A Concurrency-Optimal Binary Search Tree*

Vitaly Aksenov[1], Vincent Gramoli[2], Petr Kuznetsov[3], Anna Malova[4], and Srivatsan Ravi[5]

[1] INRIA Paris / ITMO University
[2] University of Sydney
[3] LTCI, Télécom ParisTech, Université Paris-Saclay
[4] Washington University in St Louis
[5] Purdue University

Abstract. The paper presents the first *concurrency-optimal* implementation of a binary search tree (BST). The implementation, based on a standard sequential implementation of a partially-external tree, ensures that every *schedule*, i.e., interleaving of steps of the sequential code, is accepted unless linearizability is violated. To ensure this property, we use a novel read-write locking protocol that protects tree *edges* in addition to its nodes.
Our implementation performs comparably to the state-of-the-art BSTs and even outperforms them on few workloads, which suggests that optimizing the set of accepted schedules of the sequential code can be an adequate design principle for efficient concurrent data structures.

Keywords: Concurrency optimality; Binary search tree, Linearizability

1 Introduction

To meet modern computational demands and to overcome the fundamental limitations of computing hardware, the traditional single-CPU architecture is being replaced by a concurrent system based on multi-cores or even many-cores. Therefore, at least until the next technological revolution, the only way to respond to the growing computing demand is to invest in smarter concurrent algorithms.

Synchronization, one of the principal challenges in concurrent programming, consists in arbitrating concurrent accesses to shared *data structures*: lists, hash tables, trees, etc. Intuitively, an efficient data structure must be *highly concurrent*: it should allow multiple processes to "make progress" on it in parallel. Indeed, every new implementation of a concurrent data structure is usually claimed to enable such a parallelism. But what does "making progress" means precisely?

Optimal concurrency. If we zoom in the code of an operation on a typical concurrent data structure, we can distinguish *data accesses*, i.e., reads and

* Vincent Gramoli was financially supported by the Australian Research Council (Discovery Projects funding scheme, project number 160104801 entitled "Data Structures for Multi-Core"). Vitaly Aksenov was financially supported by the Government of Russian Federation (Grant 074-U01) and by the European Research Council (Grant ERC-2012-StG-308246).

updates to the data structure itself, performed as though the operation works on the data in the absence of concurrency. To ensure that concurrent operations do not violate correctness of the implemented high-level data type (e.g., *linearizability* [1] of the implemented set abstraction), data accesses are "protected" with *synchronization primitives*, e.g., acquisitions and releases of locks or atomic read-modify-write instructions like compare-and-swap. Intuitively, a process makes progress by performing "sequential" data accesses to the shared data, e.g., traversing the data structure and modifying its content. In contrast, synchronization tasks, though necessary for correctness, do not contribute to the progress of an operation.

Hence, "making progress in parallel" can be seen as allowing concurrent execution of pieces of locally sequential fragments of code. The more synchronization we use to protect "critical" pieces of the sequential code, the less *schedules*, i.e., interleavings of data accesses, we accept. Intuitively, we would like to use exactly as little synchronization as sufficient for ensuring linearizability of the high-level implemented abstraction. This expectation brings up the notion of a *concurrency-optimal* implementation [2] that only rejects a schedule if it does violate linearizability.

To be able to reason about the "amount of concurrency" exhibited by implementations employing different synchronization techniques, we consider the recently introduced notion of "local serializability" (on top of linearizability) and the metric of the "amount of concurrency" defined via sets of accepted (locally sequential) schedules [3]. Local serializability, intuitively, requires the sequence of sequential steps locally observed by every given process to be consistent with *some* execution of the sequential algorithm. Note that these sequential executions can be different for different processes, i.e., the execution may not be *serializable* [4]. Combined with the standard correctness criterion of *linearizability* [5, 6]), local serializability implies our basic correctness criterion called *LS-linearizability*. The concurrency properties of LS-linearizable data structures can be compared on the same level: implementation A is "more concurrent" than implementation B if the set of schedules accepted by A is a strict superset of the set of schedules accepted by B. Thus, a *concurrency-optimal* implementation accepts *all* correct (LS-linearizable) schedules.

A concurrency-optimal binary search tree. It is interesting to consider binary search trees (BSTs) from the optimal concurrency perspective, as they are believed, as a representative of *search* data structures [7], to be "concurrency-friendly" [8]: updates concerning different keys are likely to operate on disjoint sets of tree nodes (in contrast with, e.g., operations on queues or stacks).

We present a novel LS-linearizable concurrent BST-based set implementation. We prove that the implementation is optimally concurrent with respect to a standard internal sequential tree [3]. The proposed implementation employs the optimistic "lazy" locking approach [9] that distinguishes *logical* and *physical* deletion of a node and makes sure that read-only operations are *wait-free* [1], i.e., cannot be delayed by concurrent processes.

The algorithm also offers a few algorithmic novelties. Unlike most implementations of concurrent trees, the algorithm uses multiple locks per node: one lock for the *state* of the node, and one lock for each of its descendants. To ensure that only conflicting operations can delay each other, we use *conditional* read-write locks, where the lock can be acquired only under certain condition. Intuitively, only changes in the relevant part of the tree structure may prevent a thread from acquiring the lock. The fine-grained conditional read-write locking of nodes and edges allows us to ensure that an implementation rejects a schedule only if it violates linearizability.

Concurrency optimality and performance. Of course, optimal concurrency does not necessarily imply performance nor maximum progress (à la *wait-freedom* [10]). An extreme example is the transactional memory (TM) data structure. TMs typically require restrictions of serializability as a correctness criterion. And it is known that rejecting a schedule that is rejected only if it is not serializable (the property known as *permissiveness*), requires very heavy local computations [11, 12]. But the intuition is that looking for concurrency-optimal search data structures like trees pays off. And this work answers this question in the affirmative by demonstrating empirically that the Java implementation of our concurrency optimal BST outperforms state-of-the-art BST implementations ([13–16]) on most workloads. Apart from the obvious benefit of producing a highly efficient BST, this work suggests that optimizing the set of accepted schedules of the sequential code can be an adequate design principle for building efficient concurrent data structures.

Roadmap. The rest of the paper is organized as follows. § 2 describes the details of our BST implementation, starting from the sequential implementation of *partially-external* binary search tree, our novel conditional read-lock lock abstraction to our concurrency optimal BST implementation. § 3 formalizes the notion of concurrency optimality and sketches the relevant proofs; detailed proofs are delegated to the appendix. § 4 provides details of our experimental methodology and extensive evaluation of our Java implementation. § 5 articulates the differences with related BST implementations and presents concluding remarks.

2 Binary Search Tree Implementation

This section consists of two parts. At first, we describe our sequential implementation of the *set* using partially-external binary search tree. Then, we build the concurrent implementation on top of the sequential one by adding synchronization separately for each field of a node. Our implementation takes only the locks that are neccessary to perform correct modifications of the tree structure. Moreover, if the field is not going to be modified, the algorithm takes the read lock instead of the write lock.

We start with the specification of the set type which our binary search tree should satisfy. An object of the *set* type stores a set of integer values, initially empty, and exports operations $\mathsf{insert}(v)$, $\mathsf{remove}(v)$, $\mathsf{contains}(v)$. The update operations, $\mathsf{insert}(v)$ and $\mathsf{remove}(v)$, return a boolean response, *true* if and only if

v is absent (for insert(v)) or present (for remove(v)) in the *set*. After insert(v) is complete, v is present in the set, and after remove(v) is complete, v is absent. The contains(v) returns a boolean response, *true* if and only if v is present.

A *binary search tree*, later called BST, is a rooted ordered tree in which each node v has a left child and a right child, either or both of which can be null. The node is named a *leaf*, if it does not have any child. The order is carried by a value property: the value of each node is stricly greater than the values in its left subtree and strictly smaller than the values in the right subtree.

2.1 Sequential implementation

As for a sequential implementation we chose the well-known *partially-external* binary search tree. Such tree combines the idea of the internal binary search tree, where the set is represented by the values from all nodes, and the external binary search tree, where the set is represented by the values in the leaves while the inner nodes are used for routing (note, that for the external tree the value property does not consider leafs). The partially-external tree supports two types of nodes: routing and data. The set is represented by the values contained by the data nodes. To bound the number of routing vertices by the number of data nodes the tree should satisfy the condition: all routing nodes have exactly two children.

