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] University of Southern California, Information Sciences Institute

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" mean 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 updates to the data structure itself, performed as though the operation works

[⋆] 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).

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 how concurrent two different implementations of the same data structure employing different synchronization techniques are, we consider the recently introduced metric of the "amount of concurrency" defined via sets of accepted (*correct*) schedules [2]. A correct schedule, 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* [3]. Combined with the standard correctness criterion of linearizability, the concurrency properties of two correct data structure implementations 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 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 [4], to be "concurrency-friendly" [5]: 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 linearizable concurrent BST-based set implementation. We prove that the implementation is concurrency-optimal with respect to a standard *partially-external* sequential tree [2]. The proposed implementation employs the optimistic "lazy" locking approach [6] 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 a 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* [7]). 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 only if it is not serializable (the property known as *permissiveness*), requires very heavy local computations [8,9]. 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 is either out-performing or is competitive against state-of-the-art BST implementations ([10–13]) on all basic 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, including the sequential implementation of *partially-external* binary search tree and our novel conditional read-write lock abstraction. § 3 formalizes the notion of concurrency-optimality and sketches the relevant proofs; detailed proofs are given in the technical report [14]. § 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 type using partially-external binary search tree. Then, we present our concurrent implementation (Algorithm 2), constructed from the sequential one by adding synchronization primitives.

We begin with a formal specification of the set type. A set object stores a set of integer values, initially empty, and exports operations insert(v), remove(v), contains(v). The update operations, insert(v) and 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* (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. A node without children is called a *leaf*. The order is carried by a *value property*: the value of each node is strictly greater than the values in its left subtree and strictly smaller than the values in its right subtree.

Algorithm 1 Concurrent implementation: node structure and traversal function.

```
Shared Variables:
  node is a record with fields:
  $val$, its value
  $slock$ = $Lock<state>$, a lock that guards its $state$,
  where $state \in \{DATA, ROUTING\}$
  $llock$ = $Lock<left>$, a lock that guards a pointer $left$ to the left child
  $rlock$ = $Lock<right>$, a lock that guards a pointer $right$ to the right child
  $deleted$, a boolean flag indicating the node is logically deleted or not
  Initially the tree contains one node $root$,
  $root.val \leftarrow +\infty$
  $root.slock.init(DATA)$
  $root.llock.init(null)$
  $root.rlock.init(null)$
  $deleted \leftarrow false$

traversal($v$):                          ▷ wait-free traversal from vertex start
  $gprev \leftarrow null; prev \leftarrow null$
  $curr \leftarrow root$                 ▷ starting traversal from root
  while $curr \neq null$ do
    if $curr.val = v$ then
      break
    else
      $gprev \leftarrow prev$
      $prev \leftarrow curr$
      if $curr.val < v$ then
        $curr \leftarrow curr.left$      ▷ go to the left subtree
      else
        $curr \leftarrow curr.right$     ▷ go to the right subtree
  return $\langle gprev, prev, curr \rangle$
```

2.1 Sequential implementation

As for a sequential implementation, we chose the well-known *partially-external* binary search tree. Such a tree combines the idea of an *internal* binary search tree, where the values from all nodes constitute the implemented set, and an *external* binary search tree, where the set is represented by the values in the leaves, and the internal nodes are used for "routing" from the root to the leaves. A partially-external tree supports two types of nodes: *routing* and *data*. To bound the number of routing vertices by the number of data nodes, the tree should satisfy an additional condition that all routing nodes must have exactly two children.

The **traversal** function takes a value v, traverses the tree down from the root respecting the value property, as long as the current node is not null or its value is not v. The function returns the last three visited nodes. The **contains** function takes a value v, 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 with value v is found, the algorithm checks whether the node is data or routing: in the former case, the function returns *false* (v is already in the set); in the latter case, the algorithm simply changes the state of the node from routing to data. If the node with value v is not found, then the algorithm assumes that v is not in the set and inserts a new node with 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 with value v is not found or its state is routing, the algorithm assumes that 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 child, 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

Our concurrency-optimal BST is built on top of the described above sequential implementation using the *optimistic* approach. Traversals are *wait-free* (no synchronization is employed), the locations to be modified are locked and then the reads performed during the traversal are *validated*. If validation fails, the operation is restarted.

Read fields. 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 the "cache" and further accesses return the "cached" value. For example, the reads of the *left* field in Lines 53 and 54 of Algorithm 2 return the same (cached) value.

