Received <day> <Month>, <year>; | Revised <day> <Month>, <year>; | Accepted <day> <Month>, <year>

DOI: xxx/xxxx

RESEARCH ARTICLE

A Speculation-Friendly Binary Search Tree[†]

Tyler Crain*[1] | Vincent Gramoli[1] | Michel Raynal[2,3]

[1]University of Sydney, Sydney. Australia.
[2]Universite Rennes 1, Rennes. France.
[3]Institut Universitaire de France.

Correspondence
*Tyler Crain, Email: tyler.crain@sydney.edu.au

Abstract

We introduce the first concurrent data structure algorithm designed for speculative executions. Prior to this work, concurrent structures were mainly designed for their pessimistic (non-speculative) accesses to have a predictable asymptotic complexity. Researchers tried to evaluate transactional memory using such structures whose prominent example is the red-black tree library developed by Oracle Labs that is part of multiple benchmark distributions. Although well-engineered, such structures remain badly suited for speculative accesses, whose step complexity might raise dramatically with contention.

We propose a binary search tree data structure whose key novelty stems from the *decoupling* of update operations: instead of performing an update operation in a single large transaction, it is split into one transaction that modifies the abstraction state and several other transactions that restructure the tree implementation in the background. This results in a speculation-friendly tree (*s-tree*) that outperforms previous HTM-based and STM-based trees by being transiently unbalanced during contention peaks and by rebalancing in quadratic time when contention disappears. In particular, the s-tree is shown correct, reusable and speeds up a transaction-based travel reservation application by up to $3.5\times$.

KEYWORDS:
Background Rebalancing, Optimistic Concurrency, Transactional Memory, HTM

1 | INTRODUCTION

The multicore era is changing the way we write concurrent programs. In such a context, concurrent data structures are becoming a bottleneck building block of a wide variety of concurrent applications. Generally, these data structures rely on invariants used to upper bound the asymptotic complexity of their accesses. A tree structure with n nodes must, for example, remain sufficiently balanced at any point of a concurrent execution to ensure that its accesses complete in the order of $\log n$ steps.

Speculative synchronization techniques, like transactions[3,4,5], keep gaining in popularity for their ability to simply and efficiently protect concurrent data structures. The GNU compiler collection, *gcc*, has provided transactional constructs since 2012[1], speculative lock elision[6] is hard-coded in the Intel Haswell processor lineup, and one of the greenest and most efficient supercomputer[2], IBM BlueGene/Q, embeds hardware transactional memory[7]. There exist various alternative synchronisation techniques to transactions, like read-copy-update, copy-on-write, various locking mechanisms and lock-free techniques[22]. Unfortunately, implementing efficient binary trees based on this techniques is non-trivial and becomes particularly difficult if the data structure gets used as a library in a larger software as it is the case in database system for example. Composing these data structures may typically lead to deadlocks, or even inconsistencies while transactional data structures compose. A well-known example is the the linux kernel file `mmap.c` that includes 50 lines of commented code to explain how locks should be used or the lock-free tree that cannot rebalance[42].

[†]The short version of this paper appeared in the proceedings of PPoPP 2012[1,2]. The current version extends the other with the proofs of correctness, the complexity analysis as well as new experiments based on Intel's HTM.

[1]http://gcc.gnu.org/wiki/TransactionalMemory

[2]Sources: http://www.top500.org and http://www.green500.org

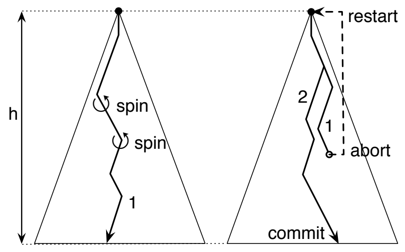


FIGURE 1 A balanced search tree whose complexity, in terms of the amount of accessed elements, is **(left)** proportional to h in a pessimistic execution and **(right)** also proportional to the number of restarts in an optimistic execution

Most existing transactions build upon *optimistic synchronization* that lets a sequence of shared accesses execute speculatively and potentially abort to preserve atomicity of other sequences. They simplify concurrent programming for two reasons. First, the programmer only needs to delimit regions of sequential code into transactions (or to replace critical sections by transactions) to obtain a safe concurrent program. Second, the resulting transactional program is reusable by any programmer, hence a programmer composing operations from a transactional library into other transactions is guaranteed to obtain new operations that execute atomically and are free from deadlocks as they do not block each other. By contrast, *pessimistic synchronization*, where each access to some location x blocks further accesses to x, is harder to program with [8,9] and hampers reusability [10].

Yet it is unclear how one can adapt a data structure to access it efficiently through transactions. To convert a sequential data structure, one would typically encapsulate its operations into transactions. To convert a pessimistically synchronized program, one would simply replace critical sections by transactions. Unfortunately, this often results in highly contended executions whose optimistic accesses may keep aborting and restarting.

To illustrate the difference between optimistic and pessimistic executions, consider the example of Figure 1 depicting their step complexity when traversing a tree of height h from its root to a leaf node. On the left, steps are executed pessimistically, potentially spinning before being able to acquire a lock, on the path converging towards the leaf node. On the right, steps are executed optimistically and some of them may abort and restart, depending on concurrent thread steps. The pessimistic execution of each thread is guaranteed to execute $O(h)$ steps (possibly spinning at times, waiting on locks), yet the optimistic one may need to execute $\Omega(hr)$ steps, where r is the number of restarts. Note that r depends on the probability of conflicts with concurrent transactions that depends, in turn, on the transaction length and h. Consider for example a transactional tree that implements a key-value store abstraction, mapping a unique key to a value, as used by main-memory databases. A committing transaction should not insert a key-value pair $\langle k, v\rangle$ successfully in a store where key k has already been inserted by a concurrent transaction. However, it remains unclear whether a transaction must be aborted when it risks of slightly unbalancing the tree because, afterall, the store abstraction remains consistent.

We introduce a *speculation-friendly* tree (s-tree for short) as a tree that transiently breaks its balance structural invariant without hampering the abstraction consistency in order to speed up transaction-based accesses. Here are our four contributions.

1. We propose an s-tree data structure algorithm that decouples the operations that modify the abstraction (we call these *abstract transactions*) from operations that modify the tree structure itself but not the abstraction (we call these *structural transactions*). We prove the algorithm correct.

2. We compare its performance against existing transaction-based versions of an AVL tree as well as a red-black tree, which has been extensively used to evaluate transactions over the last thirteen years [11,12,13,14,15,16,17,7,18]. The s-tree improves by up to $1.6\times$ the performance of the AVL tree on the micro-benchmark and by up to $3.5\times$ the performance of the built-in red-black tree on a travel reservation application.