The pseudocode of the sequential implementation is provided in the Algorithm 1. Here, we give a brief description. The **traversal** function takes a value v and traverses down the tree from the root following the corresponding links as long as the current node is not null or its value is not v. It returns the last three visited nodes. The **contains** function takes a value v and checks the last node visited by the traversal and returns whether it is null. The **insert** function takes a value v and uses the traversal function to find the place to insert the value. If the node is not null, the algorithm checks whether the node is data or routing: in the former case it is impossible to insert; in the latter case, the algorithm simply changes the state from routing to data. If the node is null, then the algorithm assumes that the value v is not in the set and inserts a new node with the value v as the child of the latest non-null node visited by the traversal function call. The **delete** function takes a value v and uses the traversal function to find the node to delete. If the node is null or its state is routing, the algorithm assumes that the value v is not in the set and finishes. Otherwise, there are three cases depending on the number of children that the found node has: (i) if the node has two children, then the algorithm changes its state from data to routing; (ii) if the node has one children, then the algorithm unlinks the node; (iii) finally if the node is a leaf then the algorithm unlinks the node, in addition if the parent is a routing node then it also unlinks the parent.

2.2 Concurrent implementation

As the basis of our concurrent implementation we took the idea of optimistic algorithms, where the algorithm reads all necessary variables without synchro-

Algorithm 1 Sequential implementation.

```
1:  Shared variables:
2:    node is a record with fields:
3:    val, its value
4:    left, its pointer to the left child
5:    right, its pointer to the right child
6:    state ∈ {DATA, ROUTING}, its state
7:    Initially the tree contains one node root,
8:    root.val = +∞
9:    root.state = DATA

10: traversal(v):                    ▷ wait-free traversal
11:   gprev ← null; prev ← null
12:   curr ← root                    ▷ start from root
13:   while curr ≠ null do
14:     if curr.val = v then
15:       break
16:     else
17:       gprev ← prev
18:       prev ← curr
19:       if curr.val < v then
20:         curr ← curr.left
21:       else
22:         curr ← curr.right
23:   return ⟨gprev, prev, curr⟩

24: contains(v):                     ▷ wait-free contains
25:   ⟨gprev, prev, curr⟩ ← traversal(v)
26:   return curr ≠ null and curr.state = DATA

27: insert(v):
28:   ⟨gprev, prev, curr⟩ ← traversal(v)
29:   if curr ≠ null then ▷ node has value v or is a place to insert
30:     go to Line 8
31:   else
32:     go to Line 12
33:   return true

34: Update existing node:
35:   if curr.state = DATA then
36:     return false
                                     ▷ v is already in the set
37:   curr.state ← DATA

38: Insert new node:
39:   newNode.val ← v ▷ allocate a new node
40:   if v < prev.val then
41:     prev.left ← newNode
42:   else
43:     prev.right ← newNode

44: delete(v):
45:   ⟨gprev, prev, curr⟩ ← traversal(v)
46:   if curr = null or curr.state ≠ DATA then
47:     return false
                                     ▷ v is not in the set
48:   if curr has exactly 2 children then
49:     go to Line 34
50:   if curr has exactly 1 child then
51:     go to Line 36
52:   if curr is a leaf then
53:     if prev.state = DATA then
54:       go to Line 46
55:     else
56:       go to Line 51
57:   return true

58: Delete node with two children:
59:   curr.state ← ROUTING

60: Delete node with one child:
61:   if curr.left ≠ null then
62:     child ← curr.left
63:   else
64:     child ← curr.right
65:   if curr.val < prev.val then
66:     prev.left ← child
67:   else
68:     prev.right ← child

69: Delete leaf with DATA parent:
70:   if curr is left child of prev then
71:     prev.left ← null
72:   else
73:     prev.right ← null

74: Delete leaf with ROUTING parent:
75:     ▷ save second child of prev into child
76:   if curr is left child of prev then
77:     child ← prev.right
78:   else
79:     child ← prev.left
80:   if prev is left child of gprev then
81:     gprev.left ← child
82:   else
83:     gprev.right ← child
```

nizations and right before the modification, the algorithm takes all the locks and checks the consistency of all the information it read. As we show in the next section, we build upon the partially-external property of the BST to provide a concurrency-optimal BST. Let us first give more details on how the algorithm is implemented.

Field reads. Since our algorithm is optimistic we do not want to read the same field twice. To overcome this problem when the algorithm reads the field it stores it in "cache" and the further accesses return the "cached" value. For example, the reads of the $left$ field in Lines 28 and 29 of Algorithm 2 return the same (cached) value.

Deleted mark. As usual in concurrent algorithms with wait-free traversals, the deletion of the node happens in two stages. At first, the delete operation logically removes a node from the tree by setting the boolean flag to **deleted**. Secondly, the delete operation updates the links to physically remove the node. By that, any traversal that suddenly reaches the "under-deletion" node, sees the deletion node and could restart the operation.

Locks. In the beginning of the section we noted that we have locks separately for each field of a node and the algorithm takes only the necessary type of lock: read or write. For that, we implemented read-write lock simply as one *lock* variable. The smallest bit of *lock* indicates whether the write lock is taken or not, the rest part of the variable indicates the number of readers that have taken a lock. In other words, *lock* is zero if the lock is not taken, *lock* is one if the write lock is taken, otherwise, *lock* divided by two represents the number of times the read lock is taken. The locking and unlocking are done using the atomic compare-and-set primitive. Along, with standard tryWriteLock, tryReadLock, unlockWrite and unlockRead we provide additional six functions on a node: tryLockLeftEdge(Ref|Val)(exp), lockRightEdge(Ref|Val)(exp) and try(Read|Write)LockState(exp) (Starting from here, we use the notation of bar to not duplicate the similar names; such notation should be read as either we choose the first option or the second option.)

Function tryLock(Left|Right)EdgeRef ensures that the lock is taken only if the field ($left$ or $right$) guarded by that lock is equal to exp, i.e., the child node has not changed, and the current node is not deleted, i.e., its deleted mark is not set. Function tryLock(Left|Right)EdgeVal ensures that the lock is taken only if the value of the node in the field ($left$ or $right$) guarded by that lock is equal to exp , i.e., the node could have changed by the value inside does not, and the current node is not deleted, i.e., its deleted mark is not set. Function try(Read|Write)LockState(exp) ensures that the lock is taken only if the value of the *state* is equal to exp and the current node is not deleted, i.e., its deleted mark is not set.

These six functions are implemented in the same manner: the function reads necessary fields and lock variable, checks the conditions, if successful it takes a corresponding lock, then checks the conditions again, if unsuccessful it releases lock. In most cases in the pseudocode we used a substitution $tryLockEdge(Ref|Val)(node)$ instead of $tryLock(Left|Right)Edge(Ref|Val)(exp)$. This substitution, given not-null value, decides whether the *node* is the left or right child of the current node and calls the corresponding function providing *node* or *node.value*.

3 Concurrency optimality and correctness

In this section, we show that our implementation is *concurrency-optimal* [3]. Intuitively, a concurrency-optimal implementation employs as much synchronization as necessary for ensuring correctness of the implemented high-level abstraction — in our case, the linearizable set object [1].

Recall our *sequential* BST implementation and imagine that we run it in a *concurrent* environment. We refer to an execution of this concurrent algorithm as a *schedule*. A schedule thus consists of reads, writes, node creation events, and invocation and responses of high-level operations.

Notice that in every such schedule, any operation witnesses a *consistent* tree state locally, i.e., it cannot distinguish the execution from a sequential one. It is easy to see that the local views *across operations* may not be mutually consistent, and this simplistic concurrent algorithm is not linearizable. For example, two insert operations that concurrently traverse the tree may update the same node so that one of the operations "overwrites" the other (so called the "lost update" problem). To guarantee linearizability, one needs to ensure that only correct (linearizable) schedules are accepted. We show first that this is indeed the case with our algorithm: all the schedules it *accepts* are correct. More precisely, a schedule σ is accepted by an algorithm if it has an execution in which the sequence of high-level invocations and responses, reads, writes, and node creation events (modulo the restarted fragments) is σ [3].

