Efficient Support for Range Queries and Range Updates Using Contention Adapting Search Trees*

Konstantinos Sagonas and Kjell Winblad

Department of Information Technology, Uppsala University, Sweden

Abstract. We extend contention adapting trees (CA trees), a family of concurrent data structures for ordered sets, to support linearizable range queries, range updates, and operations that atomically operate on multiple keys such as bulk insertions and deletions. CA trees differ from related concurrent data structures by adapting themselves according to the contention level and the access patterns to scale well in a multitude of scenarios. Variants of CA trees with different performance characteristics can be derived by changing their sequential component. We experimentally compare CA trees to state-of-the-art concurrent data structures and show that CA trees beat the best data structures we compare against with up to 57% in scenarios that contain basic set operations and range queries, and outperform them by more than 1200% in scenarios that also contain range updates.

1 Introduction

Data intensive applications on multicores need efficient and scalable concurrent data structures. Many concurrent data structures for ordered sets have recently been proposed (e.g [2, 4, 8, 11]) that scale well on workloads containing *single key operations*, e.g. insert, remove and get. However, most of these data structures lack efficient and scalable support for operations that atomically access multiple elements, such as range queries, range updates, bulk insert and remove, which are important for various applications such as in-memory databases. Operations that operate on a single element and those that operate on multiple ones have inherently conflicting requirements. The former achieve better scalability by using fine-grained synchronization, while the latter are better off performance-wise if they employ coarse-grained synchronization. The few data structures with scalable support for some multi-element operations [1, 3] have to be parameterized with the granularity of synchronization. Setting this parameter is inherently difficult since the usage patterns and contention level are sometimes impossible to predict. This is especially true when the data structure is provided as a general purpose library.

Contention adapting trees (*CA trees*) [18] is a new family of concurrent data structures for ordered sets, that adapt their synchronization granularity according to the contention level and the access patterns even when these change dynamically. In this work, we extend CA trees with support for operations that atomically access multiple elements. As we will see, CA trees provide good scalability both in contended and

* Research supported in part by the European Union grant IST-2011-287510 "RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software" and the Linnaeus centre of excellence UPMARC (Uppsala Programming for Multicore Architectures Research Center).

uncontended situations. Moreover they are flexible: CA tree variants with different performance characteristics can be derived by selecting their underlying sequential data structure component. CA trees support the common interfaces of sets, maps and key-value stores as well as range queries, range updates, bulk inserts, bulk removes and other operations that atomically access multiple keys. Experiments on scenarios with a variety of mixes of these operations show that CA trees provide performance that is significantly better than that obtained by state-of-the-art data structures for ordered sets and range queries. All these make CA trees suitable for a multitude of applications, including in-memory databases, key-value stores and general purpose data structure libraries.

Definitions. A *range query* operation atomically takes a snapshot of all elements belonging to a range $[a, b]$ of keys. A *range update* atomically applies an update function to all values associated with keys in a specific key range. A *bulk insert* atomically inserts all elements in a list of keys or key-value pairs. (A *bulk remove* is defined similarly.) We call operations that operate on a range of elements *range operations* and use *multi-element operations* as a general term for operations that atomically access multiple elements.

Overview. We start by reviewing related work (Section 2) before we introduce the CA trees in detail (Section 3) and compare them experimentally to related data structures (Section 4). The paper ends with some discussion and concluding remarks (Section 5).

2 Related Work

In principle, concurrent ordered sets with linearizable range operations can be implemented by utilizing software transactional memory (TM): the programmer simply wraps the operations in transactions and lets the TM take care of the concurrency control to ensure that the transactions execute atomically. Even though some scalable data structures have been derived by carefully limiting the size of transactions (e.g. [1, 7]), currently transactional memory does not offer a general solution with good scalability; cf. [1].

Brown and Helga have extended the *non-blocking k-ary search tree* [4] to provide lock-free range queries [3]. A k-ary search tree is a search tree where all nodes, both internal and leaves, contain up to k keys. The internal nodes are utilized for searching, and leaf nodes contain all the elements. Range queries are performed in k-ary search trees with immutable leaf nodes by using a scan and a validate step. The scan step scans all leaves containing keys in the range and the validate step checks a dirty bit that is set before a leaf node is replaced by a modifying operation. Range queries are retried if the validation step fails. Unfortunately, non-blocking k-ary search trees provide no efficient way to perform atomic range updates or multi-element modification operations. Additionally, k-ary search trees are not balanced, so pathological inputs can easily make them perform poorly. Robertson investigated the implementation of lock-free range queries in a skip list: range queries increment a version number and a fixed size history of changes is kept in every node [15]. This solution does not scale well because of the centralized version number counter. Also, it does not support range updates.

Functional data structures or copy-on-write is another approach to provide linearizable range queries. Unfortunately, this requires copying all nodes in a path to the root in a tree data structure which induces overhead and makes the root a contended hot spot.

The *Snap tree* data structure [2] provides a fast $\mathcal{O}(1)$ linearizable clone operation by letting subsequent write operations create a new version of the tree. Linearizable range queries can be performed in a Snap tree by first creating a clone and then performing the query in the clone. Snap's clone operation is performed by marking the root node as shared and letting subsequent update operations replace shared nodes while traversing the tree. To ensure that no existing update operation can modify the clone, an epoch object is used. The clone operation forces new updates to wait for a new epoch object by closing the current epoch and then waits for existing modifying operations (that have registered their ongoing operation in the epoch object) before a new epoch object is installed. The *Ctrie* data structure [13] also has a fast clone operation whose implementation and performance characteristics resembles Snap; see [3].

Range operations can be implemented in data structures that utilize fine-grained locking by acquiring all necessary locks. For example, in a tree data structure where all elements reside in leaf nodes, the atomicity of the range operation can be ensured by locking all leaves in the range. This requires locking at least n/k nodes, if the number of elements in the range is n and at most k elements can be stored in every node. When n is large or k is small the performance of this approach is limited by the locking overhead. On the other hand, when n is small or k is large the scalability is limited by coarse-grained locking. In contrast, as we will see, in CA trees k is dynamic and adapted at runtime to provide a good trade-off between scalability and locking overhead.