3. We illustrate the portability of our s-tree by evaluating it on both Hardware Transactional Memory (HTM) and two Transactional Memories (TMs) libraries, TinySTM [15] and $\mathscr{E}$-STM [19] and on distinct TM configuration settings. We compare this gain against the performance gain coming from the relaxation of the transaction semantics through the use of elastic transactions [19] showing that changing the data structure algorithm can be more beneficial than relaxing the transaction semantics.

4. We present a theoretical analysis that show that our binary search tree becomes well balanced when contention disappears and modifications stop. In particular, we demonstrate that from any reachable state our tree gets rebalanced after only $O(n)$ node-to-node propagations of balance information and $O(n^2)$ local steps of the distributed rotation, where n is the number of nodes.

The paper is organized as follows. In Section 2 we describe the problem related to the use of transactions in existing balanced trees. In Section 3 we present our s-tree and prove that it implements a linearizable key-value store abstraction. In Section 4 we propose an optimized variant of the

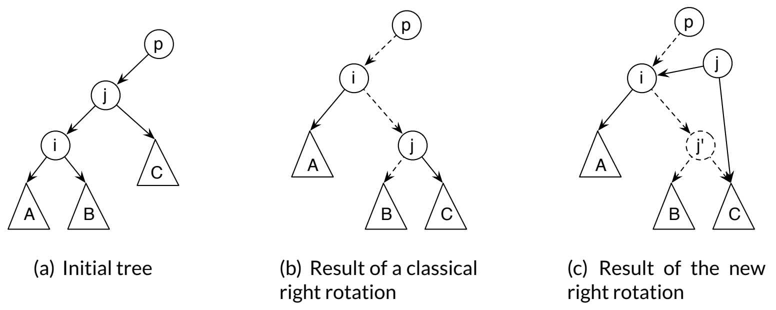

p
j
i
A
B
C
(a) Initial tree
(b) Result of a classical right rotation
j'
(c) Result of the new right rotation

FIGURE 2 The classical rotation modifies node j in the tree and forces a concurrent traversal at this node to backtrack; the new rotation left j unmodified, adds j′ and postpones the physical deletion of j

algorithm and show it correct. In Section 5 we give an upper-bound on the amount of rotations and propagations needed to rebalance the tree. In Section 6 we analyze experimentally our tree and illustrate its portability and reusability. In Section 7 we describe the related work and Section 8 concludes.

2 | THE PROBLEM WITH BALANCED TREES

In this section we focus on the structural invariant of existing binary tree algorithms, namely their *balance*, and outline the impact of their restructuring, namely the *rebalancing*, on contention.

Binary trees provide logarithmic access time complexity given that they are balanced, meaning that among all downward paths from the root to a leaf, the length of the shortest path is not far apart the length of the longest path. Upon tree update, if their difference exceeds a given threshold, the balance invariant is broken and a rebalancing is triggered to restructure accordingly. This threshold depends on the considered algorithm: AVL trees [20] do not tolerate the longest path to exceed the shortest by two edges whereas red-black trees [21] tolerate the longest to be twice the shortest, thus restructuring less frequently. Yet in both cases the restructuring is triggered immediately when the threshold is reached to hide the imbalance from further operations.

Generally when writing an algorithm using transactions, one takes an existing tree algorithm and encapsulates each access within a transaction to obtain a concurrent tree whose accesses are guaranteed atomic. The obtained concurrent transactions *conflict* (i.e., one accesses the same location another is modifying) especially when one updates the topmost region of the tree, resulting in the need to abort one of these transactions and wasting efforts. Most of these conflicts are actually due to the fact that encapsulating an *update* operation (i.e., a successful insert or remove operation) into a transaction boils down to encapsulating four phases in the same transaction:

1. the modification of the abstraction,
2. the corresponding structural adaptation,
3. a check to detect whether the threshold is reached and
4. the potential rebalancing.

A transaction-based red-black tree. An example is the transaction-based binary search tree developed by Oracle Labs (formerly Sun Microsystems) and other researchers, and that was extensively used to evaluate transactional memories [11,12,13,14,15,16,17,7,18]. This library relies on the classical red-black tree algorithm that bounds the step complexity of pessimistic insert/delete/contains. It has been slightly optimized for transactions by removing sentinel nodes to reduce false-conflicts, and we are aware of two benchmark-suite distributions that integrate it, STAMP [13] and Synchrobench [22].

Each of its update transactions encapsulate all four of the phases given above even though phase (1) could be decoupled from phases (3) and (4) if transient violations of the balance invariant were tolerated. Such a decoupling is appealing given that phase (4) is subject to conflicts. In fact in phase (4), the algorithm balances the tree by executing rotations starting from the position where a node is inserted or deleted and possibly going all the way up to the root. As depicted in Figures 2(a) and (b), a rotation consists of replacing the node where the rotation occurs by the child and adding this replaced node to one of its subtrees. The drawback is that these modifications create conflicts with other transactions, for example, two rotations cannot access common nodes as one rotation may unbalance the other. Moreover a node cannot even be accessed concurrently by a transaction that encapsulates the operation that modifies the abstraction (also called *abstract transaction*) and a rotation (we refer to a rotating transaction as a *structural transaction*), otherwise the abstract transaction might miss the node it targets while being rotated downward, leading to inconsistencies.

Moreover, the red-black tree does not allow any abstract transaction to access a node that is concurrently being deleted because phases (1) and (2) are tightly coupled within the same transaction. If this was allowed the abstract transaction might end up on the node that is no longer part of the tree. Fortunately, if the modification operation is a deletion then phase (1) can be decoupled from the structural modification of phase (2) by simply marking the targeted node as logically deleted in phase (1), effectively removing it from the abstraction prior to unlinking it physically in phase (2). This improvement is important as it lets a concurrent abstract transaction travel through the node that is concurrently being deleted in phase (1) without conflicting. Making things worse due to the "all or nothing" semantics of transactions that either commit entirely or abort, having to abort within phase (4) would typically require the three previous phases to restart as well. Finally, only contains operations are guaranteed not to conflict with each other. With the decoupling of these phases, insert/delete/contains do not conflict with each other unless they terminate on the same node as we detail in Section 3.

More specifically, for the transactions to preserve the atomicity and invariants of such a tree algorithm, they typically have to keep track of a large *read set* and *write set*, i.e., the sets of accessed memory locations that are protected by a transaction. Possessing large read/write sets increases the probability of conflicts and thus reduces concurrency. This is especially problematic in trees because the distribution of nodes in the read/write sets is skewed so that the probability of the node being in the set is much higher for nodes near the root and the root is guaranteed to be in the read set.

