natural way to deal with this problem is to design sets that can adapt their synchronization granularity to the workload at hand. In earlier work [17], we have described the first data structure that dynamically changes its synchronization granularity based on heuristics that take both the performance of single-item operations and multi-item operations into account, called the contention adapting search tree (*CA tree* for short). The CA tree is lock-based.

The lock-based CA tree performs well in a variety of scenarios due to its ability to adapt its synchronization granularity to the workload at hand. However, lock-based data structures are prone to a number of problems that are inherent from the use of locks, e.g., waiting, priority inversion, convoying, and lock overhead (memory usage, acquire and release time). The performance of lock-based algorithms is also heavily dependent on the lock implementation itself and on scheduling. For all these reasons, lock-free data structures that guarantee system-wide progress even in the presence of adversary scheduling can be preferable to lock-based ones. Furthermore, it is crucial with a wait-free (i.e., makes progress within a bounded number of steps) lookup operation for applications where lookups are very common, which the lock-based CA tree is lacking.

[3]The benchmark used to produce the graphs in Fig. 1 was executed on an Intel machine with 64 logical cores using the Oracle JVM. With small and large range queries we refer to range queries that on average include about 2.5 items and 25k items respectively. Further details about the experiment can be found in Section 7.

This paper presents a lock-free variant of the CA tree, called the *lock-free contention adapting search tree* (*LFCA tree* for short). LFCA trees support lock-free insert, remove and range query operations as well as a wait-free lookup operation. Operations update the data stored in an LFCA tree by swapping immutable leaf nodes pointing to immutable balanced search trees storing the actual items. The granularity of the LFCA tree is adjusted by splitting and joining such leaf nodes that also store a statistics value which is updated based on CAS successes or failures as well as based on the number of leaf nodes range queries need to access.

The technique that the LFCA tree uses for supporting range queries is interesting in its own right as it is applicable to other lock-free data structures such as the lock-free k-ary search tree [4]. A range query operation is performed by replacing the needed leaf nodes with nodes of a special node type that contains the information needed by other threads to help in completing the operation. The previously proposed method for doing range queries in the k-ary search tree [4] is prone to starvation [2, 22].

Overall, we claim that LFCA trees are important concurrent data structures as they provide an unique set of desirable properties:

Efficient non-blocking operations Our experimental comparison shows that LFCA trees achieve substantially better throughput than the best of the competing lock-free data structures over a wide range of scenarios. Also, LFCA trees perform better than lock-based CA trees in many scenarios (especially in scenarios with more threads than hardware threads).

Configuration-less As an LFCA tree automatically adjusts its structure using heuristics, there is no need for the user to configure the LFCA tree to use a certain synchronization granularity.

Adaptive As the synchronization granularity changes with the workload, the data structure can perform very well even when the workload changes during the lifetime of the data structure. As the adaptations of the granularity happen through local changes, the data structure can even adapt to scenarios where the workload is different in different parts of the data structure.

Flexible Performance characteristics of an LFCA tree can be changed by providing a different set implementation. We do not experiment with this property, but see no reason why the LFCA tree would be different from the lock-based CA tree in this regard [17].

Outline: We start with a high level description of how LFCA trees work and an overview of related work (Section 3) before we describe the algorithm in detail (Section 4) and give a proof sketch (Section 5). An optimization for range queries is described (Section 6) before we experimentally evaluate the LFCA tree (Section 7) and conclude the paper (Section 8).

2 A BIRD'S EYE VIEW OF LFCA TREES

A lock-free CA tree consists of *route nodes* (round boxes in Fig. 2a) and *base nodes* (square shaped boxes in Fig. 2a). The route nodes form a binary search tree with the base nodes as leaves. The actual items that are stored in the set represented by an LFCA tree are located in immutable data structures rooted in the base nodes, called *leaf containers*. All operations use the binary search tree property of the route nodes to find the base node(s) whose leaf container(s) should contain the items involved in the operation if they exist. An update operation (insert or remove) is illustrated

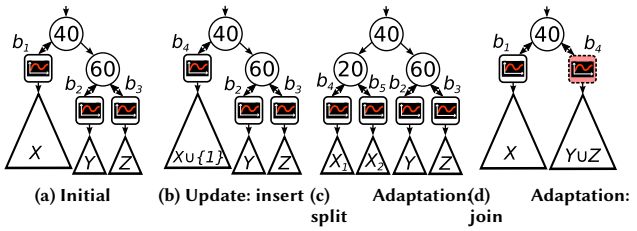


Figure 2: LFCA Trees illustrating various operations.

by Figs. 2a and 2b. An update operation uses a compare-and-swap (CAS) to attempt to replace a base node b_1 with a new base node b_4 reflecting the update, until the update succeeds. The update can be made efficient even though the data structure in the base node is immutable, since many immutable balanced tree data structures (e.g., red-black trees and treaps) support creating a new instance with an update in $O(\log n)$ time, where n is the number of items in the data structure [12, 19]. Before an update operation returns it checks whether the statistics value stored in the updated base node indicates that a structural adaptation should happen. The first kind of adaptation, called *split*, is illustrated by Figs. 2a and 2c. A split aims at reducing the contention in the LFCA tree and replaces a base node b_1 with a route node linking together two new base nodes (b_4 and b_5) so that approximately half of the original items are in each of them. The second kind of adaptation, called *join*, is illustrated with Figs. 2a and 2d and aims at optimizing the structure of the LFCA tree for range queries that span multiple base nodes and for situations where the contention is low. A join splices out a base node b_2 and its parent and replaces the base node b_3 with a new base node b_4 containing the items of both b_2 and b_3. Splits and joins of the base nodes can also be supported efficiently (i.e., in $O(\log n)$ time, where n is the number of items in the involved instances) in many immutable balanced tree data structures (e.g., treaps [19] that are used by our LFCA tree implementation).

3 RELATED WORK

There are several data structures with range query support. The *SnapTree* by Bronson *et al.* [3] has an efficient linearizable clone operation that returns a copy of the data structure from which a range query operation can easily be derived. *SnapTree*'s clone operation waits for active update operations to complete and forces subsequent update operations to copy nodes lazily before node modifications, so that the clone is not modified.

The lock-free k-ary search tree is an external (i.e. the items are stored in leaf nodes) unbalanced search tree with up to k keys stored in every node [5]. Range queries in k-ary search trees are performed by doing a read scan and a validation scan of the immutable leaf nodes containing items in the range [4]. The range query operation needs to retry if the validation scan fails. The k-ary search tree is an example of the fixed synchronization granularity approach discussed in the introduction. Another example of this approach based on software transactional memory is the Leaplist [1]. Both the k-ary search tree and the Leaplist make use of immutable data structures to reduce the conflict time of range queries in the way explained in the introduction. As they both use arrays as their immutable data structures, updates become very expensive when the parameter that decides both the synchronization granularity and the maximum size of the immutable data structures is set too high. Even though this problem could be fixed by using immutable balanced search trees instead of the arrays, they would still use a fixed synchronization granularity and would thus only perform well in certain scenarios.

Chatterjee has proposed a general method for performing range queries in lock-free ordered set data structures [6] based on an idea for doing snapshots by Petrank and Timnat [14]. Chatterjee's method makes use of a list of so-called range collector objects that all updates and range queries need to access. Unfortunately, the scalability of Chatterjee's method suffers from a global sequential hot spot in the list of range-collector objects that all range queries have to modify in the worst case.

The *KiWi* data structure by Basin *et al.* [2] supports wait-free range queries and lookup operations as well as lock-free update operations. Update operations help range queries by storing additional versions of inserted items when it is needed for the range queries. Similarly to Robertson's data structure [15], *KiWi*'s range queries atomically increment a global version counter which is used by update operations to decide whether storing an additional version for an item is necessary. *KiWi*'s global version number counter is bound to become a scalability bottleneck with a high enough level of parallel range queries. LFCA trees do not suffer from such global scalability bottleneck as their range query operation only need to synchronize with update operations that operate on items in the same range as the range query.