Theorem 1 (Correctness). *The schedule corresponding to any execution of our BST implementation is observably correct.*

A complete proof of Theorem 1 is given in the appendix.

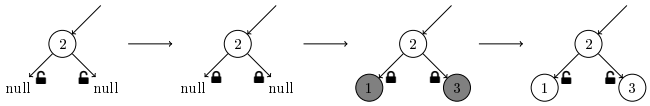
Further, we show that, in a strict sense, our algorithm accepts *all* correct schedules. In our definition of correctness, we demand that at all times the algorithm maintains a *BST* that does not contain nodes that were previously *physically deleted*. Formally, a set of nodes reachable from the *root* is a *BST* if: (i) they form a tree rooted at node *root*; (ii) this tree satisfies the *value property*: for each node with value v all the values in the left subtree are less than v and all the values in the right subtree are bigger than v; (iii) each routing node in this tree has two children.

Now we say that a schedule is *observably correct* if each of its prefixes σ satisfies the following conditions: (i) subsequence of high-level invocations and responses in σ is linearizable with respect to the `set` type; (ii) the data structure after performing σ is a BST; (iii) the BST after σ does not contain a node x such that there exist σ' and σ'', such that σ' is a prefix of σ'', σ'' is a prefix of σ, x is in the BST after σ', and x is not in the BST after σ''.

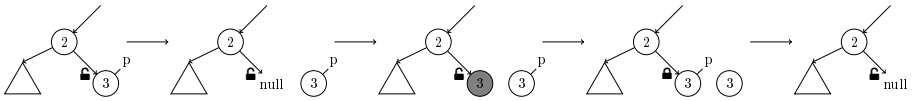
We say that an implementation is *concurrency-optimal* if it accepts *all* observably correct schedules.

Theorem 2 (Optimality). *Our BST implementation is concurrency-optimal.*

A complete proof of Theorem 1 is given in the appendix. The intuition behind the proof of Theorem 2 is the following. We show that for each observably correct



(a) Scenario depicting a concurrent execution of insert(1) and insert(3); rejected by popular BSTs like [13–16], it is accepted by a concurrency-optimal BST



(b) Scenario depicting an execution of two concurrent delete(3) operations, followed by a successful insert(3); rejected by all the popular BSTs [13–17], it is accepted by a concurrency-optimal BST

Fig. 1: Examples schedules rejected by concurrent BSTs not concurrency-optimal

schedule there exists a matching execution of our implementation. Therefore, only schedules not observably correct can be rejected by our algorithm. The construction of an execution that matches an observably correct schedule is possible, in particular, due to the fact that every critical section in our algorithm contains exactly one event of the schedule. Thus, the only reason to reject a schedule is that some condition on a critical section does not hold and, as a result, the operation must be restarted. By accounting for all the conditions under which an operation restarts, we show that this may only happen if, otherwise, the schedule violates observable correctness.

Suboptimality of related BST algorithms. To understand the hardness of building linearizable concurrency optimal BSTs, we explain how some typical correct schedules are rejected by current state-of-the-art BST algorithms against which we evaluate the performance of our algorithm. Consider the concurrency scenario depicted in Figure 1a. There are two concurrent operations insert(1) and insert(3) performed on a tree. They traverse to the corresponding links (part a)) and lock them concurrently (part b)). Then they insert new nodes (part c)). Note that this is a correct schedule of events; however, most BSTs including the ones we compare our implementation against [13–16] reject this schedule or similar. However, using multiple locks per node allows our concurrency-optimal implementation to accept this schedule.

The second schedule is shown in the Figure 1b. There is one operation $p =$ delete(3) performed on a tree shown in part a). It traverses to a node v with value 3. Then, some concurrent operation delete(3) unlinks node v (part b)). Later, another concurrent operation inserts a new node with value 3 (part c)). Operation p wakes up and locks a link since the value 3 is the same (part d)). Finally, p unlinks the node with value 3 (part e)). Note that this is a correct schedule since both the delete operations can be successful; however, all the BSTs we are aware of reject this schedule or similar [13–17]. While, there is an execution of our concurrency-optimal BST that accepts this schedule.

4 Implementation and evaluation

Experimental setup. For our experiments we used two machines to evaluate the versioned binary search tree. The first is a 4-processor Intel Xeon E7-4870 2.4 GHz server (Intel) with 20 threads per processor (yielding 80 hardware threads in total), 512 Gb of RAM, running Fedora 25. This machine has Java 1.8.0_111-b14 and HotSpot VM 25.111-b14. Second machine is a 4-processor AMD Opteron 6378 2.4 GHz server (AMD) with 16 threads per processor (yielding 64 threads in total), 512 Gb of RAM, running Ubuntu 14.04.5. This machine has Java 1.8.0_111-b14 and HotSpot JVM 25.111-b14.

Binary Search Tree Implementations. We compare our algorithm, denoted as Concurrency Optimal or CO, against four other implementations of concurrent BST. They are: 1) the lock-based contention-friendly tree by Crain et al. ([13], Concurrency Friendly or CF), 2) the lock-based logical ordering AVL-tree by Drachsler et al. ([14], Logical Ordering or LO) 3) the lock-based tree by Bronson et al. ([15], BCCO) and 4) the lock-free tree by Ellen et al. ([16], EFRB). All these implementations are written in Java and taken from the `synchrobench repository` [18]. In order to make the comparison equitable, we remove rotation routines from the CF-, LO- and CO- trees implementations. We are aware of efficient lock-free tree by Natarajan and Mittal ([17]), but unfortunately we were unable to find it written on Java.

Experimental methodology. For our experiments, we use the environment provided by the synchrobench library. To compare the performance we considered the following parameters:

- **Workloads.** Each workload distribution is characterized by the percent $x\%$ of update operations. This means that the tree will be requested to make $100-x\%$ of `contains` calls, $x/2\%$ of `insert` calls and $x/2\%$ of `delete` calls. We considered three different workload distributions: 0%, 20% and 100%.
- **Tree size.** On the workloads described above, the tree size depends on the size of the key space (the size is approximately half of the range). We consider three different key ranges: 2^{15}, 2^{19} and 2^{21}. To ensure consistent results, rather than starting with an empty tree, we pre-populated the tree before execution.
- **Degree of contention.** This depends on the number of threads in a machine. We take enough points to reason about the behaviour of curves.

In fact, we made experiments on a larger number of settings but we shortened our presentation due to lack of space. We chose the settings such that we had two extremes and one middle point. For workload, we chose 20% of attempted updates as a middle point, because it corresponds to real life situation in database management where the percentage of successful updates is 10%. (In our testing environment we expect only half of update calls to succeed)

Results. To get meaningful results we average through up to 25 runs. Each run is carried out for 10 seconds with a warmup of 5 seconds. Figure 2a (and resp. 2b) contains the results of executions on Intel (and resp. AMD) machine. It can be seen that with the increase of the size the performance of our algorithm

becomes better relatively to CF-tree. This is due to the fact that with bigger size the cleanup-thread in CF-tree implementation spends more time to clean the tree out of logically deleted vertices, thus, the traversals has more chances to pass over deleted vertices, leading to longer traversals. By this fact and the trend shown, we could assume that CO-tree outperforms CF-tree on bigger sizes. On the other hand, BCCO-tree was much worse on 2^{15} and became similar to CO-tree on 2^{21}. This happened because the races for the locks become more unlikely. This helped much to BCCO-tree, because it uses high-grained locking. Since, our algorithm is "exactly" the same without order of locking, On bigger sizes we could expect that our implementation will continue to perform similarly to CO-tree, because the difference in CO-tree and CF-tree implementations is only in grabbing locks method. By that, we could state that our algorithm works well not depending on the size. As the percentage of contains operations increases, the difference between our algorithm and CF-tree becomes smaller, moreover, our algorithm seems to perform better than other trees.

5 Related Work and Discussion

Measuring concurrency. Measuring concurrency via comparing a concurrent data structure to its sequential counterpart was originally proposed [19]. The metric was later applied to construct a concurrency-optimal linked list [20], and to compare synchronization techniques used for concurrent *search* data structures, organizing nodes in a directed acyclic graph [3]. Although lots of efforts have been devoted to improve the performance of BSTs as under growing concurrency, to our knowledge, the existence of a concurrency-optimal BST has not been earlier addressed.