The *Leaplist* [1] is a concurrent ordered set implementation with native support for range operations. Leaplist is based on a skip list data structure with fat nodes that can contain up to k elements. The efficient implementation of the Leaplist uses transactional memory to acquire locks and to check if read data is valid. The authors of the Leaplist mention that they tried to derive a Leaplist version based purely on fine-grained locking but failed [1], so developing a Leaplist without dependence on STM seems to be difficult. As in trees with fine-grained locking, the size of the locked regions in Leaplists is fixed and does not adapt according to the contention as in CA trees. Furthermore, the performance of CA trees does not depend on the availability and performance of STM.

Operations that atomically operate on multiple keys can be implemented in any data structure by utilizing coarse-grained locking. By using a readers-writer lock, one can avoid acquiring an exclusive lock of the data structure for some operations. Unfortunately, locking the whole data structure is detrimental to scalability if the data structure is contended. The advantage of coarse-grained locking is that it provides the performance of the protected sequential data structure in the uncontended case. As we will soon see, CA trees provide the high performance of coarse-grained locking in the uncontended cases and the scalability of fine-grained synchronization in contended ones by adapting their granularity of synchronization according to the contention level.

3 Contention Adapting Search Trees

The structure and components of CA trees are as follows. The elements (key-value pairs or keys) contained in a CA tree are stored in sequential ordered set data structures (e.g., AVL trees, skip lists, etc.) which are rooted by *base nodes*. Each base node contains a *lock* that maintains statistics about the current level of the node's contention. The synchronization of accesses to a particular base node is handled independently of all other base nodes.

Base nodes are linked together by *routing nodes* as depicted in Fig. 1. The routing nodes do not contain elements; instead they contain keys which are only used to facilitate searching. As in ordinary binary search trees, all elements contained in the left branch of a routing node with key K have keys smaller than K and all elements contained in the right branch have keys greater than or equal to K. When it is detected that contention on a particular base node B is high, the subtree rooted by B is *split* to reduce the contention. Symmetrically, if contention on a base node B is detected to be low, B is *joined* with a neighbor base node to reduce the search path and to make atomic access of larger parts of the CA tree more efficient. An example of a split and a join operation is shown in Fig. 2.

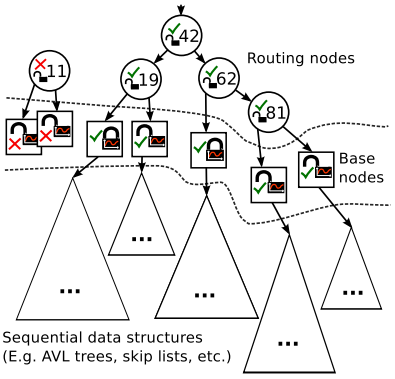
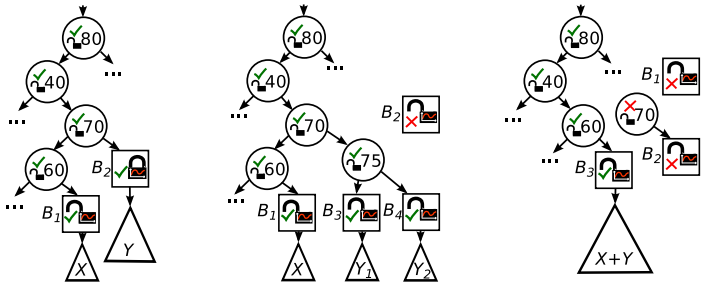


Fig. 1: The structure of a CA tree. Numbers denote keys, a node whose flag is valid is marked with a green hook; an invalid one with a red cross.

Contention detection is done by simply checking whether waiting for the lock of a base node was required or not, and increasing or decreasing the statistics counter (which is located in the base node lock) accordingly. Thresholds for this counter are used to decide when adaptation shall be performed. A good heuristic is to do adaptation for high contention eagerly and adaptation for low contention only when the contention has been low for many operations. This heuristics also avoids too frequent adaptations back and forth [18]. This mechanism for contention detection has low overhead and works well in practice. Still, other mechanisms can be used, e.g., based on the back-off time in an exponential back-off spin lock [12].

Searching in the routing node layer is done without acquiring any locks. However, as seen in Fig. 1, besides a key, routing nodes also have a valid flag (✓or ✗) and a lock. These are used to synchronize between concurrent join operations (i.e., adaptations for low contention). Since, as explained above, join operations happen relatively infrequently in CA trees, the locks in the routing nodes do not limit scalability in practice.

Single-key Modification Operations. Operations such as insert and remove start from the root of the CA tree and search for the base node B under which the element/key that is given as parameter to the operation will be inserted or removed. Recall that the traversal of the routing nodes does not acquire any locks. When B is reached, it is locked and then its valid flag is checked. If this flag is false (✗), the search needs to be retried. A base node becomes invalid when it is replaced by a split or a join. A search that ends up in an invalid base node thus needs to be retried until a valid base node is found. When this has happened, the operation is simply forwarded to the sequential data structure rooted by the base node. Before the base node is unlocked and the operation completes, we check if enough contention or lack of contention has been detected to justify an adaptation. If high contention is detected, the elements in the base node are split into two new base nodes that are linked together by a routing node. Figures 2a and 2b show CA trees before and after base node B_2 and the data structure Y is split (75 is the split key). In the reverse direction, if low contention is detected, the sequential data structure of the



(a) Initial CA tree (b) CA tree after a split (c) CA tree after a join

Fig. 2: Effect of the split and join operations on the CA tree of Fig. 2a.

base node B is joined with that of a neighbor base node and the parent routing node of B is spliced out together with B. Figures 2a and 2c show CA tree structures before and after base node B_2 is spliced out from the tree and the elements of its Y structure are joined with those of X. We refer to [18] for pseudocode and a detailed description of the algorithms for splitting and joining base nodes and single key operations.