Illustration. To briefly illustrate the effect of tightly coupling update operations on the step complexity of classical transactional balanced trees we have counted the maximal number of reads necessary to complete typical insert/remove/contains operations. Note that this number includes the reads executed by the transaction each time it aborts in addition to the read set size of the transaction obtained at commit time.

We evaluated the aforementioned red-black tree, an AVL tree, and our s-tree on a 48-core AMD machine using the same TM library[3]. The expectation of the tree sizes is fixed to 2^{12} during the experiments by performing a successful insert and a successful remove with the same probability. Table 1 depicts the maximum number of transactional reads per operation run by 48 concurrent threads as we increase the update ratio, i.e., the proportion of insert/remove operations over contains operations.

For all three trees, the transactional read complexity of an operation increases with the update ratio due to the additional aborted efforts induced by the contention. Although the red-black and the AVL trees objective is to keep the asymptotic complexity of pessimistic accesses $O(\log_2 n)$ (proportional to 12 in this case), where n is the tree size, the read complexity of optimistic accesses grows significantly ($14\times$ more at 10% update than at 0%, where there are no aborts) as the contention increases. As described in the following, the s-tree succeeds in limiting the step complexity raise ($2.6\times$ more at 10% update) of data structure accesses when compared against the transactional versions of state-of-the-art tree algorithms.

3 | THE SPECULATION-FRIENDLY SEARCH TREE

We introduce the speculation-friendly binary search tree (s-tree for short) by describing its implementation of a key-value store, mapping a key to a value. In short, the tree speeds up the access transactions by decoupling two conflict-prone operations: the node deletion and the tree rotation. Although these two decouplings have been used for decades in the context of data management[23,24], our algorithm novelty lies in applying their combination to reduce transaction aborts. We first depict, in Algorithm 1, the pseudocode that looks like sequential code encapsulated within transactions before presenting, in Algorithm 2, more complex optimizations.

For the sake of simplicity our key-value store abstraction simply provides the code of insert, delete and contains operations from which get-key or update-key can easily be derived. The *sequential specification* of the store object is defined as follows. The contains operation takes as an argument a key k and returns true only if any key-value pair $\langle k, *\rangle$ was present in the store. Operation insert takes as an argument a key k and a value v, and adds the corresponding pair $\langle k, v\rangle$ and returns true if no key k is already present in the store, otherwise it returns false. The delete operation takes as an argument a key k, removes the associated key-value pair $\langle k, *\rangle$ from the store and returns true if a key-value pair $\langle k, *\rangle$ is present in the store, otherwise it returns false. In the remainder we use the dotted notation to refer to any field *flag* of a specific tree node i as *i.flag*.

Decoupling the tree rotation. The motivation for rotation decoupling stems from two separate observations: (i) a rotation is tied to the modification that triggers it, hence the process modifying the tree is also responsible for ensuring that its modification does not break the balance invariant and (ii) a rotation affects different parts of the tree, hence an isolated conflict can abort the rotation performed at multiple nodes. In response to these two issues we introduce a dedicated rotator thread responsible for performing structural adaptations, allowing an abstract transaction to complete faster and we distribute the rotations into multiple local transactions, each consisting of a node-local rotation. Note that our rotator thread is similar to the collector thread proposed by Dijkstra et al.[23] to garbage collect stale nodes.

This decoupling allows the read set of the insert/delete operations to only contain the path from the root to the node(s) being modified and the write set to only contain the single node that needs to be modified in order to ensure the modification of the abstraction (i.e., the node at the bottom of the search path), thus reducing conflicts significantly. Let us consider a specific example. If rotations are performed within the insert/delete operations then each rotation increases the read and write set sizes. Take an insert operation that triggers a right rotation such as the one depicted

[3]TinySTM-CTL, i.e., with lazy acquirement[15].

Algorithm 1 A Portable Speculation-Friendly Binary Search Tree by Process p

```
State of node i:
  node a record with fields:
    $k \in \mathbb{N}$, the node key
    $v \in \mathbb{N}$, the node value
    $\ell, r \in \mathbb{N}$, left/right child pointers,
      initially $\bot$
    $left\text{-}h, right\text{-}h \in \mathbb{N}$, local height of
      left/right child, initially 0
    $local\text{-}h \in \mathbb{N}$, expected local height,
      initially 1
    $del \in \{true, false\}$, indicate whether
      logically deleted, initially false

State of process p:
  root, shared pointer to root

find(k)_p:
  $next \leftarrow root$
  while true do
    $curr \leftarrow next$
    $val \leftarrow curr.k$
    if $val = k$ then break
    if $val >$ k then $next \leftarrow$ read($curr.r$)
    else $next \leftarrow$ read($curr.\ell$)
    if $next = \bot$ then break
  end while
  return curr

contains(k)_p:
  transaction {
    $result \leftarrow$ true
    $curr \leftarrow$ find(k)
    if $curr.k \neq$ k then $result \leftarrow$ false
    else if read($curr.del$) then
      $result \leftarrow$ false
  } // current transaction tries to commit
  return result

insert(k,v)_p:
  transaction {
    $result \leftarrow$ true
    $curr \leftarrow$ find(k)
    if $curr.k = k$ then
      if read($curr.del$) then
        write($curr.del$, false)
      else $result \leftarrow$ false
    else // allocate a new node
      $new.k \leftarrow k$
      $new.v \leftarrow v$
      if $curr.k > k$ then
        write($curr.r, new$)
      else write($curr.\ell, new$)
  } // current transaction tries to commit
  return result

right_rotate(parent, left-child)_p:
  transaction {
    if left-child then $j \leftarrow$ read($parent.\ell$)
    else $j \leftarrow$ read($parent.r$)
    if $j = \bot$ then return false
    $\ell \leftarrow$ read($j.\ell$)
    if $\ell = \bot$ then return false
    $\ell r \leftarrow$ read($\ell.r$)
    write($j.\ell, \ell r$)
    write($\ell.r, j$)
    if left-child then write($parent.\ell, \ell$)
    else write($parent.r, \ell$)
    update-balance-values()
  } // current transaction tries to commit
  return true

delete(k)_p:
  transaction {
    $result \leftarrow$ true
    $curr \leftarrow$ find(k)
    if $curr.k \neq$ k then
      $result \leftarrow$ false
    else
      if read($curr.del$) then
        $result \leftarrow$ false
      else write($curr.del$, true)
  } // current transaction tries to commit
  return result

remove(parent, left-child)_p:
  transaction {
    if left-child then
      $j \leftarrow$ read($parent.\ell$)
    else $j \leftarrow$ read($parent.r$)
    if ($j = \bot \vee \neg$read($j.del$)) then
      return false
    if ($child \leftarrow$ read($j.\ell$)) $\neq \bot$ then
      if read($j.r$) $\neq \bot$ then return false
    else $child \leftarrow$ read($n.r$)
    if left-child then
      write($parent.\ell, child$)
    else write($parent.r, child$)
    update-balance-values()
  } // current transaction tries to commit
  return true
```

in Figures 2(a)-2(b). Before the rotation the read set for the nodes p, j, i is $\{p.\ell, j.r\}$, where ℓ and r represent the left and right pointers, and the write set is $\emptyset$. Now with the rotation the read set becomes $\{p.\ell, i.r, j.\ell, j.r\}$ and the write set becomes $\{p.\ell, i.r, j.\ell\}$ as denoted in the figure by dashed arrows, creating a conflict with any concurrent traversal. In the worst case insert/delete operations trigger rotations all the way up to the root, resulting in conflicts with all concurrent transactions. Below we present our distributed rotation that copes with this issue.