Deletion 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. At 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) (from there on, 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(exp) 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(exp) 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 but the value inside did not, and the current node is not deleted, i.e., its deletion 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 node is not deleted, i.e., its deletion mark is not set.

These six functions are implemented in the same manner: the function reads necessary fields and a *lock* variable, checks the conditions, if successful it takes a corresponding lock, then checks the conditions again, if unsuccessful it releases the 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 $node.value$.

Concurrency-optimal BST. Above we already described everything needed to write the algorithm. Each node is represented as a union of records (see Algorithm 1): *val*, the value in the node of arbitrary comparable type, *slock*, the lock that guards the state, *llock* and *rlock*, the locks that guard the pointers to left and right children, correspondingly, and *deleted*, the flag of logical deletion. The pseudocode of the concurrent algorithm is provided in Algorithms 1 and 2. To shrink the pseudocode in size we have not put a restart instruction explicitly for all failed "try lock" operations, but it should be read so. The traversal function is identical to the sequential algorithm (see Algorithm 1). The contains has an additional check, whether the node's deleted mark is set or not. In the former case, the function returns *false*. The insert and delete functions largely follow the structure of the sequential code, but take locks on the *modification* part of the tree using the above described read-write conditional lock. Due to space constraints, we refer the reader to the pseudocode of Algorithm 2 for the specifics of the implementation.

3 Concurrency optimality and correctness

In this section, we show that our implementation is *concurrency-optimal* [2]. 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 [2].

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 technical report [14].

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''.

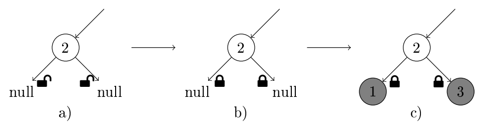
We say that an implementation is *concurrency-optimal* if it accepts *all* observably correct schedules.

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

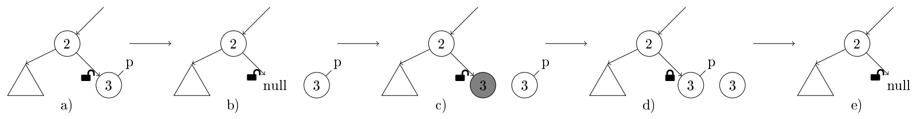
A proof of Theorem 2 is given in the technical report [14]. The intuition behind the proof is the following. We show that for each observably correct schedule there exists a matching execution of our implementation. Therefore, only schedules not observably correct can be rejected by our algorithm. The 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 [10–13] 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



(a) Scenario depicting a concurrent execution of insert(1) and insert(3); rejected by popular BSTs like [10–13], 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 [10–13, 15], it is accepted by a concurrency-optimal BST

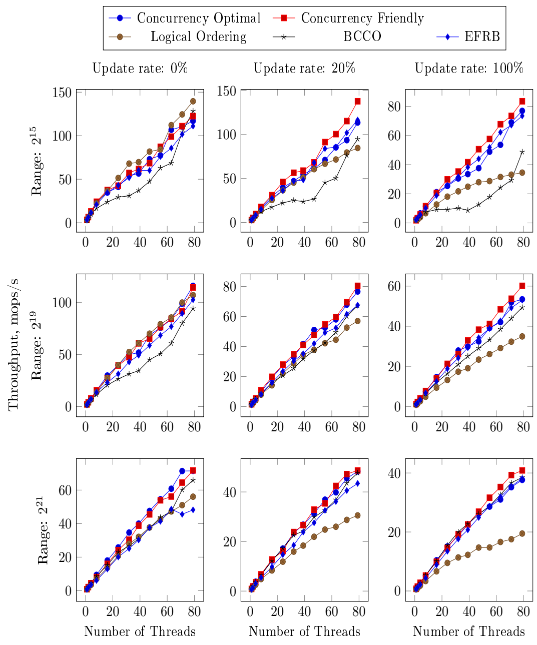
Fig. 1: Examples schedules rejected by concurrent BSTs not concurrency-optimal

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 [10–13, 15]. While, there is an execution of our concurrency-optimal BST that accepts this schedule.