Single-key Read-only Operations. Read-only operations like get, contains, findMax, etc. can work in a similar fashion as modification operations. However, on a multicore system, acquiring even a RW lock in read mode for read-only operations can cause bad scalability due to increased cache coherence traffic. Therefore, the performance and scalability of read-only operations can be improved if acquiring a lock can be avoided. By using a *sequence lock* [10] in the base nodes, read-only operations can attempt to perform the operation optimistically by checking the sequence number in the lock before and after the read-only operation has been performed on the base node. If the optimistic attempt fails, the base node lock can be acquired non-optimistically. This sequence lock optimization avoids writing to shared memory in the common case when the base node is not contended, which greatly improves performance in practice [18]. The concurrency in the data structure can be further improved by using a sequence lock with support for concurrent execution of read-only critical sections. By using such a lock, one can acquire the base node lock in read-only mode when the optimistic read attempt fails, and thus allowing concurrent reads to read from the base node at the same time. Note that an optimistic read does not change the statistics counter, because that would involve writing to shared memory and would defeat the purpose of having such operations. If the optimistic read fails and the lock is acquired in read mode, our implementation adds to the contention statistics to decrease the likelihood of optimistic read failures in the future[1].

Multi-element Operations. CA trees also support operations that atomically operate on several keys, such as bulk insert, bulk remove, and swap operations that swap the values associated with two keys. Generic pseudocode for such operations appears in Fig. 4a; its

[1] We perform the change to the contention statistics counter non-atomically. Thus, it is possible for a concurrent read operation to overwrite the change. Note that this does not effect the correctness of the data structure as it only affects the frequency of its adaptations.

helper function `manageCont` appears in Fig. 3a. Such operations start by sorting the elements given as their parameter (line 7). Then all the base nodes needed for the operations are found (line 12) and locked (lines 15–16) in sorted order. Locking base nodes in a specific order prevents deadlocks. The method `lockIsContended` in the base node, locks the base node lock and return `true` if contention was detected while locking it and the method `lockNoStats` locks the base node lock without recording any contention. When multi-element operations are given keys that all reside in one base node, only this base node needs to be locked. One simply has to query the sequential data structure in the current base node for the maximum key (line 26) to see which of the given elements must belong to a base node. This can be compared to data structures that utilize non-adaptive fine-grained synchronization and thus either need to lock the whole data structure or all involved nodes individually. Finally, multi-key operations end by adjusting the contention statistics, unlock all acquired locks, and split or join one of the base nodes (lines 35–46) if required.

```
1  void manageCont(BaseNode base, boolean contended) {
2    if (contended) base.lock.statistics += FAIL_CONTRIB;
3    else base.lock.statistics -= SUCC_CONTRIB;
4    if (base.lock.statistics > MAX_CONTENTION) {
5      if (size(base.root) < 2) base.lock.statistics = 0;
6      else highContentionSplit(tree, base, base.parent);
7    } else if (base.lock.statistics < MIN_CONTENTION) {
8      if (base.parent == null) base.lock.statistics = 0;
9      else lowContentionJoin(tree, base, base.parent);
10   }
11 }
```

(a) Manage contention

```
1  BaseNode, List<RouteNode>
2  getNextBaseNodeAndPath(BaseNode b, List<RouteNode> p) {
3    List<RouteNode> newPathPart;
4    BaseNode bRet;
5    if (p.isEmpty()) { // The parent of b is the root
6      return null, null;
7    } else {
8      List<RouteNode> rp = p.reverse();
9      if (rp.head().left == b) {
10       bRet, newPathPart =
11         leftmostBaseNodeAndPath(rp.head().right);
12       return bRet, p.append(newPathPart);
13     } else {
14       K pKey = rp.head().key; // pKey = key of parent
15       rp.removeFirst();
16       while (rp.notEmpty()) {
17         if (rp.head().isValid() && pKey < rp.head().key) {
18           bRet, newPathPart =
19             leftmostBaseNodeAndPath(rp.head().right);
20           return bRet, rp.reverse().append(newPathPart);
21         } else {
22           rp.removeFirst();
23         }
24       }
25     }
26     return null, null;
27   }
28 }
```

(b) Find next base node

Fig. 3: Helper functions for Fig. 4.

Range Operations. We will now describe an algorithm for atomic range operations that locks all base nodes that can contain keys in the range $[a, b]$. Generic pseudocode for such operations can be seen in Fig. 4b and its helper function `manageCont` and `getNextBaseNodeAndPath` can be seen in Fig. 3. To prevent deadlocks, the base nodes are always locked in increasing order of the keys that they can contain. Therefore, the first base node to lock is the one that can contain the smallest key a in the range. This first base node can be found (line 5 in Fig. 4b) and locked (line 6) using the algorithm described for single-key operations [18]. Finding the next base node (line 21 in Fig. 4b) is not as simple as it might first seem, since routing nodes can be spliced out and base nodes can be split. The two problematic cases that may occur are illustrated in Fig. 2. Suppose that the base node marked B_1 has been found through the search path with routing nodes with keys $80, 40, 70, 60$ as depicted in Fig. 2a. If the tree stays as depicted in Fig. 2a, the base node B_2 would be the next base node. However, B_2 may have been spliced out while the range operation was traversing the routing nodes (Fig. 2c) or split (Fig. 2b). If one of these cases happens, we will detect this since we will end up in an invalid base node in which case the attempt to find the next base node will be retried. When we find the next base node we will *not* end up in the same invalid base node twice if the following algorithm is applied (also depicted in Fig. 3b):

1. If the last locked base node is the left child of its parent routing node P then find the leftmost base node in the right child of P (Fig. 3b, line 11).