Rotation. As previously described, rotations are not required to ensure the atomicity of the insert/delete/contains operations so it is not necessary to perform them in the same transaction as the insert or delete. Instead we dedicate a separate thread, the rotator thread, to continuously check for imbalances and rotate accordingly within its own node-local transactions.

More specifically, neither do the insert/delete operations comprise any rotation, nor do the rotations execute on a large block of nodes. Local rotations that occur near the root can still cause a large amount of conflicts, but rotations performed further down the tree are less subject to conflict. If local rotations are performed in a single transaction block then even the rotations that occur further down the tree will be part of a likely conflicting transaction, so instead we perform each local rotation as a single transaction. Keeping the insert/delete/contains and rotate/remove transactions small (in terms of the number of their shared accesses) allows more operations to execute concurrently without conflicts, diminishing contention. Moreover, local rotations are able to use the most up to date balance information for each rotation, for example a concurrent insert and delete might mean that a rotation is no longer necessary, thus possibly avoiding repeating rotations at the same location. The actual code for the (local) rotation is straightforward. Each rotation is performed just as it would be performed in a sequential binary tree (see Figure 2(a)-2(b)), but within a separate transaction.

Deciding when to perform a rotation is done based on local balance information omitted from the pseudocode. This technique was introduced by[25] and works as follows. *left-h* (resp. *right-h*) is a node-local variable to keep track of the estimated height of the left (resp. right) subtree. *local-h* (also a node-local variable) is always 1 larger than the maximum value of *left-h* and *right-h*. If the difference between *left-h* and *right-h* is greater than 1 then a rotation is triggered. After the rotation these values are updated accordingly based on the type of rotation performed as indicated by a dedicated function (line 63). Since these values are local to the node the estimated heights of the subtrees might not always be accurate. The propagate operation (described in the next paragraph) is used to update the estimated heights. Using the propagate operation and local rotations, the tree is guaranteed to be eventually perfectly balanced as shown in Section 5.

Propagation. The rotator thread executes continuously a depth-first traversal to propagate the balance information. Although it might propagate outdated height information due to concurrency, in the absence of concurrent modifications, the accurate information gets propagated in linear time (cf. Section 5). The only requirement to ensure balance when using propagations and local rotations is that a node i knows when it has an empty subtree (i.e., when $i.\ell$ is $\perp$, $i.\mathit{left\text{-}h}$ must be 0). This requirement is guaranteed by the fact that a new node is always added to the tree with *left-h* and *right-h* set to 0 and that these values are updated when a node is removed or a rotation takes place. Each propagate operation is performed as a sequence of distributed transactions each acting on a single node. Such a transaction first travels to the left and right child nodes, checking their *local-h* values and using these values to update *left-h*, *right-h*, and *local-h* of the parent node. As a single rotator thread is used and no abstract transactions access these three values, they never conflict with propagate operations meaning their synchronization is not necessary.

Limitations. Unfortunately, spreading rotations and modifications into distinct transactions still does not allow insert/delete/contains operations that are being performed on separate keys to execute concurrently. Consider a delete operation that deletes a node at the root. In order to remove this node a successor is taken from the bottom of the tree so that it becomes the new root. This now creates a point of contention at the root and where the successor was removed. Every concurrent transaction that accesses the tree will have a read/write conflict with this transaction. Below we discuss how to address this issue.

Decoupling the node deletion. The s-tree exploits logical deletion to further reduce the amount of transaction conflicts. This two-phase deletion technique has been previously used for memory management like in[24], for example, to reduce locking in database indexes, or more recently to obtain lock-free[26] or lazy linked lists[27]. Each node has a *del* flag, initialized to false when the node is inserted into the tree. First, the delete phase consists of removing the given key k from the abstraction—it logically deletes a node by setting a *del* flag to true (line 75). Second, the remove phase physically removes the node from the tree to prevent the tree from growing too large. Each of these are performed as a separate transaction. The rotator thread, which is responsible of propagating information, also takes care of garbage collecting nodes as explained in Section 4.4.

The deletion decoupling reduces conflicts by two means. First, it spreads out the two deletion phases in two separate transactions, hence reducing the size of the delete transaction. Second, deleting logically node i simply consists in setting the *del* flag to true (line 75), thus avoiding conflicts with concurrent abstract transactions that have traversed i as the *del* flag is not accessed during traversal.

3.1 | Find

The find procedure is a helper function called implicitly by other functions within a transaction, thus it is never called explicitly by the application programmer. This procedure looks for a given key k by parsing the tree similarly to a sequential code. At each node it goes right if the key of the node is larger than k (line 21), otherwise it goes left (line 22). Starting from the root it continues until it either finds a node with k (line 20) or until it reaches a leaf (line 23), returning that node (line 25). Notice that if it reaches a leaf, it has performed a transactional read on the child pointer of this leaf (lines 21–22), which if it returns $\perp$ ensures that some other concurrent transaction will not insert a node with key k.

3.2 | Contains

The contains operation first executes the find procedure starting from the root, which returns a node (line 29). If the key of the node returned is equal to the key being searched for, then it performs a transactional read of the *del* flag (line 32). If the flag is false the operation returns true, otherwise it returns false. If the key of the returned node is not equal to the key being searched for then a node with the key being searched for is not in the tree and thus, false is returned (lines 30 and 34).

3.3 | Insertion

The insert(k, v) operation uses the find procedure that returns a node (line 38). If a node is found with the key k then the *del* flag is checked using a transactional read (line 40). If the flag is false then the tree already contains k and false is returned (lines 42 and 50). If the flag is true then the flag is updated to false (line 41) and true is returned. Otherwise if the key of the node returned is not equal to k then a new node is allocated and linked to the tree by updating the appropriate child pointer of the node returned by the find operation (lines 43-48). Notice that only in this final case does the operation modify the structure of the tree.