The above non-adaptive set data structures with support for efficient range queries and scalable updates [1, 2, 4, 6, 15, 16] have range queries with a conflict time that depends at least linearly on the number of items covered by the range given to the range query. The CA trees (both the lock-based and the lock-free) can do much better over a wide range of scenarios as the conflict time of their range queries can be constant (i.e., independent of the number of items covered by the range query) and their heuristics work towards getting a good trade-off between range queries conflict time and the scalability of updates.

Even though the fundamental ideas behind the lock-based CA tree [17] and LFCA tree are the same, lock-freedom gives LFCA trees better progress guarantees that are of importance for real-time systems. The lookup operation of LFCA trees is wait-free and can thus perform efficiently regardless of how contended the data structure is, which is crucial for many applications as lookups often dominate the workload. Still, the lock-based CA tree has a few advantages over LFCA tree. The use of locks makes it possible to use mutable sequential data structures to store the items. This can be advantageous in systems where memory management is expensive and when range queries are infrequent, as it reduces the number of memory allocations that are needed. The use of locks also makes it easier to extend the interface of the data structure with more linearizable operations.

The technique for joining base nodes in LFCA trees has some similarities with the replace operation of non-blocking Patricia tries [20]. The replace operation deletes an item and adds another item in a way that appears atomic. Our join is simpler, as a single thread can "mark" the involved nodes in a non-collaborative fashion, since other threads can abort the join without affecting correctness.

Several works have previously explored the idea of dynamically switching between a data structure that uses coarse-grained synchronization to a data structure that use fine-grained synchronization in one transformation step [7, 11, 13]. The drawback of the global mode switching approach proposed in these papers compared to LFCA tree's approach is that the switch between the modes is time-consuming and coarse-grained, whereas the LFCA tree can smoothly transition between different levels of synchronization granularity. Work has also been done to adapt to contention in other types of data structures (e.g., [8]).

4 ALGORITHM

Pseudo-code for all the non-trivial parts of the lock-free CA tree can be found in Figs. 3 to 5 and 7. The pseudo-code is derived from a model of the lock-free CA tree implemented in the C programming language with some minor adjustments for readability. This section contains a detailed description of the algorithm and the pseudo-code. In the next section, a detailed proof sketch will be given, showing that the operations are linearizable and have the stated progress guarantees.

Node types. A lock-free CA tree is built from instances of the node types that are defined in Fig. 3, lines 14–52. Note that the keyword `with_fields_from` (on lines 26, 32 and 39) is used to add fields from another `struct` definition. All internal route nodes are of type `route_node` (lines 14–20). The route nodes contain a key field (line 15) which is used to direct searches for a specific item in the tree. Together, they form a binary search tree. Leaves are called base nodes (lines 21–42) and have a `data` field (line 22) that points to immutable data structure instances (called *leaf containers*) that contain the items that are in the represented set. That a base node B is of type `normal_base` (lines 21–25) indicates that B is not involved in an ongoing operation. A base node of type `range_base` is a node that currently is or has been involved in a range query. Similarly, a base node of type `join_main` or `join_neighbor` is a node that currently is or has been involved in a join operation.

Data in nodes that can be modified by more than one thread are marked with the modifier `atomic`. These fields can only be accessed by the atomic and sequentially consistent functions `aload` (that loads the value at a given address), `astore` (that stores the given value at the given address) and `CAS` (a compare-and-swap that stores the value of its third parameter at the location of a given address (first parameter) iff the value at the given address is equal to the second parameter and returns true, or returns false otherwise).

Lookup. The wait-free lookup operation (lines 135–138) calls the `find_base_node` function (pseudo-code for this and other similar functions appears in an extended version of this paper [23]), which traverses the route nodes using binary search until a base node is found, and then performs the lookup in the corresponding immutable data structure.

Insert and Remove. The functions for inserting and removing a single item (Fig. 4, lines 129–134) are performed by the `do_update` function (lines 106–127) with the remove or insert function and the item in question as arguments. The `do_update` function searches for a base node using the given key, and then tries to replace that base node and its leaf container with a new one in which the key

```
1  // Constants
2  #define CONT_CONTRIB     250      // For adaptation
3  #define LOW_CONT_CONTRIB 1        // ...
4  #define RANGE_CONTRIB    100      // ...
5  #define HIGH_CONT        1000     // ...
6  #define LOW_CONT         -1000    // ...
7  #define NOT_FOUND        (node*)1 // Special pointers
8  #define NOT_SET          (treap*)1// ...
9  #define PREPARING        (node*)0 // Used for join
10 #define DONE             (node*)1 // ...
11 #define ABORTED          (node*)2 // ...
12 enum contention_info { contended, uncontened, noinfo }
13 // Data Structures
14 struct route_node {
15   int key;                        // Split key
16   atomic node* left;              // < key
17   atomic node* right;             // >= key
18   atomic bool valid = true;       // Used for join
19   atomic node* join_id = NULL; // ...
20 }
21 struct normal_base {
22   treap* data = NULL;  // Items in the set
23   int stat = 0;        // Statistics variable
24   node* parent = NULL; // Parent node or NULL (root)
25 }
26 struct join_main with_fields_from normal_base {
27   node* neigh1;  // First (not joined) neighbor base
28   atomic node* neigh2 = PREPARING; // Joined n...
29   node* gparent; // Grand parent
30   node* otherb;  // Other branch
31 }
32 struct join_neighbor with_fields_from normal_base {
33   node* main_node  // The main node for the join
34 }
35 struct rs { // Result storage for range queries
36   atomic treap* result = NOT_SET; // The result
37   atomic bool more_than_one_base = false;
38 }
39 struct range_base with_fields_from normal_base {
40   int lo; int hi; // Low and high key
41   rs* storage;
42 }
43 enum node_type {
44   route, normal, join_main, join_neighbor, range
45 }
46 struct node with_fields_from normal_base, range_base,
47                              join_main, join_neighbor{
48   node_type type;
49 }
50 struct lfcat{
51   atomic node* root;
52 }
53 // Help functions
54 bool try_replace(lfcat* m, node* b, node* new_b){
55   if( b->parent == NULL )
56     return CAS(&m->root, b, new_b);
57   else if(aload(&b->parent->left) == b)
58     return CAS(&b->parent->left, b, new_b);
59   else if(aload(&b->parent->right) == b)
60     return CAS(&b->parent->right, b, new_b);
61   else return false;
62 }
63 bool is_replaceable(node* n) {
64   return (n->type == normal ||
65           (n->type == join_main &&
66            aload(&n->neigh2) == ABORTED) ||
67           (n->type == join_neighbor &&
68            (aload(&n->main_node->neigh2) == ABORTED ||
69             aload(&n->main_node->neigh2) == DONE)) ||
70           (n->type == range &&
71            aload(&n->storage->result) != NOT_SET));
72 }
```