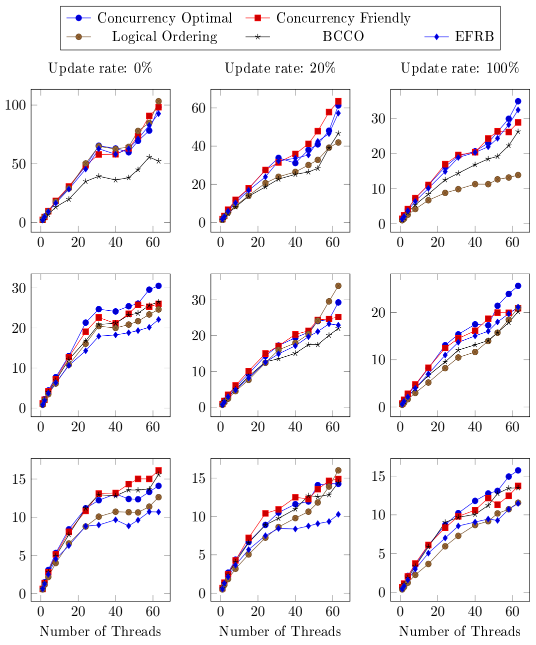
4 Implementation and evaluation

Experimental setup. For our experiments, we used two machines to evaluate our concurrency-optimal 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. The 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. Both machines have 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 Concurrency Friendly (or CF) tree by Crain et al. [10], (2) the lock-based Logical Ordering (or LO) AVL-tree by Drachsler et al. [11], (3) the lock-based BCCO tree by Bronson et al. [12], and (4) the lock-free EFRB tree by Ellen et al. [13]. All these implementations are written in Java and taken from the `synchrobench repository` [16]. In order to make a fair comparison, we remove rotation routines from the CF-, LO- and CO-trees implementations. We are aware of efficient lock-free tree by Natarajan and Mittal [15], but, unfortunately, we are unaware of any implementation in Java.



(a) Evaluation of BST implementations on Intel



(b) Evaluation of BST implementations on AMD

Fig. 2: Performance evaluation of concurrent BSTs

Experimental methodology. For our experiments, we use the environment provided by the Synchrobench library. To compare the performance we considered the following parameters: (i) **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%. (ii) **Tree size.** On the above workloads, 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. (iii) **Degree of contention.** This depends on the number of hardware threads, but we take enough points to reason about the behavior 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. 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 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 the bigger size the cleanup-thread in CF-tree implementation spends more time to clean the tree out of logically deleted vertices, thus, the traversals have more chances to pass over deleted vertices, leading to longer traversals. By this fact and the shown trend, 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 the BCCO-tree, because its locking is coarse-grained. Since, our algorithm is "exactly" the same without the order of locking, on bigger sizes we could expect that CO- and BCCO- trees will perform similarly. We could conclude that our algorithm works well regardless of the size. As the percentage of contains operations increases, the difference between our algorithm and CF-tree becomes smaller and in most workloads, we perform better than other BSTs.

5 Related Work and Discussion

Measuring concurrency. Measuring concurrency via comparing a concurrent data structure to its sequential counterpart was originally proposed [17]. The metric was later applied to construct a concurrency-optimal linked list [18], and to compare synchronization techniques used for concurrent *search* data structures, organizing nodes in a directed acyclic graph [2]. 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 [19] 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 [12]. The speculation-friendly tree [20] 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 [21]. The contention-friendly tree [10] is a lock-based partially-external binary search tree that provides lock-free lookups and rebalances when contention disappears. The logical ordering tree [11] combines the lock-free lookup with on-time removal during deletes. The first lock-free tree proposal [13] uses a single-word CAS and does not rebalance. Howley and Jones [22] proposed an internal lock-free binary search tree where each node keeps track of the operation currently modifying it. Chatterjee et al. [23] proposed a lock-free BST, but we are not aware of any implementation. Natarajan and Mittal [15] proposed an efficient lock-free binary search tree implementation that uses edge markers. It outperforms both the lock-free BSTs from Howley and Jones [22] and Ellen et al. [13]. 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. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26** (1979) 631–653
4. Chaudhri, V.K., Hadzilacos, V.: Safe locking policies for dynamic databases. J. Comput. Syst. Sci. **57**(3) (1998) 260–271
5. Sutter, H.: Choose concurrency-friendly data structures. Dr. Dobb's Journal (June 2008)
6. 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
7. Herlihy, M., Shavit, N.: On the nature of progress. In: OPODIS. (2011) 313–328
8. Guerraoui, R., Henzinger, T.A., Singh, V.: Permissiveness in transactional memories. In: DISC. (2008) 305–319
9. Kuznetsov, P., Ravi, S.: On the cost of concurrency in transactional memory. In: International Conference on Principles of Distributed Systems (OPODIS). (2011) 112–127
10. Crain, T., Gramoli, V., Raynal, M.: A contention-friendly binary search tree. In: Euro-Par. Volume 8097 of LNCS. (2013) 229–240
11. 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
12. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. In: PPoPP. (2010)
13. Ellen, F., Fatourou, P., Ruppert, E., van Breugel, F.: Non-blocking binary search trees. In: PODC. (2010) 131–140
14. Aksenov, V., Gramoli, V., Kuznetsov, P., Malova, A., Ravi, S.: A concurrency-optimal binary search tree. CoRR **abs/1702.04441** (2017)