3.4 | Logical deletion

The delete uses also the find procedure in order to locate the node to be deleted (line 69). If the find procedure does not return a node with the same key as the one being searched for then false is returned (lines 71 and 77). Otherwise, a transactional read is then performed on the *del* flag (line 73). If *del* is true then the operation returns false (lines 74 and 77), if *del* is false it is set to true (line 75) and the operation returns true. Notice that this operation never modifies the tree structure.

Consequently, the insert/delete/contains operations can only conflict with each other in two cases:

1. Two insert/delete/contains operations are being performed concurrently on some key k and a node with key k exists in the tree. If at least one of the operations is an insert or delete, then there will be a read/write conflict on the node's *del* flag. Note that there will be no conflict with any other concurrent operation that is being done on a different key.

2. An insert is performed for some key k where no node with key k exists in the tree. Here the insert operation will add a new node to the tree, and will have a read/write conflict with any operation that had read the now updated pointer when it was $\perp$ (i.e., before it was changed to point to the new node).

3.5 | Physical removal

Removing a node that has no children is as simple as unlinking the node from its parent (lines 89–90). Removing a node that has 1 child is done by just unlinking it from its parent, then linking its parent to its child (also lines 89–90). Note that each of these removal procedures is a very small transaction, only performing a single transactional write. Therefore, this transaction conflicts only with concurrent transactions that read the link from the parent before it is changed.

In a classical binary tree, upon removal of a node i with two children, the node in the tree with the immediately larger key than i's must be found at the bottom of the tree. Doing this performs reads all the way to the leaf in addition to a write at the parent of i, creating a conflict with any operation that has traversed this node. Fortunately, in practice such removals are not necessary. In fact in the s-tree only nodes with no more than one child are removed from the tree (if the node has two children, the remove operation returns without doing anything, cf. line 86). It turns out that removing nodes with no more than one child is enough to keep the tree from growing so large that it affects performance as seen in Section 6.

The removal operation is performed by the rotator thread. While it is traversing the tree performing right-rotate/left-rotate and propogate operations it also checks for logically deleted nodes to be removed.

3.6 | Limitations

The traversal phase of most functions is prone to "false" conflicts, as it comprises read operations that do not actually need to return values from the same snapshot. For example by the time a traversal transaction reaches a leaf, the value it read at the root likely no longer matters, thus a conflict with a concurrent root update could simply be ignored. Nevertheless, the standard TM interface forces all transactions to adopt the same strongest semantics prone to false-conflicts [28]. In Section 4 we discuss how to extend the basic TM interface to cope with such false-conflicts.

3.7 | Correctness

In this section we sketch a proof showing that the s-tree provides a linearizable key-value store abstraction.

A *binary search tree* (BST) is a connected acyclic graph made up of nodes with unique keys. Each node has a left and right child, which can either be a node or $\perp$. A node has exactly one parent, except for the root which has none. The subtree created by the left (resp. right) child of a node contains nodes with keys larger (resp. smaller) than the node's key. Note that by definition, given a tree T and a key k, there is exactly one location in this tree

Algorithm 2 Optimizations to the Speculation-Friendly Binary Search Tree by Process p

```
State of node i:
  $node$ the same record with an extra field:
    $rem \in \{$true, true_left_rot, false$\}$
      indicate whether physically
      removed (and by left rotation),
      initially false

remove($parent, left\text{-}child$)$_p$:
  transaction {
    if read($parent.rem$) then
      return false
    if $left\text{-}child$ then
      $j \leftarrow$ read($parent.\ell$)
    else
      $j \leftarrow$ read($parent.r$)
    if ($j = \perp \vee \neg$read($j.deleted$)) then
      return false
    if ($child \leftarrow$ read($j.\ell$)) $\neq \perp$ then
      if read($j.r$) $\neq \perp$ then
        return false
    else
      $child \leftarrow$ read($j.r$)
    if $left\text{-}child$ then
      write($parent.\ell, child$)
    else
      write($parent.r, child$)
    write($j.\ell, parent$)
    write($j.r, parent$)
    write($j.rem$, true)
    update-balance-values()
  } // current transaction tries to commit
  return true

find(k)$_p$:
  $curr \leftarrow root$; $next \leftarrow root$; $rem \leftarrow$ true
  while true do
    while true do
      $parent \leftarrow curr$
      $curr \leftarrow next$
      $val \leftarrow curr.k$
      if $val = k$ then
        $rem \leftarrow$ read($curr.rem$)
        if $\neg rem$ then
          break
      if $val >$ k then
        $next \leftarrow$ uread($curr.r$)
      else if $rem =$ true_left_rot then
        $next \leftarrow$ uread($curr.r$)
      else $next \leftarrow$ uread($curr.\ell$)
      if $next = \perp$ then
        $rem \leftarrow$ read($curr.rem$)
        if $\neg rem$ then
          if $val >$ k then
            $next \leftarrow$ read($curr.r$)
          else $next \leftarrow$ read($curr.\ell$)
          if $next = \perp$ then break
        else
          if $rem =$ true_left_rot then
            $next \leftarrow$ uread($curr.r$)
          else $next \leftarrow$ uread($curr.\ell$)
    end while
    if $curr.k > parent.k$ then
      $tmp \leftarrow$ read($parent.r$)
    else $tmp \leftarrow$ read($parent.\ell$)
    if $curr = tmp$ then
      break
    else
      $next \leftarrow curr$
      $curr \leftarrow parent$
  end while
  return $curr$

right_rotate($parent, left\text{-}child$)$_p$:
  transaction {
    if read($parent.rem$) then
      return false
    if $left\text{-}child$ then
      $j \leftarrow$ read($parent.\ell$)
    else
      $j \leftarrow$ read($parent.r$)
    if $j = \perp$ then
      return false
    $\ell \leftarrow$ read($j.\ell$)
    if $\ell = \perp$ then
      return false
    $\ell r \leftarrow$ read($\ell.r$)
    $r \leftarrow$ read($j.r$)
    // allocate a new node
    $new.k \leftarrow j.k$
    $new.\ell \leftarrow \ell r$
    $new.r \leftarrow r$
    write($\ell.r, new$)
    write($j.rem$, true) // or true_left_rot
    if $left\text{-}child$ then
      write($parent.\ell, \ell$)
    else
      write($parent.r, \ell$)
    update-balance-values()
  } // current transaction tries to commit
  return true
```

where k can exist, we call this valid_point(T, k). If there is a node π in T with key k then valid_point(T, k) is π, otherwise valid_point(T, k) is the node ρ in T that has no child where a node with key k would exist.

A *standard traversal* of a BST T for a key k starts at $root$ and results in valid_point(T, k). More precisely, when traversing a node π in T it follows π's left child if $\pi.\mathrm{k} < \mathrm{k}$ or π's right child if $\pi.\mathrm{k} > \mathrm{k}$.