Figure 3: Data structures and help functions.

```
73  // Help functions
74  void help_if_needed(lfcatree* t, node* n){
75    if(n->type == join_neighbor) n = n->main_node;
76    if(n->type == join_main &&
77       aload(&n->neigh2) == PREPARING){
78      CAS(&n->neigh2, PREPARING, ABORTED);
79    }else if(n->type == join_main &&
80             aload(&n->neigh2) > ABORTED){
81      complete_join(t, n);
82    }else if(n->type == range &&
83             aload(&n->storage->result) == NOT_SET){
84      all_in_range(t, n->lo, n->hi, n->storage);
85    }
86  }
87  int new_stat(node* n, contention_info info){
88    int range_sub = 0;
89    if(n->type == range &&
90       aload(&n->storage->more_than_one_base))
91      range_sub = RANGE_CONTRIB;
92    if (info == contended && n->stat <= HIGH_CONT) {
93      return n->stat + CONT_CONTRIB - range_sub;
94    }else if(info == uncontened && n->stat >= LOW_CONT){
95      return n->stat - LOW_CONT_CONTRIB - range_sub;
96    }else return n->stat;
97  }
98  void adapt_if_needed(lfcatree* t, node* b){
99    if(!is_replaceable(b)) return;
100   else if(new_stat(b, noinfo) > HIGH_CONT)
101     high_contention_adaptation(t, b);
102   else if(new_stat(b, noinfo) < LOW_CONT)
103     low_contention_adaptation(t, b);
104 }
105
106 bool do_update(lfcatree* m,
107                treap*(*u)(treap*,int,bool*), int i){
108   contention_info cont_info = uncontened;
109   while(true){
110     node* base = find_base_node(aload(&m->root), i);
111     if(is_replaceable(base)){
112       bool res;
113       node* newb = new node{
114         type = normal,
115         parent = base->parent,
116         data = u(base->data, i, &res),
117         stat = new_stat(base, cont_info)
118       }
119       if(try_replace(m, base, newb)){
120         adapt_if_needed(m, newb);
121         return res;
122       }
123     }
124     cont_info = contended;
125     help_if_needed(m, base);
126   }
127 }
128 // Public interface
129 bool insert(lfcat* m, int i){
130   return do_update(m, treap_insert, i);
131 }
132 bool remove(lfcat* m, int i){
133   return do_update(m, treap_remove, i);
134 }
135 bool lookup(lfcat* m, int i){
136   node* base = find_base_node(aload(&m->root), i);
137   return treap_lookup(base->data, i);
138 }
139 void query(lfcat* m, int lo, int hi,
140            void (*trav)(int, void*), void* aux){
141   treap* result = all_in_range(m, lo, hi, NULL);
142   treap_query(result, lo, hi, trav, aux);
143 }
```

Figure 4: Help functions and public interface.

has been removed or inserted (lines 113–119). The replacement is done using the `try_replace` function, which uses a CAS to attempt to change the pointer of the base node's parent to a new base node with the updated leaf container. If successful, the CAS operation is the linearization point of the operation; if unsuccessful, the whole operation is retried. A replacement attempt is made only if the found base node is replaceable (line 111 and lines 63–72). If the base node is irreplaceable, then it may be involved in another operation; in this case `do_update` will first attempt to help this operation (line 125 and lines 74–86) before proceeding.

Note that the new base node gets a value for its `stat` field which is based on the replaced node's `stat` field and type as well as on whether conflicting operations have been detected (i.e., a base node which was not replaceable has been found or a `try_replace` call has failed; see line 117 and lines 87–97). Once a base node has been successfully replaced, the update operation calls `adapt_if_needed` to adapt the granularity of the data structure if the heuristics suggests that this is beneficial (line 120 and lines 98–104), before returning.

Range queries. The range query operation (lines 139–143) first calls `all_in_range` (line 141) to create a snapshot of all base nodes in the requested range and then traverses the snapshot to complete the range query (line 142). The function `all_in_range` (lines 162–215) goes through all base nodes that may contain items in the range in ascending key order, starting with the base node containing the smallest keys, to replace them, using a CAS, by base nodes of another type. The base nodes are found by the functions `find_base_stack` and `find_next_base_stack` in a depth-first traversal of the concerned portion of the tree. This traversal uses a stack `s` to store the search path to the current base node. The replacing base node is of type `range_base` (lines 39–42), which has fields for specifying the range of the ongoing range query (`lo` and `hi`) as well as a `storage` field pointing to a result storage (lines 35–38). This storage has a `result` field, which is initially `NOT_SET` and will be set once the range query has been completed (line 211). A `range_base` node is initially irreplaceable as long as its `result` field is `NOT_SET`; see the `is_replaceable` function (lines 63–72). A range query's linearization point is when all concerned base nodes have been replaced and the `result` field of the result storage has been replaced by the actual result of the query (line 211). The `lo` and `hi` fields of nodes of the `range_base` type are used by operations that try to help an uncompleted range query in order to make an irreplaceable `range_base` node replaceable; see `help_if_needed` (lines 74–86). Thus, range queries achieve atomicity by ensuring that all base nodes containing items in the range are irreplaceable for a short instance, and are able to maintain the non-blocking progress guarantee by enabling other threads to help them.

Once a range query has been completed, a special field in the result storage associated with the range query is set to a value indicating if more than one base node were needed to complete the range query. This information is used by the `new_stat` function (lines 87–97) when calculating a statistics value for a base node. Before `all_in_range` returns, `adapt_if_needed` (lines 98–104) is called (line 213) with a random base node within the range as parameter. This is done to ensure that the structure of the tree stays up-to-date with the collected heuristics.

```
144 node* find_next_base_stack(stack* s) {
145   node* base = pop(s);
146   node* t = top(s);
147   if(t == NULL) return NULL;
148   if(aload(&t->left) == base)
149     return leftmost_and_stack(aload(&t->right), s);
150   int be_greater_than = t->key;
151   while(t != NULL){
152     if(aload(&t->valid) && t->key > be_greater_than)
153       return leftmost_and_stack(aload(&t->right), s);
154     else { pop(s); t = top(s); }
155   }
156   return NULL;
157 }
158 node* new_range_base(node* b, int lo, int hi, rs* s){
159   return new node{... = b, // assign fields from b
160                   lo = lo, hi = hi, storage = s}; }
161 treap*
162 all_in_range(lfcat* t, int lo, int hi, rs* help_s){
163   stack* s = new_stack();
164   stack* backup_s = new_stack();
165   stack* done = new_stack();
166   node* b;
167   rs* my_s;
168  find_first:b = find_base_stack(aload(&t->root),lo,s);
169   if(help_s != NULL){
170     if(b->type != range || help_s != b->storage){
171       return aload(&help_s->result);
172     }else my_s = help_s;
173   }else if(is_replaceable(b)){
174     my_s = new rs;
175     node* n = new_range_base(b, lo, hi, my_s);
176     if(!try_replace(t, b, n)) goto find_first;
177     replace_top(s, n);
178   }else if( b->type == range   && b->hi >= hi){
179     return all_in_range(t, b->lo, b->hi, b->storage);
180   }else{
181     help_if_needed(t, b);
182     goto find_first;
183   }
184   while(true){   // Find remaining base nodes
185     push(done, b);
186     copy_state_to(s, backup_s);
187     if(!empty(b->data) && max(b->data) >= hi) break;
188   find_next_base_node: b = find_next_base_stack(s);
189     if(b == NULL) break;
190     else if(aload(&my_s->result) != NOT_SET){
191       return aload(&my_s->result);
192     }else if(b->type == range && b->storage == my_s){
193       continue;
194     }else if(is_replaceable(b)){
195       node* n = new_range_base(b, lo, hi, my_s);
196       if(try_replace(t, b, n)) {
197         replace_top(s, n); continue;
198       } else {
199         copy_state_to(backup_s, s);
200         goto find_next_base_node;
201       }
202     }else{
203       help_if_needed(t, b);
204       copy_state_to(backup_s, s);
205       goto find_next_base_node;
206     }
207   }
208   treap* res = done->stack_array[0]->data;
209   for(int i = 1; i < done->size; i++)
210     res = treap_join(res, done->stack_array[i]->data);
211   if(CAS(&my_s->result,NOT_SET,res) && done->size > 1)
212     astore(&my_s->more_than_one_base, true);
213   adapt_if_needed(t, done->array[r() % done->size]);
214   return aload(&my_s->result);
215 }
```

Figure 5: Helper function for the range query operation.