Concurrent BSTs. The transactional red-black tree [21] uses software transactional memory without sentinel nodes to limit conflicts between concurrent transactions, but restarts the update operation after its rotation aborts. Optimistic synchronization, as seen in transactional memory, was used to implement a practical lock-based BST [15]. The speculation-friendly tree [22] is a partially-external binary search tree that marks internal nodes as logically deleted to reduce conflicts between software transactions. It decouples a structural operation from abstract operations to rebalance when contention disappears. Some red-black trees were optimized for hardware transactional memory and compared with bottom-up and top-down fine-grained locking techniques [23]. The contention-friendly tree [13] is a lock-based partially-external binary search tree that provides lock-free lookups and rebalances when contention disappears. The logical ordering tree [14] combines the lock-free lookup with on-time removal during deletes. The first lock-free tree proposal [16] uses a single-word CAS and does not rebalance. Howley and Jones [24] proposed an internal lock-free binary search tree where each node keeps track of the operation currently modifying it. Chatterjee et al. [25] proposed a lock-free BST, but we are not aware of any implementation. Natarajan and Mittal [17] proposed an efficient lock-free binary search tree implementation that uses edge markers. It outperforms both the

lock-free BSTs from Howley and Jones [24] and Ellen et al. [16]. Since it is not implemented in Java, we could not compare it against ours; however, we know that neither this nor any of the above mentioned BSTs are concurrency-optimal (cf. Figure 1).

Search for concurrency-optimal data structures. Concurrent BSTs have been studied extensively in literature; yet by choosing to focus on minimizing the amount of synchronization, we identified an extremely high-performing concurrent BST implementation. We proved our implementation to be formally correct and established the concurrency-optimality of our algorithm. Apart from the intellectual merit of understanding what it means for an implementation to be highly concurrent, our findings suggest a relation between concurrency-optimality and efficiency. We hope this work will inspire the design of other concurrency-optimal data structures that currently lack efficient implementations.

References

1. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems **13**(1) (1991) 123–149
2. Gramoli, V., Kuznetsov, P., Ravi, S.: In the search for optimal concurrency. In: Structural Information and Communication Complexity - 23rd International Colloquium, SIROCCO 2016, Helsinki, Finland, July 19-21, 2016, Revised Selected Papers. (2016) 143–158
3. Gramoli, V., Kuznetsov, P., Ravi, S.: In the search for optimal concurrency. In: Structural Information and Communication Complexity - 23rd International Colloquium, SIROCCO 2016, Helsinki, Finland, July 19-21, 2016, Revised Selected Papers. (2016) 143–158
4. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26** (1979) 631–653
5. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3) (1990) 463–492
6. Attiya, H., Welch, J.: Distributed Computing. Fundamentals, Simulations, and Advanced Topics. John Wiley & Sons (2004)
7. Chaudhri, V.K., Hadzilacos, V.: Safe locking policies for dynamic databases. J. Comput. Syst. Sci. **57**(3) (1998) 260–271
8. Sutter, H.: Choose concurrency-friendly data structures. Dr. Dobb's Journal (June 2008)
9. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N., Shavit, N.: A lazy concurrent list-based set algorithm. In: OPODIS. (2006) 3–16
10. Herlihy, M., Shavit, N.: On the nature of progress. In: OPODIS. (2011) 313–328
11. Guerraoui, R., Henzinger, T.A., Singh, V.: Permissiveness in transactional memories. In: DISC. (2008) 305–319
12. Kuznetsov, P., Ravi, S.: On the cost of concurrency in transactional memory. In: International Conference on Principles of Distributed Systems (OPODIS). (2011) 112–127
13. Crain, T., Gramoli, V., Raynal, M.: A contention-friendly binary search tree. In: Euro-Par. Volume 8097 of LNCS. (2013) 229–240

14. Drachsler, D., Vechev, M., Yahav, E.: Practical concurrent binary search trees via logical ordering. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '14 (2014) 343–356
15. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. In: PPoPP. (2010)
16. Ellen, F., Fatourou, P., Ruppert, E., van Breugel, F.: Non-blocking binary search trees. In: PODC. (2010) 131–140
17. Natarajan, A., Mittal, N.: Fast concurrent lock-free binary search trees. In: PPoPP. (2014) 317–328
18. Gramoli, V.: More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In: PPoPP. (2015) 1–10
19. Gramoli, V., Kuznetsov, P., Ravi, S.: From sequential to concurrent: correctness and relative efficiency (brief announcement). In: Principles of Distributed Computing (PODC). (2012) 241–242
20. Gramoli, V., Kuznetsov, P., Ravi, S., Shang, D.: A concurrency-optimal list-based set (brief announcement). In: Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9. (2015)
21. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC. (2008)
22. Crain, T., Gramoli, V., Raynal, M.: A speculation-friendly binary search tree. In: PPoPP. (2012) 161–170
23. Siakavaras, D., Nikas, K., Goumas, G., Koziris, N.: Performance analysis of concurrent red-black trees on htm platforms. In: 10th ACM SIGPLAN Workshop on Transactional Computing (Transact). (2015)
24. Howley, S.V., Jones, J.: A non-blocking internal binary search tree. In: SPAA. (2012) 161–171
25. Chatterjee, B., Nguyen, N., Tsigas, P.: Efficient lock-free binary search trees. In: PODC. (2014)

A Proof of correctness

In general, the correctness of the parallel algorithm is carried by the proofs of linearizability and deadlock-freedom. In our paper we add additional constraints on the possible executions of our algorithm: they have to carry the observably correct schedules. We consider the schedule to be observably correct if it satisfies three conditions: the prefix of the schedule is linearizable; at any time the tree is a BST; and the algorithm never links the unlinked node back. This notion could be formally defined as follows.

Definition 1. *A schedule is observably correct if each of its prefixes σ satisfies the following conditions:*

- *subsequence of high-level invocations and responses of operations that made a write in σ has a linearization with respect to the set type;*
- *the data strucure after performing σ is a BST B;*
- *BST after performing σ does no contain a node x such that there exist σ' and σ'', such that σ' is a prefix of σ'', σ'' is a prefix of σ, x is in the BST after σ', and x is not in the BST after σ''.*

The theorem about the correctness of the algorithm could be stated as follows.

Theorem 3. *The algorithm is correct if:*
- *the schedule corresponding to any execution of the algorithm is observably correct.*
- *the algorithm is deadlock-free.*

We split our proof into three parts: the structural properties, i.e., the tree is a BST and an unlinked node cannot be linked back, the linearizability and deadlock-freedom.

A.1 Structural correctness

At first, we prove that our search tree satisfies the structural properties at any point in time, i.e., the second and the third property of observably correctness. Later we refer to these properties as Properties 1, 2, 3 and 4.

Theorem 4. *The following properties are satisfied at any point of time during the execution:*
- *The value property of BST is preserved.*
- *Every routing node has two children.*
- *Any non-physically deleted node is reachable from the root.*
- *Any physically deleted node is non-reachable from the root.*

Proof. The first two properties are non-trivial by themselves, but we could refer to papers [15] and [13] that use the similar partially-external algorithm.

The last two properties follows in a straightforward way from the fact that during physical deletion the algorithm takes locks.

A.2 Linearizability

To prove the linearizability of our algorithm, we need to define the linearization points of *insert*, *delete* and *contains* operations. When defined the linearization points it could be straightforwardly seen that if the execution is linearizable then each prefix of the corresponding schedule is linearizable. So, for us, it will be enough just to prove that any execution is linearizable.

High-level histories and linearizability. A *high-level history* $\widetilde{H}$ of an execution α is the subsequence of α consisting of all invocations and responses of (high-level) operations.

A complete high-level history $\widetilde{H}$ is *linearizable* with respect to an object type τ if there exists a sequential high-level history S equivalent to $\widetilde{H}$ such that

1. $\rightarrow_{\widetilde{H}} \subseteq \rightarrow_S$
2. S is consistent with the sequential specification of type τ.

Now a high-level history $\widetilde{H}$ is linearizable if it can be *completed* (by adding matching responses to a subset of incomplete operations in $\widetilde{H}$ and removing the rest) to a linearizable high-level history.