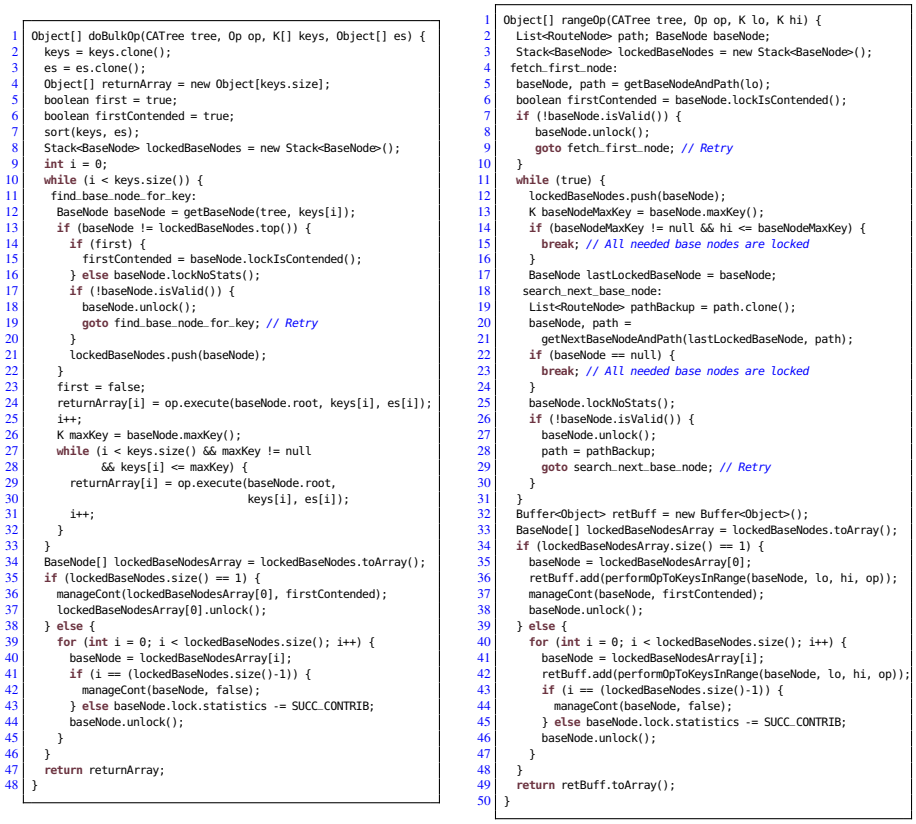
2. Otherwise, follow the reverse search path from P until a *valid* routing node R with a key greater than the key of P is found (Fig. 3b, line 17). If such an R is not found, the current base node is the rightmost base node in the tree so all required base nodes are already locked (Fig. 3b, lines 6 and 26). Otherwise, find the leftmost base node in the right branch of R (Fig. 3b, line 19).

The argument why this algorithm is correct is briefly as follows. For case 1, note that the parent of a base node is guaranteed to stay the same while the base node is valid; cf. also [16]. For case 2, note that once we have locked a valid base node we know that no routing nodes can be added to the search path that was used to find the base node, since the base node in the top of the path must be locked for a new routing node to be linked in. Also, the above algorithm never ends up in the same invalid base node more than once since the effect of a split or a join is visible after the involved base nodes have been unlocked. Finally, if the algorithm ever finds a base node B_2 that is locked and valid and the previously locked base node is B_1, then there cannot be any other base node B' containing keys between the maximum key of B_1 and the minimum key of B_2. This is true because if a split or a join had created such a B', then B_2 would not be valid.

An Optimistic Read Optimization for Range Queries. For the same reasons, as discussed previously for single-key read-only operations, it can be advantageous to perform range queries without writing to shared memory. This can be done by first reading the sequence numbers (in the locks) and validating the base nodes containing the elements in the range. This optimistic attempt is aborted if a sequence number indicates that a write operation is currently changing the content of the base node. After acquiring sequence numbers for all involved base nodes, the range query is continued by reading all elements in the range, checking the sequence number again after the elements in a base node have been read. If the sequence numbers have not changed from the initial scan to after the elements have been read, then one can be sure that no write has interfered with the operation. Thus, the range query will appear to be atomic. As soon as a validation of a sequence number fails or inconsistent state is detected in the sequential data structure, the optimistic attempt will abort. Range queries for which the optimistic attempt failed are performed by acquiring the base node locks belonging to the range in read mode.

Contention Statistics in Multi-element Operations. A multi-element operation performed by non-optimistic locking that only requires one base node changes the contention statistics counter in the same way as single-element operations and also uses the same split and join thresholds as single-element operations. The pseudocode that handles contention in this case can be found in Fig. 3a and is called from line 36 in Fig. 4a and line 37 in Fig. 4b. When contention is detected, the contention statistics counter in that base node is increased (line 2) to make a base node split more likely and otherwise the contention statistics counter is decreased (line 3) to make a base node join more likely. Lines 4 to 10 check if one of the thresholds for adaptation has been reached and performs the appropriate adaptation in that case.

If a multi-element operation performed by non-optimistic locking requires more than one base node, the contention statistics counter is decreased (lines 42–43 in Fig. 4a and lines 44–45 in Fig. 4b) in all involved base nodes to reduce the chance that future



```
1  Object[] doBulkOp(CATree tree, Op op, K[] keys, Object[] es) {
2    keys = keys.clone();
3    es = es.clone();
4    Object[] returnArray = new Object[keys.size];
5    boolean first = true;
6    boolean firstContended = true;
7    sort(keys, es);
8    Stack<BaseNode> lockedBaseNodes = new Stack<BaseNode>();
9    int i = 0;
10   while (i < keys.size()) {
11    find_base_node_for_key:
12     BaseNode baseNode = getBaseNode(tree, keys[i]);
13     if (baseNode != lockedBaseNodes.top()) {
14       if (first) {
15         firstContended = baseNode.lockIsContended();
16       } else baseNode.lockNoStats();
17       if (!baseNode.isValid()) {
18         baseNode.unlock();
19         goto find_base_node_for_key; // Retry
20       }
21       lockedBaseNodes.push(baseNode);
22     }
23     first = false;
24     returnArray[i] = op.execute(baseNode.root, keys[i], es[i]);
25     i++;
26     K maxKey = baseNode.maxKey();
27     while (i < keys.size() && maxKey != null
28            && keys[i] <= maxKey) {
29       returnArray[i] = op.execute(baseNode.root,
30                                   keys[i], es[i]);
31       i++;
32     }
33   }
34   BaseNode[] lockedBaseNodesArray = lockedBaseNodes.toArray();
35   if (lockedBaseNodes.size() == 1) {
36     manageCont(lockedBaseNodesArray[0], firstContended);
37     lockedBaseNodesArray[0].unlock();
38   } else {
39     for (int i = 0; i < lockedBaseNodes.size(); i++) {
40       baseNode = lockedBaseNodesArray[i];
41       if (i == (lockedBaseNodes.size()-1)) {
42         manageCont(baseNode, false);
43       } else baseNode.lock.statistics -= SUCC_CONTRIB;
44       baseNode.unlock();
45     }
46   }
47   return returnArray;
48 }
```