Adaptations. The granularity of the immutable parts of a LFCA tree can be changed with two different types of adaptations. The first one, called *high-contention adaptation* (or *split*), splits the items in a base node into two new base nodes, in order to decrease the contention in a part of the tree where the contention has been high. The second type, called *low-contention adaptation* (or *join*), joins the content of two base nodes into a new base node, in order to improve the performance of range queries. Joins can potentially also improve the performance of the LFCA tree for uncontended single-item operations as a join can make the search paths to items shorter (because the part of the tree consisting of route nodes may be unbalanced), but joins may also make updates slightly more expensive (due to increased amount of memory allocation and coping when creating new instances of the leaf containers). An adaptation is issued by the function `adapt_if_needed` (lines 98–104) that is executed by the update operations (line 120) and by range queries (line 213). Whether an adaptation should occur and what kind of adaptation it should be is decided based on a statistics value calculated by the `new_stat` function (lines 87–97). High-contention or low-contention adaptation is issued if this statistics value is above (line 100) or below (line 102) a threshold, respectively. The `new_stat` function calculates the statistics value based on its two parameters: a base node and a parameter that is encoding information about detected contention. The core idea behind the heuristics is to make the synchronization more fine-grained in parts of the data structure where contention has been common and to make it more coarse-grained in parts where contention has been uncommon or where range queries often need to access more than one base node.

If no contention information is given to the `new_stat` function (as is the case when this function is called by `adapt_if_needed`), then the value is the value of the `stat` field in the base node subtracted by x, where x is a positive constant if it is a base node of type range whose corresponding range query was completed by reading more than one base node (lines 89–91). When update operations call `new_stat` to get the value that will be used for the `stat` field of the new base node that the update operation creates they also pass information whether contention was detected (cf. line 117). This information is used to increase the statistics value if contention has been detected (line 93), and decrease the statistics value otherwise (line 95). The constant that is used to increase the statistics value when contention is detected is larger than the constant that decreases the statistics value when no contention has been detected so that adaptations happen quickly when contention is common and to avoid frequent adaptations back-and-forth. The constants used in our heuristics can be found in lines 2 to 6.

High-contention adaptation. The function for high-contention adaptation (lines 277–287) splits the content of a base node `b` into two new base nodes that are linked together with a route node `r`. The function attempts to replace the base node `b` with `r` using a `CAS` operation (line 286). This replacement is atomic to other operations and does not change the contents of the tree.

Low-contention adaptation. The function for low-contention adaptation (lines 268–276) intuitively replaces two neighboring base nodes `b` and `n0` by a new node `n2`, which contains the union of the items in `b` and `n0`. It splices out `b` and its parent route node from

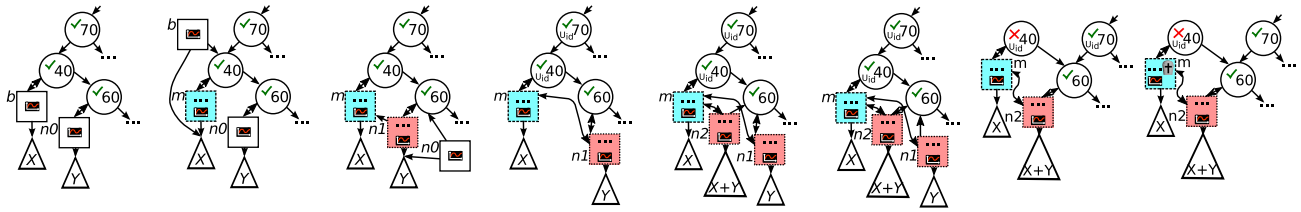


Figure 6: Figures showing the main steps of the low-contention join operation.

the tree and replaces `n0` by `n2`. Figure 6 illustrates the major steps which are done by a successful low-contention adaptation.

A successful low-contention adaptation (lines 268–276) is done in two phases. The first phase, illustrated in Figs. 6a to 6e and performed by `secure_join_left` in lines 216–250 (or the corresponding function for the right case), "marks" the part of the tree that will be involved in the join, to prevent other threads from changing it while the join is ongoing. Other threads cannot help complete this phase, but system-wide-progress is guaranteed as other threads can interrupt this phase by killing the join (line 78). The second phase (Figs. 6f to 6h), which is performed by `complete_join` (lines 251–267), is executed only if the first phase was successful (line 272). Other threads can help the join to complete this second phase (line 81). The second phase completes the join by splicing out `b` and its parent and replaces `n1` by `n2`.

We will describe how the low-contention adaptation works by going through a successful join of the base node `b` in Fig. 6a. As `b` is the left child of its parent, `secure_join_left` is called (line 270). Lines 217–222 find the neighbor `n0` of `b` (the leftmost leaf of `b`'s parent's right branch) and replaces `b` with a new base node `m` of type `join_main` (Fig. 6b). Note that the join would have aborted if `n0` would have been irreplaceable or if the replacement of `b` would have failed (which would have meant that `b` was no longer in the tree). Next, `n0` is replaced with a new base node `n1` of type `join_neighbor`, which is linked to `m` with the field `main_node` (line 227 and Fig. 6c). Both `m` and `n1` are now irreplaceable (lines 63–72). The only way for other threads to make them replaceable at this point is to set the field `neigh2` of `m` to `ABORTED`. On lines 228–233, the `join_id` field of both the parent and the grandparent of `m` is set to the reference `m` to make sure that they are not modified by any other join operation. Using the reference `m`, which is a unique identifier for the join operation, to mark the route nodes involved in the join makes it easy for threads to collaboratively change this field in the second phase. In lines 234–236, more information that is needed by `complete_join` to finish the operation is stored in `m` (cf. Fig. 6d). The final step of the first phase is done in lines 237-243 (cf. Fig. 6e). These lines attempt to set `m`'s field `neigh2` to a base node `n2` which can replace both `m` and `n1` using a `CAS` operation. If this `CAS` is successful, we know that the following are true directly after the change:

- Both `m` and `n1` must have been irreplaceable at the time of the change, as this means that the field `neigh2` has not been set to `ABORTED` by any other thread.
- The node referenced by `m`'s `parent` field is the parent of `m`, the node referenced by `m` `gparent` field is the grandparent of `m`, and `m`'s field `otherb` is set to the sibling of `m`. This follows from the observations that (i) the only type of change that can happen to a path from the root of the tree to a base node that is inside the tree is that a route node gets spliced out which can only happen if both the spliced out node and its parent has their `join_id` fields set to something different from `NULL`, (ii) the `join_id` field of the spliced out node is never set to `NULL` again, and (iii) no other threads can change the `join_id` field of parent and gparent while the `neigh2` field of `m` is set to `PREPARING`.

Once the `neigh2` field of `m` has been successfully set to the new base node, the first phase of the operation is finished and the second phase can start. Note that the changes done by the operation would have been rolled back on lines 245–248 if some `CAS` operations had failed.