As most of the proofs are based on induction of the number of operations performed on the tree, for simplifying the base case, the tree is initialized with a single node with $\mathrm{k} = \infty$, thus all nodes will be located on its left subtree.

Except where otherwise mentioned we only consider the right_rotate operation, and not the left_rotate as they are mirrors of each other.

To prove the algorithm correct, first we show that any sequence of operations results in a BST, then we use this result to show the linearizablity of the abstraction.

Lemma 1. Any execution of contains, insert, delete, remove, right_rotate operations performed by the s-tree results in a BST.

Proof. The structure of the tree can be modified by the insert, remove and right_rotate operations. By induction and given that the transactions are opaque[29], each operation is performed on a BST up to an operation x, so we need to show that the after operation $x+1$ the resulting tree is a BST. By lines 23 and 43 the insert operation only modifies the tree by inserting a new node where a node had $\perp$ as a child. Given that the find operation follows a standard traversal (lines 21-22), the new node is inserted at $\text{valid\_point}(T,k)$, resulting in a BST. By lines 83-90, the remove operation only modifies the tree by unlinking a node j with 0 or 1 child, linking j's parent to its child in the later case. By induction the empty or non-empty subtree rooted at j must be a BST, so the resulting tree remains a BST. Here the rotations performed are the same as the rotations from the original AVL tree[20] that do not violate the properties of the BST. □

Theorem 1. Any execution of contains, insert, delete, remove, right_rotate operations performed on the s-tree implements a linearizable key-value store abstraction.

Proof. The STM ensures these operations are performed within opaque transactions, and from Lemma 1, they are executed on a BST, thus to complete the proof we need to show that the store is only modified in the following ways: After an $\text{insert}(k)$ operation, k is in the store (leaving the rest of the store unmodified) and after a $\text{delete}(k)$ operation, k is not in the store (leaving the rest of the store unmodified).

First note that all three operations use the find operation by following a standard traversal (lines 21-22), thus reaching $\text{valid\_point}(T,k)$ for any BST T.

Before looking at the insert and delete, note that for the contains operation, all keys with nodes in the BST that have $\textit{deleted} = \text{false}$ are in the store and all other keys are not in the store. This is guaranteed by following a standard traversal followed by checking the *deleted* flag (line 40), and returning true if a node with key k was found, otherwise returning false (lines 30 and 32).

For the insert operation, if there is a node with key k already in the BST, then the *deleted* flag is set to false (line 41), otherwise a new node is added (line 43), resulting in k being in the store.

For the delete operation, if there is a node with key k already in the BST, then its *deleted* flag is set to true (line 75), otherwise no modification to the tree is made, ensuring that k is not in the store.

Finally, given that the remove, right_rotate do not modify the state of the store, the result follows. □

4 | OPTIONAL IMPROVEMENTS

In previous sections, we have described a s-tree that fulfills the standard TM interface[30] for the sake of portability across a large body of research work on TM. Now, we propose to further reduce aborts related to the rotation and the find procedure (i.e., the traversal) at the cost of an additional lightweight read operation, uread, that breaks this interface. We call this tree the optimized speculation-friendly binary search tree (or opt-s-tree for short). This optimization complementing Algorithm 1 is depicted in Algorithm 2, it does not affect the existing contains/insert/delete operations besides speeding up their internal find operation. Although the left_rotate operation is symmetric to the right_rotate operation, the find must distinguish a node removed by a left_rotate from a node removed by a right_rotate, a subtlety discussed in Section 4.1.

Lightweight reads. The key idea is to avoid validating superfluous read accesses when an operation traverses the tree structure. This idea has been exploited by elastic transactions that use a bounded buffer instead of a read set to validate only immediately preceding reads, thus implementing a form of hand-over-hand locking transaction for search structure[19].

The current distribution of TinySTM[15] comprises unit loads that do not record anything in the read set; we have chosen to use these unit loads, denoted by uread, to implement these optimizations. These uread are used throughout the algorithm in place of certain transactional reads. They typically return the most recent value written to memory by a committed transaction by potentially spinning by reading the value and version until it stops being concurrently modified (meaning that the version matches the value), adding nothing to the transaction's read set. It should be noted that we could also have used different extensions to implement these optimizations such as the early release operation of DSTM[11] which forces a transaction to stop keeping track of a read set entry.

By using these optimizations the read/write set sizes can be kept at a size of $O(k)$ instead of $O(k \log n)$ obtained with the previous tree algorithm, where k is the number of contains/insert/delete operations nested in a transaction. The reasoning behind this is as follows. A contains only needs to ensure that the node it found (resp. did not find) is still in (resp. not in) the tree when the transaction commits, and can ignore the state of other nodes it had traversed (no need to have these $O(\log n)$ nodes per contains/insert/delete in the read set). In a similar vein, insert and delete only need to validate the position in the tree where they aimed at inserting or deleting. Therefore, contains/insert/delete only increases the size of the read/write sets by a constant instead of a logarithmic amount, and as there are k such operations we have $O(k)$ instead of $O(k \log n)$. This optimization helps reducing even further the maximum number of reads per operation as presented in Table 1. In particular, it leads to obtaining only 3 bookkeeped reads maximum per operation (for 10% updates) and 18 bookkeeped reads maximum (for 50% updates).

Removed flag. The optimization to the algorithm requires that each node has an additional flag indicating whether or not the node has been physically removed from the tree (a node is physically removed during a successful rotate or remove operation). This removed flag can be set to false, true or true_left_rot and is initialized to false. For the sake of simplicity of the pseudocode true_left_rot is considered to be equivalent to true, only on line 138 of the find operation is this parameter value specifically checked for.

4.1 | Rotation

Rotations remain conflict-prone in Algorithm 1 as they incur a conflict when crossing the region of the tree traversed by a concurrent contains/insert/delete operation. If ureads are used in the contains/insert/delete operations then rotations only conflict with these operations if they finish at one of the two nodes that are moved by the right-rotate/left-rotate operations (for example in Figure 2(a) this would be the node i or j). A rotation at the root only conflicts with a contains/insert/delete that finished at (or at the rotated child of) the root, any operation that travels further down the tree does not conflict.

In order to ensure the linearizability of our abstraction with the optimizations we used a modified rotation operation. Figure 2(c) displays the result of the new rotation that is different from the previous one. Instead of modifying j directly, j is unlinked from its parent (effectively removing it from the tree, lines 185–187) and a new node j′ is created (line 178), taking j's place in the tree (lines 185–187). During the rotation, j has a removed flag that is set to true (line 183), letting concurrent operations know that j is no longer in the tree but its deallocation is postponed. To understand why this modification is necessary, consider a concurrent operation that is traversing the tree and is preempted on j during the rotation. If a normal rotation is performed then the concurrent operation will either have to backtrack or the transaction would have to abort (as the node it is searching for might be in the subtree A). Using the new rotation, the preempted operation still has a path to A.