(a) Bulk operations

```
1  Object[] rangeOp(CATree tree, Op op, K lo, K hi) {
2    List<RouteNode> path; BaseNode baseNode;
3    Stack<BaseNode> lockedBaseNodes = new Stack<BaseNode>();
4   fetch_first_node:
5    baseNode, path = getBaseNodeAndPath(lo);
6    boolean firstContended = baseNode.lockIsContended();
7    if (!baseNode.isValid()) {
8       baseNode.unlock();
9       goto fetch_first_node; // Retry
10   }
11   while (true) {
12     lockedBaseNodes.push(baseNode);
13     K baseNodeMaxKey = baseNode.maxKey();
14     if (baseNodeMaxKey != null && hi <= baseNodeMaxKey) {
15       break; // All needed base nodes are locked
16     }
17     BaseNode lastLockedBaseNode = baseNode;
18    search_next_base_node:
19     List<RouteNode> pathBackup = path.clone();
20     baseNode, path =
21       getNextBaseNodeAndPath(lastLockedBaseNode, path);
22     if (baseNode == null) {
23       break; // All needed base nodes are locked
24     }
25     baseNode.lockNoStats();
26     if (!baseNode.isValid()) {
27       baseNode.unlock();
28       path = pathBackup;
29       goto search_next_base_node; // Retry
30     }
31   }
32   Buffer<Object> retBuff = new Buffer<Object>();
33   BaseNode[] lockedBaseNodesArray = lockedBaseNodes.toArray();
34   if (lockedBaseNodesArray.size() == 1) {
35     baseNode = lockedBaseNodesArray[0];
36     retBuff.add(performOpToKeysInRange(baseNode, lo, hi, op));
37     manageCont(baseNode, firstContended);
38     baseNode.unlock();
39   } else {
40     for (int i = 0; i < lockedBaseNodes.size(); i++) {
41       baseNode = lockedBaseNodesArray[i];
42       retBuff.add(performOpToKeysInRange(baseNode, lo, hi, op));
43       if (i == (lockedBaseNodes.size()-1)) {
44         manageCont(baseNode, false);
45       } else baseNode.lock.statistics -= SUCC_CONTRIB;
46       baseNode.unlock();
47     }
48   }
49   return retBuff.toArray();
50 }
```

(b) Range operations

Fig. 4: Pseudocode for bulk operations and range operations.

multi-element operations will require more than one base node. Before unlocking the last base node, low-contention join or high-contention split is performed on that base node if the thresholds are reached (line 42 in Fig. 4a and line 44 in Fig. 4b).

Range operations where the optimistic attempt succeeds do not change the contention statistics of any of the base nodes that they use. Doing so would defend the purpose of the optimistic attempt which is to avoid writing to shared state. However, if the optimistic attempt fails, the contention statistics is updated as described before.

Correctness. In a previous publication [18] we provided proofs for that the algorithm for single-key operations is deadlock free, livelock free as well as a proof sketch for its linearizability. Here, we will briefly repeat the outlines of the proofs for single-key operations and provide a proof sketch that the properties deadlock freedom, livelock freedom and linearizability are all provided by CA trees when extended with the range operations and bulk operations that we have described in detail in this paper. The interested reader can find more detailed proofs in a technical report available online [16].

Deadlock freedom can be shown by proving that all locks are acquired in a specific order. All single-key operations (except operations that perform a low-contention join) acquire a single lock; cf. [16]. Low-contention join can acquire base node locks in different orders but since this is done with a non-blocking try lock call and all locks that the operation is holding are released if the try lock call fails, this cannot cause a deadlock. The proof for deadlock freedom can easily be extended to also include bulk operations and range operations that we have described in this paper. As presented earlier, these operations acquire the base node locks in a specific order (increasing order of the keys that they can store), with the exception that they might also perform a low-contention join which cannot cause deadlocks as we have described above. Thus, a CA tree with multi-element operations is deadlock free since there is a specific order in which the locks are acquired. Whenever locks are acquired in a different order, this is done with a try lock call and all held locks are released if the try lock call fails.

Livelock freedom can be shown by proving that when an operation or part of an operation has to be retried due to interference from another thread, some other thread must have made progress. The two types of retries are the same for both multi-element operations and single-key operations. The first type of retry can happen in the function for low-contention join and is caused by a concurrent low-contention join that removes a routing node. This can not cause a livelock since, if a retry is triggered at this point, another thread must have successfully spliced out a routing node from the tree and this routing node will not be observed when we retry; cf. [16]. The second type of retry happens when an invalid base node is observed. An invalid base node is only observed if another thread has successfully performed a contention-adapting split or join which means that another thread has made progress. Single-key operations handle this case by retrying the whole operation, while operations involving multiple keys only need to retry the search for the next base node. When the search for a base node is retried the same invalid base node will not be found since the effect of the split or join that sets the base node to invalid will be visible after the base node(s) involved in the split or join has(have) been unlocked.

Linearizability. The linearization point of an operation that locks all base nodes that it reads from or writes to is at some point while holding the base node locks of all the base nodes that it operates on. The linearization point of an operation that is successfully performed with an optimistic read attempt is somewhere between the first and second sequence number scan. If the optimistic read attempt fails, the operation will instead acquire the locks non-optimistically and the linearization point will be at some point while holding all the base node locks. It can be proven [16] that CA trees maintain the following property: If a thread t has searched in a CA tree for a key K using the binary search tree property and ended up in base node B that it has locked and validated, then K must be in B and not in any other base node if it is in the abstract set represented by the CA tree. Using this property as well as the properties mentioned above in the arguments for the correctness of range operations it is easy to see that the CA tree operations appear to happen atomically at their linearization points, since they are either holding locks of all base nodes that can contain keys involved in the operation or ensuring that no other thread has changed any key involved in the operation while the operation is being performed by the final check of the sequence numbers in the sequence locks.