Completions. We obtain a completion $\tilde{H}$ of history H as follows. The invocation of an incomplete contains operation is discarded. The invocation of an incomplete $\pi = \mathsf{insert}$ operation that has not performed a write at Lines 14, 22 (28) of the Algorithm 2 are discarded; otherwise, π is completed with the response *true*. The invocation of an incomplete $\pi = \mathsf{delete}$ operation that has not performed a write at Lines 47, 60 (64), 77 (82), 94 (102) of the Algorithm 2 is discarded; otherwise, it is completed with the response *true*.

Note, that the described completions correspond to the completions in which the completed operations made at least write of the sequential algorithm.

Linearization points. We obtain a sequential high-level history $\tilde{S}$ equivalent to $\tilde{H}$ by associating a linearization point l_π with each operation π. In some cases, our choice of the linearization point depends on the time interval between the invocation and the response of the execution of π, later referred to as the interval of π. For example, the linearization point of π in the timeline should lie in the interval of π.

Below we specify the linearization point of the operation π depending on its type.

Insert. For $\pi = \mathsf{insert}(v)$ that returns *true*, we have two cases:

1. A node with key v was found in the tree. Then l_π is associated with the write in Line 15 of the Algorithm 2.
2. A node with key v was not found in the tree. Then l_π is associated with the writes in Lines 22 or 28 of the Algorithm 2, depending on whether the inserted node is left or right child.

For $\pi = \mathsf{insert}(v)$ that returns *false*, we have three cases:

1. If there exists a successful $\mathsf{insert}(v)$ whose linearization point lies in the interval of π, then we take the first such $\pi' = \mathsf{insert}(\mathsf{v})$ and linearize right after $l_{\pi'}$.
2. If there exists a successful $\mathsf{delete}(v)$ whose linearization point lies in the interval of π, then we take the first such $\pi' = \mathsf{delete}(\mathsf{v})$ and linearize right before $l_{\pi'}$.
3. Otherwise, l_π is the call point of π.

Delete. For $\pi = \mathsf{delete}(v)$ that returns *true* we have four cases, depending on the number of children of the node with key v, i.e., the node *curr*:

1. *curr* has two children. Then l_π is associated with the write in Line 47 of the Algorithm 2.
2. *curr* has one child. Then l_π is associated between the writes in Line 59 (63) and in Line 60 (64) of the Algorithm 2, depending on whether *curr* is left or right child. The exact position is calculated as what comes last: Line 59 (63) or the last invocation of unsuccessful $\mathsf{insert}(v)$ or $\mathsf{contains}(v)$ that reads the node *curr*.
3. *curr* is a leaf with a data parent. Then l_π is associated between the writes in Line 76 (81) and in Line 77 (82) of the Algorithm 2, depending on whether

curr is left or right child. The exact position is calculated as what comes last: Line 76 (81) or the last invocation of unsuccessful insert(v) or contains(v) that reads the node *curr*.

4. *curr* is a leaf with a routing parent. Then l_π is associated between the writes in Line 93 (101) and in Line 94 (102) of the Algorithm 2, depending on whether *prev* is left or right child. The exact position is calculated as what comes last: Line 93 (101) or the last invocation of unsuccessful insert(v) or contains(v) that reads the node *curr*.

For every π = delete(v) that returns *false*, we have three cases:

1. If there exists a successful delete(v) whose linearization point lies in the interval of π, then we take the first such π' = delete(v) and linearize right after $l_{\pi'}$.
2. If there exists successful insert(v) whose linearization point lies in the interval of π, then we take the first such π' = insert(v) and linearize right before $l_{\pi'}$.
3. Otherwise, l_π is the invocation point of π.

Contains. For π = contains(v) that returns *true*, we have three cases:

1. If there exists successful insert(v) whose linearization point lies in the interval of π, then we take the first such π' = insert(v) and linearize right after $l_{\pi'}$.
2. If there exists successful delete(v) whose linearization point lies in the interval of π, then we take the first such π' = delete(v) and linearize right before $l_{\pi'}$.
3. Otherwise, l_π is the invocation point of π.

For π = contains(v) that returns *false*, we have three cases:

1. If there exists successful delete(v) whose linearization point lies in the interval of π, then we take the first such π' = delete(v) and linearize right after $l_{\pi'}$.
2. If there exists successful insert(v) which linearization point lies in the interval of π, then we take the first such π' = insert(v) and linearize right before $l_{\pi'}$.
3. Otherwise, l_π is the invocation point of π.

To confirm our choice of linearization points, we need an auxiliary lemma.

Lemma 1. *Consider the call π = traverse(v). If BST at the moment of the invocation of π contains the node u with value v and there is no linearization point of successful delete(v) operation in the interval of π, then π returns u.*

Proof. Consider a list $A(u)$ of ancestors of node u: $root = w_1, \ldots, w_{n-1}, w_n = u$ (starting from the root) in BST at the moment of the invocation of π.

Let us prove that at any point of time the child of w_i in the direction of the value v is w_j for some $j > i$. The only way for w_i to change the proper child is to perform a physical deletion on this child. Consider the physical deletions of w_i in their order in execution. In a base case, when no deletions happened, our invariant is satisfied. Suppose, we operated first k deletions and now we consider a deletion of w_j. Let w_i be an ancestor of w_j and w_k be a child of w_j in proper direction. After relinking w_k becomes a child of w_i in proper direction, so the invariant is satisfied for w_i because $i \leq j \leq k$, while the children of other vertices remain unchanged.

Summing up, π starts at *root*, i.e., w_1, and traverse only the vertices from $A(u)$ in strictly increasing order. Thus π eventually reaches u and returns it.

Theorem 5 (Linearizability). *The algorithm is linearizable with respect to the set type.*

Proof. First, we prove the linearizability of the subhistory with only successful insert and delete operations because other operations do not affect the structure of the tree. Then we prove the linearizability of the subhistory with only update operations, i.e., successful and unsuccessful insert(v) and delete(v). And finally, we present the proof for the history with all types of operations.

Successful update functions. Let $\tilde{S}^k_{succ}$ be the prefix of $\tilde{S}$ consisting of the first k complete successful operations insert(v) or delete(v) with respect to their linearization points. We prove by induction on k that the sequence $\tilde{S}^k_{succ}$ is consistent with respect to the *set* type.

The base case $k = 0$, i.e., there are no complete operations, is trivial.

The transition from k to $k + 1$. Suppose that $\tilde{S}^k_{succ}$ is consistent with the *set* type. Let π with argument $v \in \mathbb{Z}$ and its response r_π be the last operation in $\tilde{S}^{k+1}_{succ}$. We want to prove that $\tilde{S}^{k+1}_{succ}$ is consistent with π. For that, we check all possible types of π.

1. $\pi =$ insert(v) returns *true*.
   By induction, it is enough to prove that there are no preceding operation with an argument v or the last preceding operation with an argument v in $\tilde{S}^{k+1}_{succ}$ is delete(v). Suppose the contrary: let the last preceding operation with an argument v be $\pi' =$ insert(v). We need to investigate two cases of insertion: whether π finds the node with value v in the tree or not.
   In the first case, π finds a node u with value v. π' should have inserted or modified u. Otherwise, the BST at l_π would contain two vertices with value v and this fact violates Property 1. If π' has inserted u, then π has no choice but only to read the state of u as data, which is impossible because π is successful. If π' has changed the state of u to data, then π has to read the state of u as data, because the linearization points of π' and π are guarded by the lock on state. This contradicts the fact that π is successful.
   In the second case, π does not find a node with value v. We know that π and π' are both successful. Suppose for a moment that π wants to insert v as a child of node p, while π' inserts v in some other place. Then the tree at l_π has two vertices with value v, violating Property 1. This means, that π and π' both want to insert v as a child of node p. Because $l_{\pi'}$ precedes l_π and these linearization points are guarded by the lock on the corresponding link of p, π' takes a lock first, modifies the link to a child of p and by that forces π to restart. During the second traversal, π finds newly inserted node with value v by Lemma 1 and becomes unsuccessful. The latter contradicts the fact that π is successful.
2. $\pi =$ delete(v) returns *true*.
   By induction it is enough to prove that the preceding operation with an argument v in $\tilde{S}^{k+1}_{succ}$ is insert(v). Suppose the opposite: let the last preceding operation with v be delete(v) or there is no preceding operation with an argument v. If there is no such operation, then π could not find a node with value v, otherwise, another operation should have inserted this node and

consequently its linearization point would have been earlier. Thus in this case, π cannot successfully delete, which contradicts the result of π.
The only remaining possibility is that the previous successful operation is $\pi' = \mathsf{delete}(v)$. Because π is successful, it finds a non-deleted node u with value v. π' should have find the same node u by Lemma 1, otherwise, the BST right before $l_{\pi'}$ would contain two vertices with value v, violating Property 1. So, both π and π' take locks on the state of u to perform an operation. Because $l_{\pi'}$ precedes l_π, π' has taken the lock earlier and set the state of u to routing or marks u as deleted. When π obtains the lock, it could not read state as data and, as a result, cannot delete the node. This contradicts the fact that π is successful.