As mentioned, the second phase of a low-contention adaptation is done by `complete_join` (lines 251–267). Multiple threads can execute this function with the same base node of type `join_main` as input parameter. This happens when another thread needs to change a base node of type `join_main` or `join_neighbor` and the join associated with the base node has finished the first phase but not yet the second (which means that the base node is irreplaceable; see lines 63–72). That other thread will call the `help_if_needed` function that in turn will call `complete_join` (line 81). The first modification that is done by the second phase is to replace the base node `n1` with the base node `n2` that is referenced to by the field `neigh2` in `m` which was set in the first phase (cf. Fig. 6f). Note that any thread that executes `complete_join` with the base node `m` as parameter can perform this step as it is done with a `CAS` operation (line 254 and the `try_replace` function). The next change is to set the `valid` flag of the parent to false (line 255). The sole purpose of this is so that the range query operation can avoid traversing branches that are no longer relevant for the range query (line 152). On lines 256–257, it is determined what will be the replacement of `m`'s parent. If `m` and `n2` share the same parent (in this case `b->otherb`, which is set on line 235, will be equal to `n1`, a.k.a. `b->neigh1`) the replacement will be `n2`, otherwise the replacement will be the branch of the parent that does not lead to `m` (this case is illustrated in Fig. 6f). The parent of `m` and `m` itself is spliced out from the tree on lines 258–265 (cf. Fig. 6g). Note that only one thread can succeed with the splice out as it is done with a `CAS` operation. Likewise, only one thread can succeed in resetting the `join_id` of the grandparent of `m` (line 262). The only remaining step after these lines have executed is to set the `neigh2` field of `m` to `DONE`, which is done on line 266 (cf. Fig. 6h). This is done to indicate that the join has completed and to make `n2` replaceable.

```
node* secure_join_left(lfcatree* t, node* b){
  node* n0 = leftmost(aload(&b->parent->right));
  if(!is_replaceable(n0)) return NULL;
  node* m = new node{
    ...  = b, // assign fields from b
    type = join_main};
  if(!CAS(&b->parent->left, b, m)) return NULL;
  node* n1 = new node{
    ...       = n0, // assign fields from n0
    type      = join_neighbor,
    main_node = m};
  if(!try_replace(t, n0, n1)) goto fail0;
  if(!CAS(&m->parent->join_id, NULL, m))
    goto fail0;
  node* gparent = parent_of(t, m->parent);
  if(gparent == NOT_FOUND ||
     (gparent != NULL &&
      !CAS(&gparent->join_id,NULL,m))) goto fail1;
  m->gparent = gparent;
  m->otherb = aload(&m->parent->right);
  m->neigh1 = n1;
  node* joinedp = m->otherb==n1 ? gparent: n1->parent;
  if(CAS(&m->neigh2, PREPARING,
         new node{... = n1, // assign fields from n1
                  type = join_neighbor,
                  parent = joinedp,
                  main_node = m,
                  data = treap_join(m, n1)}))
    return m;
  if(gparent == NULL) goto fail1;
  astore(&gparent->join_id, NULL);
 fail1: astore(&m->parent->join_id, NULL);
 fail0: astore(&m->neigh2, ABORTED);
  return NULL;
}
void complete_join(lfcatree* t, node* m){
  node* n2 = aload(&m->neigh2);
  if(n2 == DONE) return;
  try_replace(t, m->neigh1, n2);
  astore(&m->parent->valid, false);
  node* replacement =
    m->otherb == m->neigh1 ? n2 : m->otherb;
  if (m->gparent == NULL){
    CAS(&t->root, m->parent, replacement);
  }else if(aload(&m->gparent->left) == m->parent){
    CAS(&m->gparent->left, m->parent, replacement);
    CAS(&m->gparent->join_id, m, NULL);
  }else if(aload(&m->gparent->right) == m->parent){
    ... // Symmetric case
  }
  astore(&m->neigh2, DONE);
}
void low_contention_adaptation(lfcatree* t, node* b){
  if(b->parent == NULL) return;
  if(aload(&b->parent->left) == b){
    node* m = secure_join_left(t, b);
    if (m != NULL) complete_join(t, m);
  }else if (aload(&b->parent->right) == b){
      ... // Symmetric case
  }
}
void high_contention_adaptation(lfcatree* m, node* b){
  if(less_than_two_items(b->data)) return;
  node* r = new node{
   type  = route,
   key   = split_key(b->data),
   left  = new node{type = normal, parent= r, stat= 0,
                    data = split_left(b->data)}),
   right = ..., // Symmetric case
   valid = true};
  try_replace(m, b, r);
}
```

Figure 7: Low and high contention adaptation.

5 CORRECTNESS

In this section, we outline the main steps for proving that the algorithm is correct (in the sense of being a linearizable [10] implementation of a set of items which also supports range queries), and has the stated progress guarantees. We first provide global invariants, thereafter argue for linearizability, and finally for progress.

We define a route or base node to be *reachable* if it can be reached from the root of the tree by following `left` and `right` pointers. We define the *contents* of the tree as the union of all items in the leaf containers of reachable base nodes. The contents serve as the abstract state of the LFCA tree in our argument for linearizability. Finally, we define a base node to be *replaceable* if the `is_replaceable` function returns true when applied to it. The LFCA tree satisfies the following invariants and properties.

(1) The reachable route nodes and items in leaf containers of reachable base nodes are organized as a binary search tree. There is one exception to this, illustrated by Fig. 6f of a join operation, when the leaf container of node `m` is included in the leaf container of node `n2`. More precisely, this happens if a reachable route node `n` (corresponding to the parent of `m` in Fig. 6f) is such that `n->left` is a base node `m` of type `join_main` and *m*`->neigh2` is a pointer to a reachable node; if so, then the items in the leaf container of `m` are exactly the set of items in the leaf container of `m->neigh2` which are smaller than `n->key`. There is also a symmetric case when `n->right` is a base node of type `join_main`. Note that in these cases, the search `find_base_node` will reach the elements in `m`'s leaf container only via `n`, and after `n` is spliced out, the same elements will be found in the leaf container of `m->neigh2`. In both cases, the tree appears like a valid binary search tree.

(2) A replaceable base node can not become irreplaceable (but can be replaced by an irreplaceable node).

(3) A replaceable base node `n` is reachable iff `n` is obtained either as `n->parent->left` or as `n->parent->right`. A reachable node that has become non-reachable cannot become reachable again. The parent of a reachable and replaceable base node never changes.

Having established these properties of the LFCA tree, we can now proceed to establishing linearizability of operations.

update An update operation replaces a replaceable base node using a CAS operation, which succeeds only if the node is pointed to by the `left` or `right` pointer of its parent route node. By invariant (2) and (3), this ensures that the replaced node is reachable when the CAS succeeds. We let the linearization point of an update operation coincide with the successful CAS. Since the operation replaces a reachable node, the corresponding operation replaces a leaf container that is included in the contents of the tree just before the linearization point. To make sure that the operation indeed changes the contents of the tree in the intended way, we should make sure that it does not replace a node that is of type `join_main` or `join_neighbor`, in a situation where it participates in an uncompleted join and can have duplicated elements in another leaf container. This follows by noting that such a node is not replaceable, and therefore cannot be subject to an update operation.

lookup By invariant (3), only joins can make reachable base nodes non-reachable by making a route node non-reachable. Thus by

invariant (1), a lookup operation always reaches a base node that was reachable and reflects the presence of the item searched for either at the point (i) when the route node that it currently "visits" is spliced out during a join operation (at lines 258–265), or if this does not happen (ii) when it reads the pointer to its destination base node. Thus, one of the above points can serve as linearization point.

range query A range query visits reachable nodes in its scope in increasing item order. For each node, it checks that the node is replaceable, and if so replaces it by a node of type `range_base`, thereby making it irreplaceable (unless the range query has been completed by another thread, in which case such a replacement has no effect). By the limitations on how nodes can be replaced, this guarantees to visit and replace a contiguous sequence of base nodes in the scope of the range query. After completing these replacements, the concerned base nodes are irreplaceable and the operation linearizes, whereafter the base nodes become replaceable.

Let us also show that a join preserves the contents of the tree. A join replaces two reachable (i.e., located in reachable base nodes) leaf containers by a new one. The new leaf container is replacing one of the old leaf containers using a successful CAS (line 254), whereafter the remaining old leaf container is spliced out at lines 258–265. Since the corresponding base nodes are irreplaceable when the replacements happen, and in addition the parent and grandparent of the `join_main` node are marked, no other operation can interfere with invariant (1), viz. that the new joined base node contains the union of the elements in the replaced base nodes.

Let us next consider progress properties of the involved operations. We first note that the lookup operation is wait-free. This follows by observing that it traverses the nodes of the tree without any possibility of being blocked. Progress is achieved at each pointer traversal, as the search space is decreased.

We then note that update operations are lock-free, since they reach the base node to be replaced without being blocked. The update may need to retry, either due to a failed CAS, or because the concerned base node is irreplaceable, because of interference from some other operation. At all such places the operation causing the interference must have made progress. Furthermore, when an operation can not proceed directly due to irreplaceable base nodes, then the operation can always make them replaceable by helping or (in case of a non-secured join) aborting the interfering operation. Range query operations are also lock-free, for analogous reasons.