Flexibility of CA Trees. A split operation in an ordered set data structure splits the data structure into two data structures so that all elements in one are smaller than the elements in the other. The join operation merges two data structures where the greatest key in one of them is smaller than the smallest key in the other. Any sequential ordered set data structure that has efficient support for the split and join operations can be used to store elements under the base nodes of CA trees. This property makes CA trees highly flexible since the underlying sequential data structure can be changed without changing the CA tree structure itself. The sequential data structure component of a CA tree could be passed as a parameter by the user when creating a CA tree instance. One could even change the sequential ordered set data structure at run time depending on which type of operations are most frequent; however, it is beyond the scope of this paper to investigate the effect of this possibility.

Many ordered set data structures support efficient split and join operations including red-black trees and AVL trees that do these operations in $\mathcal{O}(log(N))$ time [9, 19]. Skip lists are randomized data structures for ordered sets that also have efficient support for split[2] and join [14]. By using both back and forward pointers in the skip list, both split and join as well as maxKey have efficient implementations; in fact constant time in skip lists with a fixed number of levels. Skip lists also provide efficient support for range operations since all elements are connected in an ordered list at the top level of a skip list. Using a skip list with so called *fat nodes*, i.e., nodes that contain more than one element, we can further increase the performance of range operation due to improved locality. We will experiment with AVL trees and skip lists with fat nodes in the next section. Our skip list implementation can store up to k elements in its nodes. The nodes are split if an insert would cause a node to contain $k + 1$ elements, and nodes are spliced out if a remove operation would create an empty node. The keys in the skip list are kept in compact arrays to improve cache locality when searching and performing range operations.

4 Experiments

Let us now investigate the scalability of two CA tree variants: one with an AVL tree as sequential structure (CA-AVL) and one with a skip list with fat nodes (CA-SL) as sequential structure. We compare them against the lock-free k-ary search tree [3] (k-ary), the Snap tree [2] (Snap) and a lock-free skip list (SkipList). All implementations are those provided by the authors. SkipList is implemented by Doug Lea in the Java Foundation Classes as the class `ConcurrentSkipListMap`.[3]

SkipList marked with dashed gray lines in the graphs does not cater for linearizable range queries nor range updates. We include SkipList in the measurements only to show the kind of scalability one can expect from a lock-free skip list data structure if one is

[2] The efficient skip list split operation splits the data structure so that on average half the keys will be in each resulting split.

[3] We do not compare experimentally against the Leaplist [1] whose main implementation is in C. Prototype implementations of the Liplist in Java were sent to us by its authors, but they ended up in deadlocks when running our benchmarks which prevented us from obtaining reliable measurements. Instead, we refer to Section 2 for an analytic comparison to the Leaplist.

not concerned about consistency of results from range operations. Range operations are implemented in SkipList by calling the `subSet` method which returns an iterable view of the elements in the range. Since changes in SkipList are reflected in the view returned by `subSet` and vice versa, range operations are not atomic.

In contrast, the k-ary search tree supports linearizable range queries and the Snap tree supports linearizable range queries through the `clone` method. However, neither the k-ary nor the Snap tree provide support for linearizable range updates. In the scenarios where we measure range updates we implement them in these data structures by using a frequent read optimized readers-writer lock[4] with a read indicator that has one dedicated cache line per thread. Thus, all operations except range updates acquire the RW-lock in read mode. We have confirmed that this method has negligible overhead for all cases where range updates are not used, but use the implementations of the data structures without range update support in scenarios that do not have range updates.

We use $k = 32$ (maximum number of elements in nodes) both for CA-SL and k-ary trees. This value provides a good trade-off between performance of range operations and performance of single-key modification operations. For the CA trees, we initialize the contention statistics counters of the locks to 0 and add 250 to the counter to indicate contention; we decrease the counter by 1 to indicate low contention. The thresholds -1000 and 1000 are used for low contention and high contention adaptations.

The benchmark we use measures throughput of a mix of operations performed by N threads on the same data structure during T seconds. The keys and values for the operations get, insert and remove as well as the starting key for range operations are randomly generated from a range of size R. The data structure is pre-filled before the start of each benchmark run by performing $R/2$ insert operations. In all experiments presented in this paper $R = 1000000$, thus we create a data structure containing roughly 500000 elements. In all captions, benchmark scenarios are described by a strings of the form w:A% r:B% q:C%-R_1 u:D%-R_2, meaning that on the created data structure the benchmark performs $(A/2)\%$ insert, $(A/2)\%$ remove, $B\%$ get operations, $C\%$ range queries of maximum range size R_1, and $D\%$ range updates with maximum range size R_2. The size of each range operation is randomly generated between 1 and the maximum range size. The benchmarks presented in this paper were run on a machine with four AMD Opteron 6276 (2.3 GHz, 16 cores, 16M L2/16M L3 Cache), giving a total of 64 physical cores and 128 GB or RAM, running Linux 3.10-amd64 and Oracle Hotspot JVM 1.8.0_31 (started with parameters `-Xmx4g`, `-Xms4g`, `-server` and `-d64`). [5] The experiments for each benchmark scenario were run in a separate JVM instance and we performed a warm up run of 10 seconds followed by three measurement runs, each running for 10 seconds. The average of the measurement runs as well as error bars for the minimum and maximum run are shown in the graphs, though often the error bars are very small and therefore not visible.

[4] We use the write-preference algorithm [5] for coordination between readers and writers and the `StampedLock` from the Java library for mutual exclusion.

[5] We also ran experiments on a machine with four Intel(R) Xeon(R) E5-4650 CPUs (2.70GHz each with eight cores and hyperthreading) both on a NUMA setting and on a single chip, showing similar performance patterns as on the AMD machine. Results are available online [6].