Update operations. Let $\tilde{S}_m^k$ be the prefix of $\tilde{S}$ consisting of the first k complete operations $\mathsf{insert}(v)$ or $\mathsf{delete}(v)$ with respect to their linearization points. We prove by induction on k that the sequence $\tilde{S}_m^k$ is consistent with respect to the *set* type. We already proved that successful operations are consistent, then we should prove that the linearization points of unsuccessful operations are consistent too.

The base case $k = 0$, i.e., there are no complete operations, is trivial.

The transition from k to $k + 1$. Suppose that $\tilde{S}_m^k$ is consistent with the *set* type. Let π with argument $v \in \mathbb{Z}$ and response r_π be the last operation in $\tilde{S}_m^{k+1}$. We want to prove that $\tilde{S}_m^{k+1}$ is consistent with π. For that, we check all the possible types of π.

If $k + 1$-th operation is successful then it is consistent with the previous operations, because it is consistent with successful operations while unsuccessful operations do not change the structure of the tree.

If $k + 1$-th operation is unsuccessful, we have two cases.

1. $\pi = \mathsf{insert}(v)$ returns *false*. When we set the linearization point of π relying on the successful operation in the interval of π, the linearization point is correct: if we linearize right after successful $\mathsf{insert}(v)$ then π correctly returns *false*; if we linearize right before successful $\pi' = \mathsf{delete}(v)$ then by the proof of linearizability for successful operations there exists successful $\mathsf{insert}(v)$ preceding π', thus π correctly returns *false*.
   It remains to consider the case when no successful operation was linearized in the interval of π. By induction, it is enough to prove that the last preceding successful operation with v in $\tilde{S}_m^{k+1}$ is $\mathsf{insert}(v)$. Suppose the opposite: let the last preceding successful operation with an argument v be $\mathsf{delete}(v)$ or there is no preceding operation with an argument v. If there is no such operation then π could not find a node with value v, because, otherwise, another operation should have inserted the node and its linearization point would have come earlier. Thus π can successfully insert a new node with value v, which contradicts the fact that π is unsuccessful.
   The only remaining possibility is that the last preceding successful operation is $\pi' = \mathsf{delete}(v)$. Since $l_{\pi'}$ does not lie inside the interval of π then π has to find either the routing node with value v or do not find such node, since π' has

unlinked it. In both cases, insert operation could be performed successfully. This contradicts the fact that π is unsuccessful.

2. $\pi = \mathsf{delete}(v)$ returns *false*.
   When we set the linearization point of π relying on the successful operation in the interval of π, the linearization point is correct: if we linearize right after successful $\mathsf{delete}(v)$ then π correctly returns *false*; if we linearize right before successful $\pi' = \mathsf{insert}(v)$ then by the proof of linearizability for successful operations there exists successful $\mathsf{delete}(v)$ preceding π' or there are no successful operation with an argument v in $\tilde{S}_m^{k+1}$ before π', thus π correctly returns *false*.
   It remains to consider the case when no successful operation was linearized in the interval of π. By induction, it is enough to prove that there is no preceding successful operation with v or the last preceding successful operation with v in $\tilde{S}_m^{k+1}$ is $\mathsf{delete}(v)$. Again, suppose the opposite: let the previous successful operation with v be $\pi' = \mathsf{insert}(v)$.
   By Lemma 1 π finds the data node u with value v and π can successfully remove it because no other operation with argument v has a linearization point during the execution of π. This contradicts the fact that π is unsuccessful.

All operations. Finally, we prove the correctness of the linearization points of all operations.

Let $\tilde{S}^k$ be the prefix of $\tilde{S}$ consisting of the first k complete operations ordered by their linearization points. We prove by induction on k that the sequence $\tilde{S}^k$ is consistent with respect to the *set* type. We already proved that update operations are consistent, then we should prove that the linearization points of contains operations are consistent too.

The base case $k = 0$, i.e., there are no complete operations, is trivial.

The transition from k to $k + 1$. Suppose that $\tilde{S}^k$ is consistent with the *set* type. Let π with argument $v \in \mathbb{Z}$ and its response r_π be the last operation in $\tilde{S}^{k+1}$. We want to proof, that $\tilde{S}^{k+1}$ is consistent for the operation π. For that, we check all the possible types of π.

If $k + 1$-th operation is $\mathsf{insert}(v)$ and $\mathsf{delete}(v)$ then it is consistent with the previous $\mathsf{insert}(v)$ and $\mathsf{delete}(v)$ operations while $\mathsf{contains}(v)$ operations do not change the structure of the tree.

If the operation is $\pi = \mathsf{contains}(v)$, we have two cases:

1. π returns *true*.
   When we set the linearization point of π relying on a successful update operation in the interval of π, then the linearization point is correct:
   - if we linearize right after successful $\mathsf{insert}(v)$, then π correctly returns *true*.
   - if we linearize right before successful $\pi' = \mathsf{delete}(v)$, then, by the proof of the linearizability on successful operations, there exists successful $\mathsf{insert}(v)$ preceding π', thus π correctly returns *true*.

   We are left with the case when no successful operation has its linearization point in the interval of π. By induction, it is enough to prove that the last preceding successful operation with v in $\tilde{S}^{k+1}$ is $\mathsf{insert}(v)$. Suppose the oppo-

site: the last preceding successful operation with an argument v is $\mathsf{delete}(v)$ or there is no preceding successful operation with v. If there is no successful operation then π could not find a node with value v, otherwise, some operation has inserted a node before and its linearization point would have come earlier. This contradicts the fact that π is successful.
It remains to check if there exists a preceding $\pi' = \mathsf{delete}(v)$ operation. Since $l_{\pi'}$ does not lie inside the interval of π then π has to find either the routing node with value v or do not find such node, since π' has unlinked it. This contradicts the fact that π returns *true*.

2. π returns *false*.
When we set the linearization point of π relying on a successful update operation in the interval of π, then the linearization point is correct:
   - if we linearize right after successful $\mathsf{delete}(v)$, then π correctly returns *false*;
   - if we linearize right before successful $\pi' = \mathsf{insert}(v)$ then, by the proof of linearizability on successful operations either there exists a preceding π' successful $\mathsf{delete}(v)$ or there exists no operation with an argument v in $\tilde{S}$ before π'. Thus π correctly returns *false*.

   We are left with the case when no successful operation has its linearization point in the interval of π. By induction, it is enough to prove that there is no preceding successful operation with an argument v or the last preceding successful operation with an argument v in $\tilde{S}^{k+1}$ is $\mathsf{delete}(v)$. Again, suppose the opposite: the last preceding successful operation with an argument v is $\pi' = \mathsf{insert}(v)$.
   By Lemma1 π finds the data node u with value v. This contradicts the fact and π should return *false*.

A.3 Deadlock-freedom

Theorem 6 (Deadlock-freedom). *The algorithm is deadlock-free: assuming that no thread fails in the middle of its update operation, at least one live thread makes progress by completing infinitely many operations.*

Proof. A thread executing $\pi = \mathsf{contains}(v)$ makes progress in a finite number of its own steps, because contains is wait-free. Otherwise, take the highest "conflicting" node. Note if some thread t_1 failed to acquire a lock on this node it happens for two reasons:

1. There is another thread t_2 which holds a lock on this node. Since we acquire locks from children to parents and since this is the highest conflicting node, t_2 successfully acquires locks on higher nodes and makes progress.
2. Some locking conditions are violated: it means that between the traversal phase and the attempt to acquire a lock some another thread t_2 changes expected conditions. Thus, thread t_2 has already made progress.