As previously mentioned the *rem* flag can be set to one of three values (false, true or true_left_rot). Only when a node is removed during a left rotation is the flag set to true_left_rot. This is necessary to ensure that the find operation follows the correct path in the specific case that the operation is preempted on a node that is concurrently removed by a left rotation and this node has the same key k as the one being searched for or if the left child is $\perp$. In this case the find operation must travel to the right child of the removed node otherwise it might miss the node with key k that has replaced the removed node from the rotation due to the fact that the left child of the removed node is rotated upwards during a right rotation. In all other cases the find operation can follow the child pointer as normal.

4.2 | Find, contains and delete

The interesting point for the find operation is that the search continues until it finds a node with the *rem* flag set to false (lines 134 and 143). Once the leaf or a node with the same key as the one being searched for is reached, a transactional read is performed on the *rem* flag to ensure that the node is not removed from the tree (by some other operation) at least until the transaction commits. If *rem* is true then the operation continues traversing the tree, otherwise the correct node has been found. Next, if the node is a leaf, a transactional read must be performed on the appropriate child pointer to ensure that this node remains a leaf throughout the transaction (lines 145 and 146). If this read does not return $\perp$ then the operation continues traversing the tree. Otherwise the operation then leaves the nested while loop (lines 135 and 147), but the find operation does not return yet.

One additional transactional read must be performed to ensure safety. This is the read of the parent's pointer to the node about to be returned (lines 154 and 155). If this read does not return the same node as found previously, the find operation continues parsing the tree starting from the parent (lines 159 and 160). Otherwise the process leaves the while loop (line 157) and the node is returned (line 162).

4.3 | Removal

The remove operation also requires some modifications to ensure safety when using ureads during the traversal phase. Normally if a contains/insert/delete operation is preempted on a node that is removed then that operation must backtrack or abort the transaction. This can be avoided as follows. When a node is removed, its left and right child pointers are set to point to its previous parent (lines 119 and 120). This provides a preempted operation with a path back to the tree. The removed node also has its *rem* flag set to true (line 121) letting preempted operations know it is no longer in the tree (the node is left to be freed later by garbage collection).

4.4 | Garbage collection

As explained previously, there is always a single rotator thread that continuously executes a recursive depth first traversal. It updates the local, left and right heights of each node and performs a rotation or removal if necessary, garbage collecting removed nodes. Given that an application thread can be preempted on a removed node, a node's memory cannot be freed immediately after removal. Instead, nodes that are successfully removed are then added to a garbage collection list, which are later freed safely in epochs as follows. Each application thread maintains a boolean indicating a pending operation and a counter indicating the number of completed operations. Before starting a recursive traversal, the rotator thread sets a pointer to what is currently the end of the garbage collection list and copies all booleans and counters. After a traversal, if for every thread its counter has increased or if its boolean is false then the nodes up to the previously stored end pointer can be safely freed. Experimentally, we found that the size of the list was never larger than a small fraction of the size of the tree.

4.5 | Correctness

In this section we sketch a proof showing that opt-s-tree provides a linearizable key-value store abstraction. We use the same preliminary definitions as the proof of the s-tree in Section 3.7, with the following additions.

We refer to the *range* of a node i in a BST T, denoted by $\mathsf{range}(i, T)$, as the maximal inclusive interval to which all the keys of the subtree rooted in i belong and to which all other keys do not belong. For example, the range of the root node *root* is $\mathsf{range}(r, T) = [-\infty, \infty]$. If the root has key k then its left child ℓ has $\mathsf{range}(\ell, T) = [-\infty, k]$ while its right child r has $\mathsf{range}(r, T) = [k, \infty]$. Given the structure of a BST we then have the following:

Definition 1. A node in a BST has a larger range than any of its descendants.

We now sketch the proof of opt-s-tree. The structure created by opt-s-tree is made up of nodes that are part of the BST and those that have been removed from the BST, i.e., with $rem = \mathsf{true}$ and not reachable from any node in the BST, but not yet garbage collected. We use the notation T to represent the BST and the notation T^{all} to represent the entire structure. To shorten the proof we borrow some observations from the lemmas of s-tree, but to do this we rely on the fact that operations of s-tree and opt-s-tree modify the BST in the same way, i.e., after performing a modification in opt-s-tree the resulting nodes with $rem = \mathsf{false}$ create the same BST as they would if the same modification was performed by s-tree. This means that if opt-s-tree and s-tree perform the same operations in the same order at the same locations, then the resulting BST T is the same. Note that the only difference is the addition of the nodes with $rem = \mathsf{false}$ in opt-s-tree that are not part of the BST.

The correctness of the opt-s-tree is more involved mainly due to the fact that the find operation (which is used by the contains, insert, delete operations) traverses the tree using ureads. Thus, we have to show that a traversal reaches $\mathsf{valid\_point}(k, T^{\mathsf{all}})$ even if the ureads observe earlier versions of the tree T^{all}. We start by showing that the the traversal is correct if executed on a sequence of BSTs.

Lemma 2. If we are given a sequence of BSTs $T_1, \ldots, T_x$ (and corresponding $T^{\mathsf{all}}_1, \ldots, T^{\mathsf{all}}_x$) created by executing some sequence of contains, insert, delete, remove, right_rotate operations of opt-s-tree where the BSTs are equivalent to an execution of the same sequence of operations by s-tree, then a $\mathsf{find}(k)$ operation whose transactional reads are linearized on T^{all}_x returns $\mathsf{valid\_point}(k, T^{\mathsf{all}}_x)$.

Proof. By Lemma 1 we know that T_1 to T_x (created by update transactions 1 to $x - 1$) are BSTs. We also know that these BSTs must be made up of only nodes with $rem = \mathsf{false}$ as a node can have $rem = \mathsf{true}$ only when it is no longer in the BST due to a remove (lines 110-121) or a right-rotate operation (lines 182-187).

The traversal progresses through the structure by performing ureads on the left or right pointer of a node to reach its child node (lines 137, 139, and 140). Given the guarantees of the uread operation, the first uread observes a structure T^{all}_y where $0 \leq y \leq x$ and each subsequent uread will observe the same or a later structure in the history.