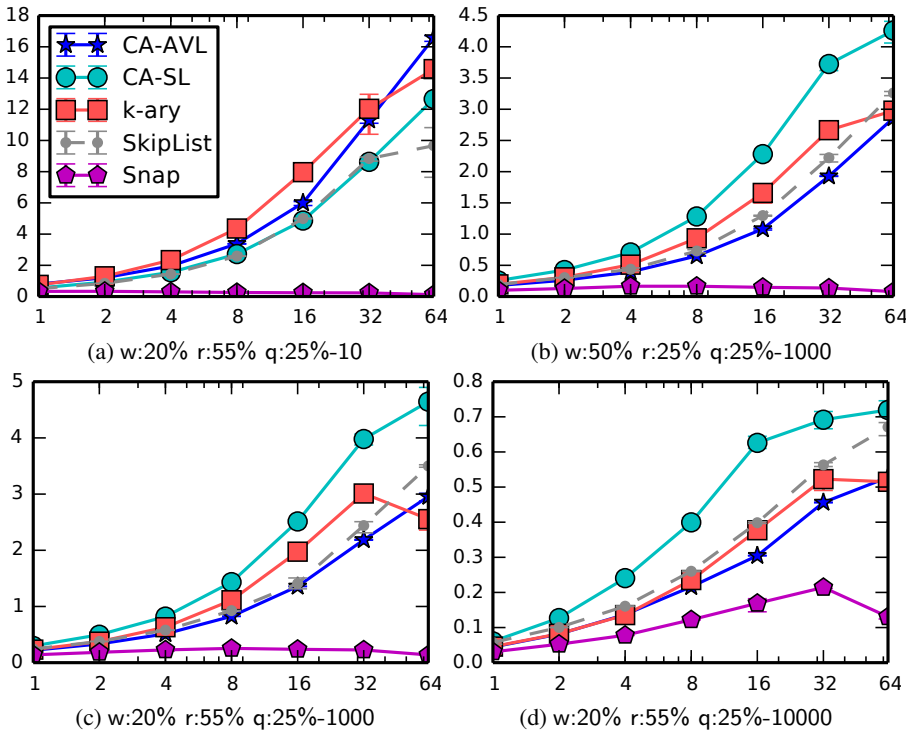
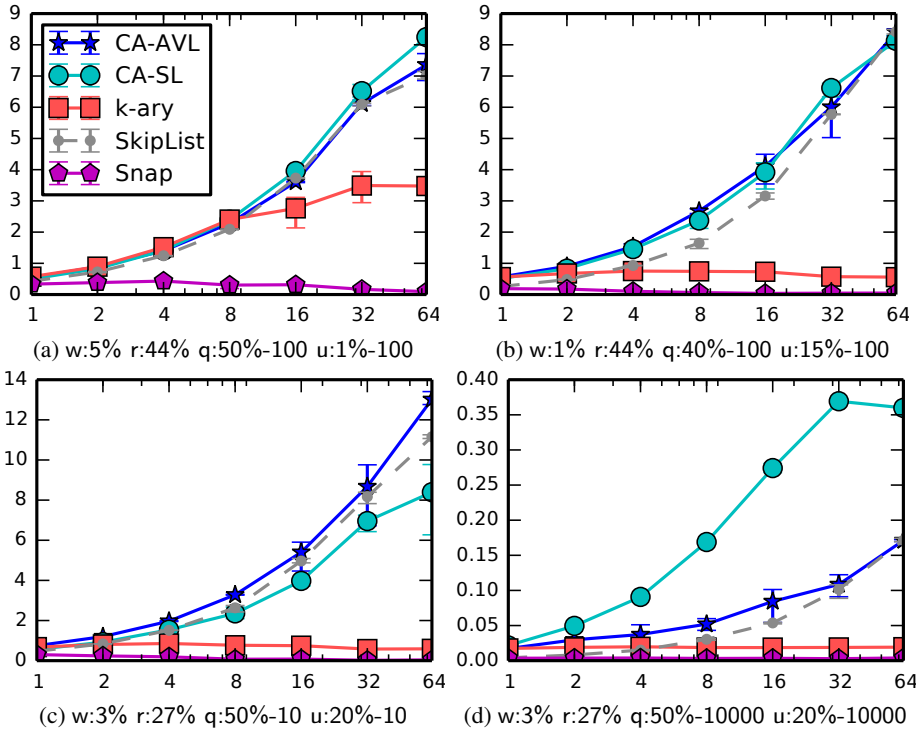


Fig. 5: Scalability of throughput (ops/μs) on the y-axis and thread count on the x-axis.

Benchmarks without Range Updates. Let us first discuss the performance results in Fig. 5, showing scenarios without range updates. Figure 5a, which shows throughput in a scenario with a moderate amount of modifications (20%) and small range queries, shows that the k-ary and CA-AVL tree perform best in this scenario, tightly followed by the CA-SL and SkipList with the non-atomic range queries. We also note that the Snap tree does not scale well in this scenario, which is not surprising since a range query with a small range size will eventually cause the creation of a copy of every node in the tree. Let us now look at Fig. 5b showing throughputs in a scenario with many modifications (50%) and larger range queries, and Fig. 5c corresponding to a scenario with the same maximum range query size and a more moderate modification rate (20%). First of all, the better cache locality for range queries in CA-SL and k-ary trees is visible in these scenarios where the range sizes are larger. k-ary only beats CA-AVL with a small amount up to 32 threads and then k-ary's performance drops. This performance drop might be caused by its starvation issue in the range query operation that can cause a range query to be retried many times (possibly forever). This can be compared to the CA trees that acquire locks for reads if the first optimistic attempt fails and thus reducing the risk of retries. The scalability of the CA trees shown in Fig. 5b, i.e., in a scenario with 50% modification operations, shows that the range queries in the CA trees tolerate high contention. Finally, the scenario of Fig. 5d with very wide range queries and moderate modification rate (20%) shows both the promise and the limit in the scalability of CA-SL.



(a) w:5% r:44% q:50%-100 u:1%-100

(b) w:1% r:44% q:40%-100 u:15%-100

(c) w:3% r:27% q:50%-10 u:20%-10

(d) w:3% r:27% q:50%-10000 u:20%-10000

Fig. 6: Scalability of throughput (ops/μs) on the y-axis and thread count on the x-axis.