B Proof of concurrency optimality

Theorem 7 (Optimality). *Our binary search tree implementation is concurrency-optimal with respect to the sequential algorithm provided in Algorithm 1.*

Proof. Consider all the executions of our algorithm in which all critical sections are executed sequentially. Since all critical sections in our algorithm contains only one operation from the sequential algorithm, the implementation accepts all the schedules in which the operation is not restarted by failing some condition in the critical sections. So, it is enough to show that each condition that forces the restart is crucial, i.e., if the operation ignores it the schedule will be not observably correct schedule.

For the next discussion we have to define two values $I(T, v)$ and $D(T, v)$ — the number of insert and delete operations with argument v that made at least one write in the prefix with length T of schedule σ, later referred as $\sigma(T)$. Since we consider the linearization of operations that performed write, $I(T, v)$ and $D(T, v)$ are exactly the number of successful insert and delete operations in any completion of $\sigma(T)$. From hereon, when we talk about the completions we mean only operations that performed write.

To slightly simplify the further proof by exhaustion we look at three common situations (later referred to as Case 1, 2 or 3) that appear under consideration, and show that they lead to not observably correct schedule:

1. The modification in the critical section of operation $\pi = \mathsf{insert}(v)$ (the case of $\mathsf{delete}(v)$ is considered similarly) does not change the set of values represented by our tree, i.e., fields left, right and state for any node reachable from the root does not change or some routing vertex becomes unlinked. Let this modification be the T-th event of the current schedule σ. Consider two prefixes of this schedule: $\sigma(T-1)$ and $\sigma(T)$. There could happen two cases:
   - If the value v is present in the set after $\sigma(T-1)$, then $I(T-1, v) = D(T-1, v)+1$, since $\sigma(T-1)$ is linearizable. We know that π is successful, then $I(T, v) = D(T, v) + 2$. By that, any completion of $\sigma(T)$ cannot be linearizable, meaning that σ is not observably correct.
   - If the value v is not present in the set after completion of $\sigma(T-1)$ then $I(T-1, v) = D(T-1, v)$, since $\sigma(T-1)$ is linearizable. We know that π is successful, then $I(T, v) = D(T, v) + 1$, but the value v is still not present in the set after $\sigma(T)$. By that, any completion $\sigma(T)$ cannot be linearizable, meaning that σ is not observably correct.
2. After the modification in the critical section of operation π with argument v a whole subtree of node u with a value different from v becomes unreachable from the root. Let this modification be the T-th event of the current schedule σ. Because of the structure of the tree, subtree of node u should contain at least one data vertex with value x not equal to v. Since x was reachable after the modification and $\sigma(T-1)$ is linearizable, we assume $I(T-1, x) = D(T-1, x) + 1$. The number of successful update operations with argument x does not change after the modification, so $I(T, x) = D(T, x) + 1$. But

the value x is not reachable from the root after $\sigma(T)$, meaning that any completion of $\sigma(T)$ cannot be linearized. Thus, σ is not observably correct.

3. After the modification in the critical section of operation π the node u with deleted mark becomes reachable from the root. Let this modification be the T-th event of the current schedule σ. Let the modification that was done in the same critical section as the deleted mark of u was set to be the $\tilde{T}$-th event of σ. It could be seen that u is reachable from the root after $\sigma(\tilde{T}-1)$ and after $\sigma(T)$, but u is not reachable from the root after $\sigma(\tilde{T})$. Thus, $\sigma(T)$ does not satisfy the third requirement to observably correct schedule, meaning that σ is not observably correct.

Now, we want to prove that all conditions that precede each modification operation are necessary and their omission leads to not observably correct schedule. The proof is done by induction on the position of modification operation in the execution. The base case, when there are no modification operations done, is trivial. Suppose, we show the correctness of our statement for the first $i-1$ modifications and want to prove it for the i-th. Let this modification be the T-th event of the schedule σ. We ignore each condition that precedes the modification one by one in some order and show that their omission makes σ not observably correct:

- Operation $\pi = \mathsf{insert}(v)$ restarts in Line 14 of Algorithm 2. This means, that at least one of the following condition holds:
  - *curr* is not a routing node (Line 14). Then the guarded operation does not change the set of values and by Case 1 σ is not observably correct.
  - Deleted mark of *curr* is set (later, we simply say *curr* is deleted) (Line 14). then *curr* is already unlinked, so the modification in Line 15 does not change the set of values and by Case 1 σ is not observably correct.
- Operation $\pi = \mathsf{insert}(v)$ restarts in Lines 18, 20 (24, 18) of Algorithm 2. This means, that at least one of the following conditions holds:
  - *prev* is deleted (Line 18 (24)). Then *prev* is already unlinked and is not reachable from the root. This means, that the modification in Line 22 (28) links the new vertex to already unlinked vertex *prev*, not changing the set of values, and by Case 1 σ is not observably correct.
  - The corresponding child of *prev* is not null (Line 18 (24)). Then the write in Line 22 (28) unlinks a whole subtree of the current child and by Case 2 σ is not observably correct.
- Operation $\pi = \mathsf{delete}(v)$ restarts in Lines 44, 45 of Algorithm 2. This means that either *curr* is not a data node or *curr* does not have two children.
  - If *curr* is not a data node or it is deleted (Line 44), then the write at Line 47 does not change the set of values and by Case 1 σ is not observably correct.
  - If *curr* does not have two children (Line 45), then after the write in Line 47 the tree has the routing node *curr* with less than two children. Thus, after $\sigma(T)$ the tree does not satisfy the second requirement to observably correct schedule, meaning that σ is not observably correct.

– Operation $\pi = \mathsf{delete}(v)$ restarts in Lines 48-51 of Algorithm 2. This means that at least one of the following conditions holds:
  - *prev* is deleted (Line 49). Then the write at Line 60 (64) does not change the set of values and by Case 1 σ is not observably correct. For later cases, we already assume, that *prev* is not deleted.
  - *child* because *child* is deleted (Line 48). Then after the write at Line 60 (64) the deleted node *curr* becomes reachable from the root, since *prev* is not deleted, and by Case 3 σ is not observably correct. From hereon, we assume that *child* is not deleted.
  - There is no link from *curr* to *child* (Line 48). Since *child* is not deleted, this case could happen only if *curr* is deleted. We know that *prev* and *child* are not deleted, thus *prev* has *child* as its child. By that, the write at Line 60 (64) does not change the set of values and by Case 1 σ is not observably correct. From hereon, we assume that *curr* is not deleted.
  - There is no link from *prev* to *curr* (Line 49), because *curr* is deleted was already covered by the previous case.
  - *curr* is not a data node (Line 50). Then the write in Line 60 (64) does not change the set of values and by Case 1 σ is not observably correct.
  - *curr* does not have exactly one child (Line 51). Since none of *curr* and *child* are deleted, the link from *curr* to *child* exists in the tree, the only possible way to violate is that *curr* has two children. Thus, the write in Line 60 (64) unlinks a whole subtree of the other child of *curr* and by Case 2 σ is not observably correct.

– Operation $\pi = \mathsf{delete}(v)$ restarts in Lines 65-71 and 75 (80) of Algorithm 2. This means that at least one of the following conditions holds:
  - *prev* is deleted (Line 75 (80)). Then, the write in Line 77 (82) does not change the set of values and by Case 1 σ is not observably correct. From hereon, we assume that *prev* is not deleted.
  - *prev* is not a data node (Line 75 (80)). Then after the write in Line 77 (82) the tree contains a routing node with less than two children. Thus, after $\sigma(T)$ the tree does not satisfy our second requirement to observably correct schedule, meaning that σ is not observably correct.
  - The child c of *prev* in the direction of v is null (Line 65). Then the write in Line 77 (82) does not change the set of values and by Case 1 σ is not observably correct.
  - The child c of *prev* in the direction of v has a key different from v (Line 65). (Note that c cannot be deleted since the link from *prev* to c is in the tree now.) The write in Line 77 (82) unlinks a whole subtree of c and by Case 2 σ is not observably correct.
  - The child c of *prev* in the direction of v is not a leaf (Line 71). Then the write in Line 77 (82) removes whole subtree of c with at least one another data node and by Case 2 σ is not observably correct. In last case we assume that c is a leaf.
  - The child c of *prev* in the direction of v is a routing node (Line 70). Then before the write in Line 77 (82) the tree contains a routing leaf c.