15. Natarajan, A., Mittal, N.: Fast concurrent lock-free binary search trees. In: PPoPP. (2014) 317–328
16. 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
17. 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
18. 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)
19. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC. (2008)
20. Crain, T., Gramoli, V., Raynal, M.: A speculation-friendly binary search tree. In: PPoPP. (2012) 161–170
21. 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)
22. Howley, S.V., Jones, J.: A non-blocking internal binary search tree. In: SPAA. (2012) 161–171
23. Chatterjee, B., Nguyen, N., Tsigas, P.: Efficient lock-free binary search trees. In: PODC. (2014)

Algorithm 2 Concurrent implementation.

```
1:  contains($v$):
2:     $\langle gprev, prev, curr \rangle \leftarrow traversal(v)$
3:     return $curr \neq null \land curr.state = DATA$

4:  insert($v$):
5:     $\langle gprev, prev, curr \rangle \leftarrow traversal(v)$
6:     if $curr \neq null$ 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:    $curr.state \leftarrow DATA$

    Insert new node
16:    $newNode.val \leftarrow v$
17:    if $v < prev.val$ then
18:       $prev.tryLockLeftEdgeRef(null)$
19:       $prev.slock.tryReadLock()$
20:       if $prev.deleted$ then
21:          Restart operation
22:       $prev.left \leftarrow newNode$
23:    else
24:       $prev.tryLockRightEdgeRef(null)$
25:       $prev.slock.tryReadLock()$
26:       if $prev.deleted$ then
27:          Restart operation
28:       $prev.right \leftarrow newNode$

29: delete($v$):
30:    $\langle gprev, prev, curr \rangle \leftarrow traversal(v)$
                  ▷ All restarts are from this Line
31:    if $curr = null \lor curr.state \neq DATA$ 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:    $curr.state \leftarrow ROUTING$

    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 $curr.left \neq null$ then
54:       $child \leftarrow curr.left$
55:    else
56:       $child \leftarrow curr.right$
57:    if $curr.val < prev.val$ then
58:       perform lock acquisition at Line 48
59:       $curr.deleted \leftarrow true$
60:       $prev.left \leftarrow child$
61:    else
62:       perform lock acquisition at Line 48
63:       $curr.deleted \leftarrow true$
64:       $prev.right \leftarrow child$

    Lock acquisition routine for leaf
65:    $prev.tryLockEdgeVal(curr)$
66:    if $v < prev.key$ then ▷ get current child
67:       $curr \leftarrow prev.left$
68:    else
69:       $curr \leftarrow prev.right$
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:       $curr.deleted \leftarrow true$
77:       $prev.left \leftarrow null$
78:    else
79:       perform lock acquisition at Line 65
80:       $prev.tryReadLockState(DATA)$
81:       $curr.deleted \leftarrow true$
82:       $prev.right \leftarrow null$

    Delete leaf with ROUTING parent
83:    if $curr.val < prev.val$ then
84:       $child \leftarrow prev.right$
85:    else
86:       $child \leftarrow prev.left$
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:       $prev.deleted \leftarrow true$
93:       $curr.deleted \leftarrow true$
94:       $gprev.left \leftarrow child$
95:    else
96:       perform lock acquisition at Line 65
97:       $prev.tryEdgeLockRef(child)$
98:       $gprev.tryEdgeLockRef(prev)$
99:       $prev.tryWriteLockState(ROUTING)$
100:      $prev.deleted \leftarrow true$
101:      $curr.deleted \leftarrow true$
102:      $gprev.right \leftarrow child$
```