However, we note that SkipList, which does not even provide atomic range queries, does not beat CA-SL that outperforms the other data structures by at least 57% at 16 threads.

Benchmarks with Range Updates. Let us now look at the scenarios that also contain range updates shown in Fig. 6. The first of them (Fig. 6a) shows that k-ary tree's scalability flattens out between 16 and 32 threads even with as little as 1% range updates. Instead, the CA trees provide good scalability all the way. Remember that we wrap the k-ary operations in critical sections protected by an RW-lock to provide linearizable range updates in the k-ary tree. In the scenario of Fig. 6b, where the percentage of range updates is 15%, we see that the k-ary tree does not scale at all while the CA trees and SkipList with the non-atomic range operations scale very well, outperforming the k-ary tree with more than 1200% in this case. The two scenarios in Fig. 6c and 6d have the same rate of operations but different maximum size for range queries and range updates. Their results clearly show the difference in performance characteristics that can be obtained by changing the sequential data structure component of a CA tree. CA-SL is faster for wider range operations due to its fat nodes providing good cache locality, but CA-SL is generally slower than the CA-AVL in scenarios with small range sizes. In Fig. 6d, where the conflict rate between operations is high, CA-SL reaches its peak performance at 32 threads where it outperforms all other data structures by more than two times.

R	10	100	1000	10000
CA-SL	14.4	8.8	4.0	2.5
CA-AVL	15.6	8.7	3.6	2.2

(a) w:3% r:27% q:50%-R u:20%-R

threads	2	4	8	16	32	64
CA-SL	0.36	0.73	1.2	1.9	2.7	4.0
CA-AVL	0.34	0.68	1.1	1.6	2.4	3.6

(b) w:3% r:27% q:50%-1000 u:20%-1000

Table 1: Average base node counts (in k) at the end of running two sets of benchmarks: one using 64 threads but varying the range size R, and one varying the number of threads.

We also report average base node counts for the CA trees in the end of running two sets of scenarios. The numbers in Table 1a show node counts (in k) for running with 64 threads but varying the maximum range size R. Table 1b shows node counts (also in k) for scenarios with R fixed to 1000 but varying the number of threads. These numbers confirm that the CA trees' synchronization is adapting both to the contention level (increasing the number of threads results in more base nodes) and to the access patterns (increased range size results in fewer base nodes). We also confirmed by increasing the running time of the experiments that the number of base nodes stabilizes around a specific value for each scenario, which means that base nodes do not get split indefinitely.

5 Concluding Remarks

Given the diversity in sizes and heterogeneity of multicores, it seems rather obvious that current and future applications will benefit from, if not require, data structures that can adapt dynamically to the amount of concurrency and the usage patterns of applications.

This paper has advocated the use of CA trees, a new family of lock-based concurrent data structures for ordered sets of keys and key-value pair dictionaries. CA trees' salient feature is their ability to adapt their synchronization granularity according to the current contention level and access patterns. Furthermore, CA trees are flexible and efficiently support a wide variety of operations: single-key operations, multi-element operations, range queries and range updates. Our experimental evaluation has demonstrated the good scalability and superior performance of CA trees compared to state-of-the-art lock-free concurrent data structures in a variety of scenarios.

In other work [17], we have described the use of CA trees for speeding and scaling up single-key operations of the `ordered_set` component of the Erlang Term Storage, Erlang's in-memory key-value store. We intend to extend that work with support for atomic multi-element and range operations and evaluate the performance benefits of doing so in "real-world" applications. The experimental results in this paper strongly suggest that the performance gains will be substantial. In addition, we intend to investigate CA trees with more kinds of adaptations: for example, adaptations in the underlying sequential data structure component.

References

1. H. Avni, N. Shavit, and A. Suissa. Leaplist: Lessons learned in designing TM-supported range queries. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 299–308, New York, NY, USA, 2013. ACM.

2. N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 257–268. ACM, 2010.
3. T. Brown and H. Avni. Range queries in non-blocking k-ary search trees. In R. Baldoni, P. Flocchini, and R. Binoy, editors, *Principles of Distributed Systems*, volume 7702 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2012.
4. T. Brown and J. Helga. Non-blocking k-ary search trees. In A. Fernàndez Anta, G. Lipari, and M. Roy, editors, *Principles of Distributed Systems*, volume 7109 of *Lecture Notes in Computer Science*, pages 207–221. Springer, 2011.
5. I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. NUMA-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 157–166, New York, NY, USA, 2013. ACM.
6. CA Trees. http://www.it.uu.se/research/group/languages/software/ca_tree.
7. T. Crain, V. Gramoli, and M. Raynal. A speculation-friendly binary search tree. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 161–170, New York, NY, USA, 2012. ACM.
8. K. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge Computer Laboratory, 2004.
9. D. E. Knuth. *The Art of Computer Programming: Sorting and Searching, vol. 3*. Addison-Wesley, Reading, 2nd edition, 1998.
10. C. Lameter. Effective synchronization on Linux/NUMA systems. In *Proc. of the Gelato Federation Meeting*, 2005.
11. A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 317–328, New York, NY, USA, 2014. ACM.
12. E. Österlund and W. Löwe. Concurrent transformation components using contention context sensors. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 223–234, New York, NY, USA, 2014. ACM.
13. A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 151–160, NY, USA, 2012. ACM.
14. W. Pugh. A skip list cookbook. Technical report, College Park, MD, USA, 1990.
15. C. Robertson. Implementing contention-friendly range queries in non-blocking key-value stores. Bachelor thesis, The University of Sydney, Nov. 2014.
16. K. Sagonas and K. Winblad. Contention adapting trees. Tech. Report, available in [6], 2014.
17. K. Sagonas and K. Winblad. More scalable ordered set for ETS using adaptation. In *ACM Erlang Workshop*, pages 3–11. ACM, Sept. 2014.
18. K. Sagonas and K. Winblad. Contention adapting trees. In *14th International Symposium on Parallel and Distributed Computing*, pages 215–224. IEEE, June 2015.
19. R. E. Tarjan. *Data Structures and Network Algorithms*, volume 14. SIAM, 1983.