6 OPTIMIZATION FOR RANGE QUERIES

If possible, it can be advantageous for range queries to avoid writing to shared memory as this induces less cache-coherence traffic. Therefore, we have applied an optimization to our LFCA tree implementation that optimistically tries to perform a range query without writing to shared memory. If this optimistic attempt fails, the range query is performed using the algorithm described in Section 4. The optimistic attempt consists of a test scan and then a validation scan of the base nodes needed for the range query. If nothing has changed between the test and the validation scan, one can be certain that all base nodes in the scans were present at some point and the optimistic attempt can succeed. This scheme is essentially the same as the one described for doing range queries in the k-ary data structure [4]. We refer to that paper for how to prove this scheme correct.

7 EVALUATION

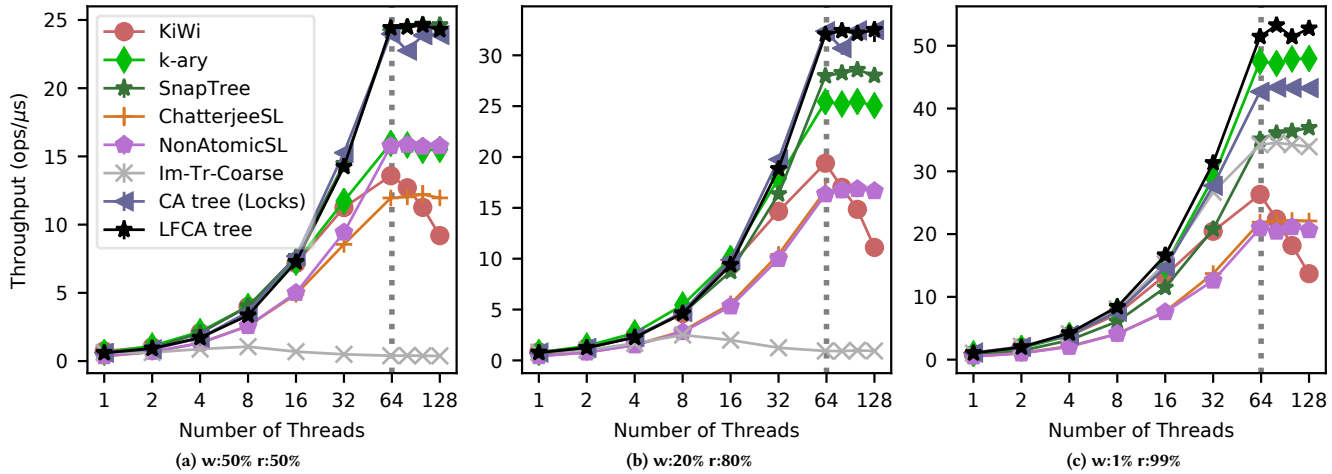
We will now experimentally evaluate LFCA trees. Our implementation uses an immutable treap for the leaf containers and employs the optimization described in Section 6. To facilitate cache friendly range queries, the treap implementation stores all items in fat leaf nodes containing arrays that can store up to 64 items. The LFCA tree is compared to recent proposals for performing linearizable range queries in ordered sets: *SnapTree* [3], k-ary [4], Chatterjee's method applied to a lock-free skiplist [6] (*ChatterjeeSL*) and *KiWi* [2]. We also include the lock-based CA tree [22] in the comparison; it uses the same immutable treap as the LFCA tree in its leaf containers, and is optimized to take advantage of the immutability of the leaf containers so that range queries and lookups do not read the items in the leaf containers while holding locks. Finally, the lock-free ConcurrentSkipListMap from the Java library, which only supports non-linearizable range queries (*NonAtomicSL*), and the coarse-grained data structure (*Imm-Tr-Coarse*) that we described in the introduction are also included. All data structures are in Java as implemented by their respective authors. The maximum number of items in the nodes is set to 64 for k-ary, *Im-Tr-CA* and *SL-CA* as this value has previously been shown to give good results [4]. *KiWi*'s constants are set as described in the *KiWi* paper [2].

The benchmarks were run on a machine with four Intel(R) Xeon(R) E5-4650 CPUs (2.70GHz each with eight cores and hyperthreading, giving a total of 32 actual and 64 logical cores), turbo boost turned off, 128GB of RAM, running Linux 4.9.0-4-amd64 and Oracle JVM 1.8.0_151 (with the JVM flags -Xmx8g -Xms8g -server -d64 -XX:+UseCondCardMark). Each data point comes from the average of three measurements runs of 10 seconds each that were preceded by three warm up runs, also of 10 seconds each, whose purpose is to give the JiT compiler enough time to compile the code.

The keys for the operations lookup, insert and remove as well as the starting keys for range queries are randomly generated integers from a range of size S. The data structure is pre-filled before the start of each benchmark run so that it contains $S/2$ random integers. We use $S = 10^6$ in all experiments presented in this section, which corresponds to a set of size approximately 5×10^5. (Results for $S = 10^5$ and $S = 10^7$ can be found in the extended version of this paper [23].) Range queries calculate the sum of the items in the range and the number of items in the range. As a sanity check, the average number of items that are traversed per range query is calculated and checked against the expected value.

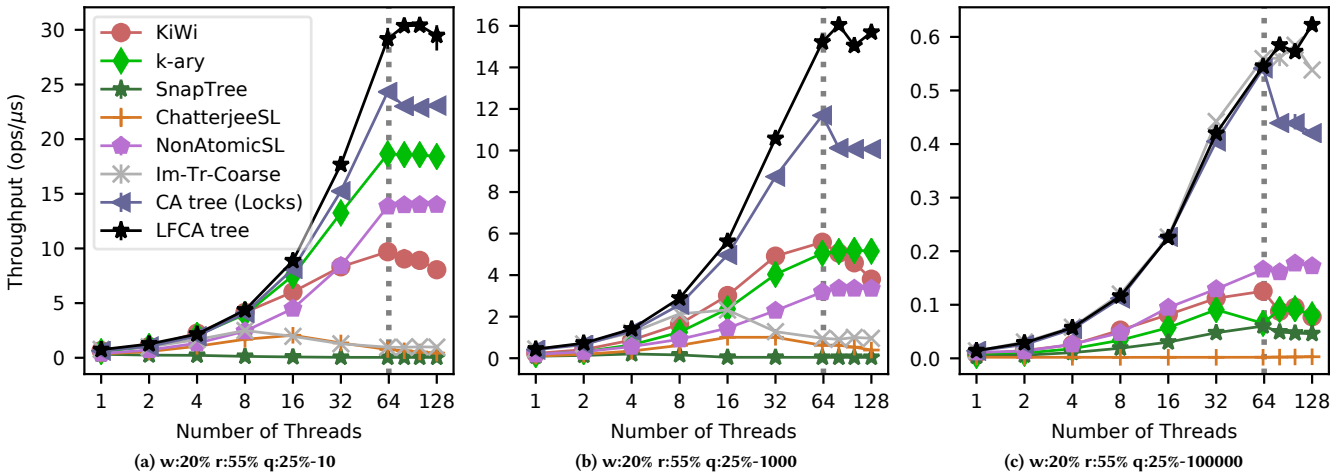
The benchmark scenarios measure throughput of a mix of operations performed by N threads. In figure captions, the scenarios are described by strings of the form w:A% r:B% q:C%-R, meaning that the benchmark performs $(A/2)$% insert, $(A/2)$% remove, B% lookup operations and C% range queries of maximum range size R. The range sizes are randomly set to values between 1 and R.

We start with three scenarios without range queries; cf. Fig. 8. As the machine only has 64 hardware threads, thread counts above 64 show how the data structures perform when there are more threads than hardware can execute in parallel. Both the lock-based and the



(a) w:50% r:50% (b) w:20% r:80% (c) w:1% r:99%

Figure 8: Single-item operations only. Throughput (operations/μs) on the y-axis and thread count on the x-axis. The sub-figures are ordered in increasing amount of lookups. (Note that the SnapTree is hidden behind the CA trees in Fig. 8a).