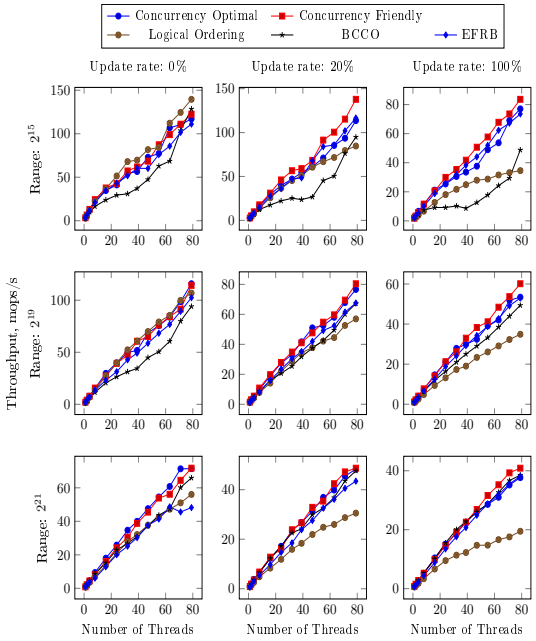
Thus, after $\sigma(T-1)$ the tree does not satisfy our second requirement to observably correct schedule, meaning that σ is not observably correct.

– Operation $\pi = \mathsf{delete}(v)$ restarts in Lines 65-71 and 89-91 (97-99) of Algorithm 2. This means that at least one of the following conditions holds:
  - *gprev* is deleted (Line 90 (98)). Then the write in Line 94 (102) does not change the set of values and by Case 1 σ is not observably correct. From hereon, we assume that *gprev* is not deleted.
  - *prev* is not a child of *gprev*. (Line 90 (98)) Since *gprev* is not deleted, this case could happen only if *prev* is deleted. *prev* could be physically deleted only if it has at most one child, thus *curr* or *child* has to be deleted. If *curr* is deleted, then the write in Line 94 (102) does not change the set of values and by Case 1 σ is not observably correct. Otherwise, *child* is deleted, then the write in Line 94 (102) links the deleted node back to the tree and by Case 3 σ is not observably correct. Later, we assume that *prev* is not deleted.
  - *prev* is a data node (Line 91 (99)). Then the write in Line 94 (102) unlinks *prev* from the tree and by the same reasoning as in Case 2 σ is not observably correct.
  - *child* is not a current child of *prev* (Line 89 (97)). *child* should be deleted, since *prev* is not. Then the write in Line 94 (102) links the deleted node back to the tree and by Case 3 σ is not observably correct.
  - The last four cases are identical to the last four cases for $\mathsf{delete}(v)$ that restarts in Lines 65-71 and 91 (99).

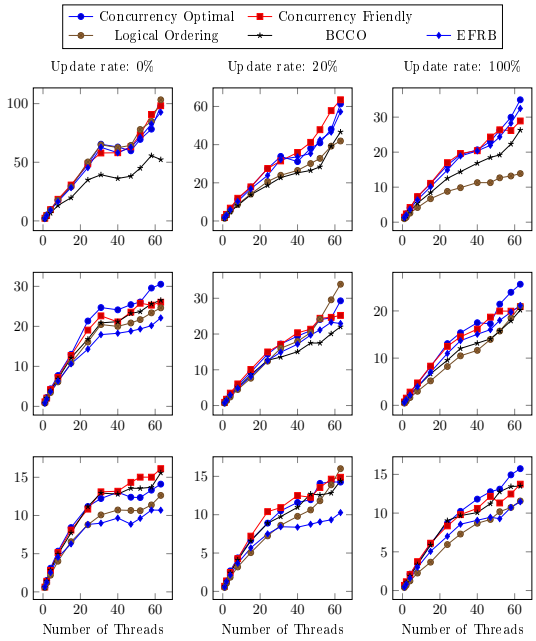
We showed that restart of operation in the execution happens only if the corresponding sequential schedule is not observably correct. Thus, our algorithm is indeed concurrency-optimal.

Algorithm 2 Concurrent implementation.

```
1:  contains(v):
2:    ⟨gprev, prev, curr⟩ ← traversal(v)
3:    return curr ≠ null ∧ curr.state = DATA
4:  insert(v):
5:    ⟨gprev, prev, curr⟩ ← traversal(v)
6:    if curr ≠ null then
7:      go to Line 12
8:    else
9:      go to Line 16
10:   Release all locks
11:   return true
      Update existing node
12:   if curr.state = DATA then
13:     return false
14:   curr.tryWriteLockState(ROUTING)
15:   curr.state ← DATA
      Insert new node
16:   newNode.val ← v
17:   if v < prev.val then
18:     prev.tryLockLeftEdgeRef(null)
19:     prev.slock.tryReadLock()
20:     if prev.deleted then
21:       Restart operation
22:     prev.left ← newNode
23:   else
24:     prev.tryLockRightEdgeRef(null)
25:     prev.slock.tryReadLock()
26:     if prev.deleted then
27:       Restart operation
28:     prev.right ← newNode
29: delete(v):
30:   ⟨gprev, prev, curr⟩ ← traversal(v)
                ▷ All restarts are from this Line
31:   if curr = null ∨ curr.state ≠ DATA then
32:     return false
33:   if curr has exactly 2 children then
34:     go to Line 44
35:   if curr has exactly 1 child then
36:     go to Line 53
37:   if curr is a leaf then
38:     if prev.state = DATA then
39:       go to Line 73
40:     else
41:       go to Line 83
42:   Release all locks
43:   return true
      Delete node with two children
44:   curr.tryWriteLockState(DATA)
45:   if curr does not have 2 children then
46:     Restart operation
47:   curr.state ← ROUTING
      Lock acquisition routine for vertex with one child
48:   curr.tryLockEdgeRef(child)
49:   prev.tryLockEdgeRef(curr)
50:   curr.tryWriteLockState(DATA)
51:   if curr has 0 or 2 children then
52:     Restart operation
      Delete node with one child
53:   if curr.left ≠ null then
54:     child ← curr.left
55:   else
56:     child ← curr.right
57:   if curr.val < prev.val then
58:     perform lock acquisition at Line 48
59:     curr.deleted ← true
60:     prev.left ← child
61:   else
62:     perform lock acquisition at Line 48
63:     curr.deleted ← true
64:     prev.right ← child
      Lock acquisition routine for leaf
65:   prev.tryLockEdgeVal(curr)
66:   if v < prev.key then ▷ get current child
67:     curr ← prev.left
68:   else
69:     curr ← prev.right
70:   curr.tryWriteLockState(DATA)
71:   if curr is not a leaf then
72:     Restart operation
      Delete leaf with DATA parent
73:   if curr.val < prev.val then
74:     perform lock acquisition at Line 65
75:     prev.tryReadLockState(DATA)
76:     curr.deleted ← true
77:     prev.left ← null
78:   else
79:     perform lock acquisition at Line 65
80:     prev.tryReadLockState(DATA)
81:     curr.deleted ← true
82:     prev.right ← null
      Delete leaf with ROUTING parent
83:   if curr.val < prev.val then
84:     child ← prev.right
85:   else
86:     child ← prev.left
87:   if prev is left child of gprev then
88:     perform lock acquisition at Line 65
89:     prev.tryEdgeLockRef(child)
90:     gprev.tryEdgeLockRef(prev)
91:     prev.tryWriteLockState(ROUTING)
92:     prev.deleted ← true
93:     curr.deleted ← true
94:     gprev.left ← child
95:   else
96:     perform lock acquisition at Line 65
97:     prev.tryEdgeLockRef(child)
98:     gprev.tryEdgeLockRef(prev)
99:     prev.tryWriteLockState(ROUTING)
100:    prev.deleted ← true
101:    curr.deleted ← true
102:    gprev.right ← child
```



(a) Evaluation of BST implementations on Intel



(b) Evaluation of BST implementations on AMD

Fig. 2: Performance evaluation of concurrent BSTs