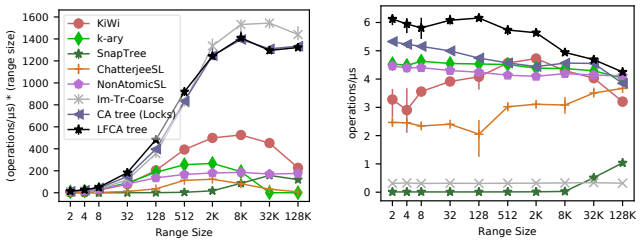
(a) w:20% r:55% q:25%-10 (b) w:20% r:55% q:25%-1000 (c) w:20% r:55% q:25%-100000

Figure 9: Range queries and single-item operations. Throughput (operations/μs) on the y-axis and thread count on the x-axis. The sub-figures are ordered in increasing range query size

lock-free CA tree compare well with the competing data structures both when we have 50% updates and 50% lookups (Fig. 8a) and in the read-heavy scenarios with up to 99% lookups (Figs. 8b and 8c). This shows that the LFCA tree can be a good choice even without range queries. The LFCA tree performs substantially better than the lock-based CA tree at 64 threads and beyond in the read-heavy scenario, which is likely due to LFCA tree's wait-free lookup operation.

Let us now consider scenarios that also contain range queries; cf. Fig. 9. These scenarios show the key strength of the CA trees. Namely, their ability to change the sizes of their immutable parts to fit the workload at hand. Figure 9a shows that the LFCA tree performs better than all the other data structures in a scenario with relatively small range queries of maximum size 10. In the scenario with moderately-sized range queries of maximum size 1000 (Fig. 9b), the LFCA tree outperforms all the other data structures with an even wider margin even though the lock-based CA tree also performs very well there. Data structures clearly benefit from fine-grained synchronization in the scenarios with range queries up to a maximum size of 1000 (e.g., *Im-Tr-Coarse* scales relatively poorly in the scenarios of Figs. 9a and 9b). In contrast, in the scenario with large range queries (Fig. 9c), it seems like the combination of immutable data and coarse-grained synchronization is the best as *Im-Tr-Coarse*'s performance is on par with LFCA trees's performance. Therefore, it seems like the LFCA tree can perform extremely well across a wide variety of workloads (only single-key operations, small range queries and large ones) due to its ability to adapt its structure to fit the workload. The performance drop that can be observed for the lock-based CA tree after 64 threads is probably due to lock-related problems that become more apparent when thread preemption becomes more common (e.g., threads may need to wait for a thread that has got preempted by the OS).

With the benchmark configurations discussed above, the threads spend much more time in range queries than in single-item operations when the range queries are large. Thus, another benchmark is needed to measure the data structures' ability to handle large range queries concurrently with frequent update operations. To



(a) Range queries (parallel updates) (b) Updates (parallel range queries)

Figure 10: On the left, throughput for the 16 range query threads, and on the right, throughput for the threads doing only inserts and removes.

this end, we use a similar benchmark to the one developed by the *KiWi* authors. In this benchmark, half the threads (16) do update operations (insert and remove with equal probability) while the other half (also 16) do range queries with a range of fixed size. In the results for the benchmarks, the throughput for updates (Fig. 10b) is presented separately from the range query throughput (Fig. 10a) so that one can study the performance of these operations separately. Note that in the graphs that show the range query throughput, the number of operations per μs is shown multiplied by the range query size on the y-axis to make the graphs more readable. The range query size used in the experiment is shown on the x-axis.

Taking both the performance of the range queries and updates into account, it is clear that the LFCA tree is the best data structure for all range query sizes. It is tightly followed by the lock-based CA tree. Looking only at the throughput for the update operations, *k*-ary, *NonAtomicSL* and *KiWi* can also keep up quite well (at least with some range query sizes). However, when also taking range queries into account, these data structures are all far behind LFCA tree for all range query sizes above 128. In short, *k*-ary, *NonAtomicSL* and *KiWi* can perform well only in a few scenarios that fit their synchronization granularity and the sizes of their immutable parts, while the CA trees can perform well in a wider range of scenarios thanks to their ability to adapt.

We now take a look at the statistics shown in Tables 1 and 2. They show the base node count (measured after the experiments), the average number of traversed base nodes per range query and the number of splits and joins per millisecond for the scenarios that are shown in Figs. 9b and 10, respectively. These statistics indicate that the heuristics works as intended. That is, larger range queries result in fewer base nodes and more threads result in more base nodes. Looking at the number of base nodes traversed per range query, it is also clear that range queries spend a relatively short time traversing shared mutable data (compared to the non-adaptive data structures in the comparison) even for large range queries. This explains how the LFCA tree can perform so much better than the non-adaptive data structures which is compared against.

The five parts of Fig. 11 show results from a time series experiment that was run in order to illustrate into how an LFCA tree adapts its structure when the workload suddenly changes and how this adaptation affects its performance. The top part of the figure shows the number of route nodes, and the bottom part the throughput which is achieved at different time points. At time zero, the LFCA tree contains only one base node with 500*K* items. The experiment begins with the workload w:20% r:55% q:25%-1000, which

Table 1: Statistics for the LFCA tree in the scenarios of Fig. 9b (w:20% r:55% q:25%-1000).

Threads	1	2	4	8	16	32	64
# route nodes	0	53	130	270	440	740	980
$\frac{\#\text{ traversed base nodes}}{\#\text{ range queries}}$	1.0	1.0	1.0	1.0	1.0	1.1	1.1
$\frac{\#\text{ splits}}{\text{milliseconds}}$	0	0.024	0.048	0.1	0.23	0.53	1.0
$\frac{\#\text{ joins}}{\text{milliseconds}}$	0	0.019	0.036	0.078	0.19	0.46	0.91

Table 2: Statistics for the LFCA tree in the scenarios of Fig. 10. 16 threads doing range queries and 16 threads doing updates.

Range Size	2	128	512	2K	8K	32K	128K
# route nodes	2.9K	2.3K	2.1K	1.3K	680	370	330
$\frac{\#\text{ traversed base nodes}}{\#\text{ range queries}}$	1.0	1.0	1.3	2.6	5.5	12	42
$\frac{\#\text{ splits}}{\text{milliseconds}}$	1.1	1.3	1.5	2.4	5.0	10	13
$\frac{\#\text{ joins}}{\text{milliseconds}}$	0.79	1.0	1.3	2.3	4.9	10	13

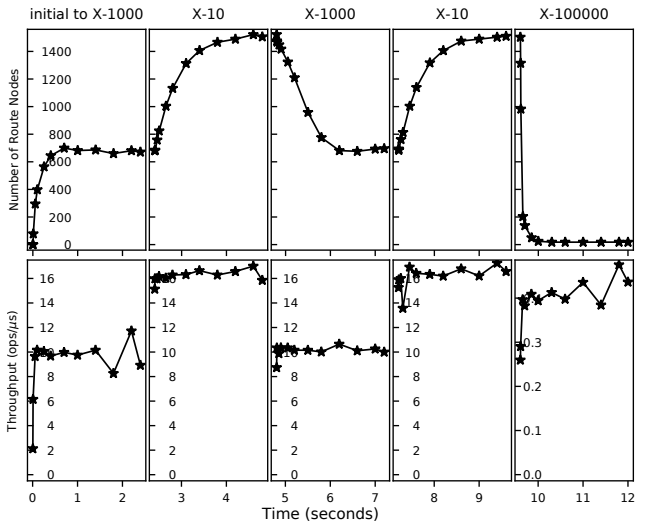


Figure 11: Time series illustrating sudden changes in the workloads. Number of route nodes in the top and throughput in the bottom. Only the maximum size of the range queries changes between the figures. X = 20% r:55% q:25%.

is executed using 30 threads for 2.4 seconds. Every 2.4 seconds the maximum range query size changes: besides the initial value (1000) it also takes the values 10, 1000, 10 and 100000 (cf. the values at the top line of Fig. 11). From the time series, we can see that, after each workload change, the rate of change for the number of route nodes gradually decreases until the number of route nodes stabilizes around a certain value. In the parts titled "initial to X-1000" and "X-100000", one can also see a positive change in throughput when the number of route nodes increases/decreases quickly. The change in throughput while the number of route nodes changes quickly is not as big in the three middle parts ("X-10", "X-1000" and "X-10"), which is not strange considering that the synchronization granularity changes relatively less in these parts.

The benchmark set up for the time series experiment is a bit involved in order to obtain numbers that are not disturbed by taking measurements during very short periods of time. It goes as follows. For every time point shown with a dot in the graphs, average measurements from five experiment runs in different JVM instances were collected. Each such run consists of 35 warm up runs and 10 measurement runs (that we take the average measurements from). A warm up or a measurement run does the following after the LFCA tree has been filled with $500K$ items: It performs a triggering run (skipped in the first workload of the time series) of 2.2 seconds that applies the workload W (where W is the previous workload in the time series), before the actual warm up/measurement run is executed for t seconds (t is always 2 seconds for the warm up runs), after which the number of route nodes and the number of performed operations by the threads are collected. For example, a run to collect measurements for time point 3.4 seconds performs a triggering run with the workload w:20% r:55% q:25%-1000 and then a measurement run with the workload w:20% r:55% q:25%-10 (running for 1 second). The throughput for a time point t_n is calculated as $\frac{o(t_n)-o(t_{n-1})}{(t_n-t_{n-1})}$, where t_{n-1} denotes the previous time point in the time series and $o(t)$ denotes the average number of performed operations measured for time point t.

8 CONCLUDING REMARKS

We have given a detailed description and correctness arguments for the LFCA tree, the first lock-free data structure supporting range queries that adapts its structure based on heuristics that take detected contention and information about range queries into account. LFCA trees make use of information gathered at runtime to get a good trade-off between the performance of operations that generally benefit from coarse-grained synchronization and those that generally benefit from fine-grained synchronization. Our experimental evaluation, in the previous section as well as in its extended version of this paper [23], shows that this has real benefits in practice, as the LFCA tree can maintain exceptionally good performance across a wide range of scenarios.

DATA AVAILABILITY STATEMENT

The source code for the LFCA tree as well as the code for the benchmarks are available online [18].

ACKNOWLEDGMENTS

This work was carried out within the Linnaeus centre of excellence UPMARC (Uppsala Programming for Multicore Architectures Research Center), and was partly supported by grants from the Swedish Research Council.

REFERENCES

[1] Hillel Avni, Nir Shavit, and Adi Suissa. 2013. Leaplist: Lessons Learned in Designing TM-supported Range Queries. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing (PODC '13)*. ACM, New York, NY, USA, 299–308. https://doi.org/10.1145/2484239.2484254

[2] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. 2017. KiWi: A Key-Value Map for Scalable Real-Time Analytics. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM, New York, NY, USA, 357–369. https://doi.org/10.1145/3018743.3018761

[3] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*. ACM, New York, NY, USA, 257–268. https://doi.org/10.1145/1693453.1693488

[4] Trevor Brown and Hillel Avni. 2012. Range Queries in Non-blocking k-ary Search Trees. In *Principles of Distributed Systems: 16th International Conference, OPODIS 2012. Proceedings*, Roberto Baldoni, Paola Flocchini, and Ravindran Binoy (Eds.). Springer, 31–45. https://doi.org/10.1007/978-3-642-35476-2_3

[5] Trevor Brown and Joanna Helga. 2011. Non-blocking k-ary Search Trees. In *Principles of Distributed Systems: 15th International Conference, OPODIS 2011. Proceedings*, Antonio Fernàndez Anta, Giuseppe Lipari, and Matthieu Roy (Eds.). Springer, 207–221. https://doi.org/10.1007/978-3-642-25873-2_15

[6] Bapi Chatterjee. 2017. Lock-free Linearizable 1-Dimensional Range Queries. In *Proceedings of the 18th International Conference on Distributed Computing and Networking (ICDCN '17)*. ACM, New York, NY, USA, Article 9, 10 pages. https://doi.org/10.1145/3007748.3007771

[7] Chao-Hong Chen, Vikraman Choudhury, and Ryan R. Newton. 2017. Adaptive Lock-free Data Structures in Haskell: A General Method for Concurrent Implementation Swapping. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. ACM, New York, NY, USA, 197–211. https://doi.org/10.1145/3122955.3122973

[8] Phuong Hoai Ha, Marina Papatriantafilou, and Philippas Tsigas. 2007. Self-tuning reactive diffracting trees. *J. Parallel and Distrib. Comput.* 67, 6 (2007), 674–694. https://doi.org/10.1016/j.jpdc.2007.01.011

[9] M. Herlihy. 1990. A Methodology for Implementing Highly Concurrent Data Structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP '90)*. ACM, New York, NY, USA, 197–206. https://doi.org/10.1145/99163.99185

[10] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

[11] Ryan R. Newton, Peter P. Fogg, and Ali Varamesh. 2015. Adaptive Lock-free Maps: Purely-functional to Scalable. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 218–229. https://doi.org/10.1145/2784731.2784734

[12] Chris Okasaki. 1999. *Purely functional data structures*. Cambridge Univ. Press.

[13] Erik Österlund and Welf Löwe. 2014. Concurrent Transformation Components Using Contention Context Sensors. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 223–234. https://doi.org/10.1145/2642937.2642995

[14] Erez Petrank and Shahar Timnat. 2013. Lock-Free Data-Structure Iterators. In *Proceedings of the 27th International Symposium on Distributed Computing - Volume 8205 (DISC 2013)*. Springer-Verlag New York, Inc., New York, NY, USA, 224–238. https://doi.org/10.1007/978-3-642-41527-2_16

[15] Callum Robertson. 2014. *Implementing Contention-Friendly Range Queries in Non-Blocking Key-Value Stores*. Bachelor Thesis. The University of Sydney.

[16] Konstantinos Sagonas and Kjell Winblad. 2016. Efficient Support for Range Queries and Range Updates Using Contention Adapting Search Trees. In *Languages and Compilers for Parallel Computing - 28th International Workshop, LCPC (LNCS)*, Xipeng Shen, Frank Mueller, and James Tuck (Eds.), Vol. 9519. Springer, 37–53. https://doi.org/10.1007/978-3-319-29778-1_3

[17] Konstantinos Sagonas and Kjell Winblad. 2017. A contention adapting approach to concurrent ordered sets. *J. Parallel and Distrib. Comput.* (2017). https://doi.org/10.1016/j.jpdc.2017.11.007

[18] Konstantinos Sagonas and Kjell Winblad. 2018. Contention Adapting Search Trees. (2018). http://www.it.uu.se/research/group/languages/software/ca_tree.

[19] R. Seidel and C. R. Aragon. 1996. Randomized search trees. *Algorithmica* 16, 4 (01 Oct. 1996), 464–497. https://doi.org/10.1007/BF01940876

[20] N. Shafiei. 2013. Non-blocking Patricia Tries with Replace Operations. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*. 216–225. https://doi.org/10.1109/ICDCS.2013.43

[21] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. 2013. F1: A Distributed SQL Database That Scales. *Proceedings of the VLDB Endowment* 6, 11 (Aug. 2013), 1068–1079. https://doi.org/10.14778/2536222.2536232

[22] Kjell Winblad. 2017. Faster Concurrent Range Queries with Contention Adapting Search Trees Using Immutable Data. In *2017 Imperial College Computing Student Workshop (ICCSW 2017) (OpenAccess Series in Informatics (OASIcs))*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

[23] Kjell Winblad, Konstantinos Sagonas, and Bengt Jonsson. 2018. Lock-free Contention Adapting Search Trees. (2018). Extended version of this paper available at http://www.it.uu.se/research/group/languages/software/ca_tree.

[24] Yahoo! Developer Network. 2017. Flurry analytics. (2017). https://developer.yahoo.com/flurry/docs/analytics/ Accessed: 2017-07-26.