For the traversal to reach $\mathsf{valid\_point}(k, T_x)$, it must only pass through nodes which have k in their range. Thus, let us now show that k will always be in the range of any node reached during the traversal. For this, first we show that no node ever has any value removed from its range, i.e. given a node p at some time $t \leq x$ (where p was inserted into the tree at or after t) with range $r = \mathsf{range}(p, T^{\mathsf{all}}_t)$, then at a later time t1, $t \leq t1 \leq x$, we have $r \in \mathsf{range}(p, T^{\mathsf{all}}_{t1})$.

Let us first consider the nodes with $\mathsf{rem} = \mathsf{false}$, for these nodes we know that the modifications performed by opt-s-tree are the same as s-tree and are part of a BST. Now by Definition 1 a node can only have its range decreased if it is moved downwards in the BST. A node in opt-s-tree can only be moved downwards during a rotation (note this is true for any AVL tree), but in opt-s-tree the node that is rotated downwards is removed from the tree (lines 182-183, see Figure 2). Thus the range of the nodes within the BST are never decreased.

Now consider the nodes with $\mathsf{rem} = \mathsf{true}$, i.e. those removed from the BST at some point. These nodes can be removed from the tree during a remove or a right_rotate. During a remove, the removed node's child pointers are both set to be its parent (lines 119-120), which is higher up in the BST and by Definition 1, has a larger range. During a right_rotate the removed node j's child pointers are not changed, thus are pointing to nodes in the BST, which we already know never have values removed from their range. This holds recursively if a node with $\mathsf{rem} = \mathsf{true}$ has its descendants removed from the tree over time as we have shown that no modification will decrease the range of any of its children.

Now that we know that a node's range is never decreased, we return to the proof that the traversal reaches $\mathsf{valid\_point}(k, T_x)$. A traversal starts by definition at *root* which has $\mathsf{range}(root, T_y^{all}) = [-\infty, \infty]$. For the lemma to be correct, when performing a uread to move from one node to its child in a structure T_y^{all} (where $y \leq x$) the traversal must move to a node *next* for which $k \in \mathsf{range}(next, T_y^{all})$. When traversing from one node in the BST to another node in the BST (the nodes with $\mathsf{rem} = \mathsf{false}$ in T_y^{all}) this is ensured by using the standard traversal.

Otherwise, when traversing through a node that has been removed by a remove operation, the left or right path can be taken as the children pointers both point to the node's previous parent, which we know has a larger range. Now consider the right_rotate operation. After the removal of node j, its right subtree remains unchanged, and the range of the left child is increased as now *new* and the entire right subtree are reachable (see Figure 2). Given this, the standard traversal is unsafe only in the case where $j.k = k$ (as k is only in the range of the left subtree) or in the case where the right subtree is $\perp$, in these cases traversal must go left. This is ensured by checking the value of j.k (line 132) and of *rem* using a transactional read (line 133 or 143) before traversing to the next node.

As soon as the traversal reaches a node with key k or a $\perp$ pointer it performs a transactional read (line 133, 142, 145, or 146), and as a result the STM ensures the tree T_x^{all} is observed (i.e. the transaction will now observe a fixed state of the tree), and the traversal will eventually reach $\mathsf{valid\_point}(k, T_x)$. Note that starvation is possible on in the presence of infinite concurrent modifications unless the STM guarantees starvation freedom. □

We now know that if we are given a sequence of BSTs created by opt-s-tree, these trees are traversed correctly. Let us thus show that opt-s-tree produces such a sequence of BSTs.

Lemma 3. Any execution of contains, insert, delete, remove, right_rotate operations performed by opt-s-tree results in a BST made up of nodes with $rem = \mathsf{false}$.

Proof. We proceed by induction on the number x of modifying transactions performed. Given this we start with a BST T_x and show that after the next modifying transaction, the resulting tree T_{x+1} will also be a BST.

The structure of the tree can be modified by the insert, remove and right_rotate operations. First note that all modifications are performed within transactions, and only on nodes that have $rem = \mathsf{false}$ (lines 103, 134, 143, and 166).

The remove operation unlinks a node, set its flag $rem = \mathsf{true}$ (line 121), and links its non-$\perp$ child (if there is one) to its parent (lines 116-118). Given that T_x is a BST, the removed node's children must also be part of the BST (all having $rem = \mathsf{false}$ given by induction) thus the remaining resulting tree remains a BST. The right-rotate unlinks node j from the tree (lines 185-187) while setting $rem = \mathsf{true}$ (or $rem = \mathsf{true\_by\_left\_rot}$, line 183). In its place it links in a new node *new* as the child of node ℓ (line 182). Node *new*'s left child is then set to the right child of ℓ before the rotation (line 180), and its right child is set to the right child of node *j* (line 181). By induction we had a BST before the rotation, and after the rotation the nodes remaining in the tree create the same structure as a traditional AVL rotation (all with $rem = \mathsf{false}$), thus, the resulting tree is a BST.

For an $\mathsf{insert}(k)$ operation that modifies T_x by inserting a new node (lines 47-48), the modification must be performed at $\mathsf{valid\_point}(T_x, k)$ for T_{x+1} to be a BST. By induction we have $T_1, \ldots, T_X$ are BSTs, thus lemma 2 ensures that $\mathsf{valid\_point}(T_x, k)$ is returned by find. Note that before find returns, it uses the STM's guarantee of opacity to ensure that its result remains correct by performing transactional reads on the node at $\mathsf{valid\_point}(T_x, k)$. Now given that the code executed by the $\mathsf{insert}(k)$ is the same as that executed by the (non-optimized) s-tree, then the proof holds by the same argument given in Lemma 1. □

Theorem 2. Any sequence of contains, insert, delete operations performed on the opt-s-tree implements a linearizable key-value store abstraction.

Proof. Using Lemmas 2 and 3, we can use the same arguments here as in Theorem 1 to complete the proof. □

5 | COMPLEXITY ANALYSIS

We analyze the time it takes for the s-tree to rebalance itself in an asynchronous model. To this end, we consider Algorithm 1 without any of the presented optimizations. Note that these results should still hold for opt-s-tree given that modifications/rotations in either algorithm result in the same binary search tree structure.

Bougé et al.[25] showed that $O(n^2)$ successful rotations and propagations are sufficient to balance a height-relaxed search tree (defined below) of n elements. Here we prove that (1) this result also holds for the s-tree and more specifically that (2) only $O(n)$ successful propagations are necessary in our case.

5.1 | Preliminary definitions

First-of-all, let the *steady state* of a s-tree be any state of the tree in which all formerly executed insert and remove have completed (if any) and no further ones get invoked. This state allows us to bound the time it takes to rebalance the tree when no interfering modifications occur.

We redefine the height relaxed search tree to which the known quadratic bound by Bougé et al.[25] applies using the terminology used in Algorithm 1.

Definition 2 (Height-relaxed search tree)**.** A *height-relaxed search tree* is a search tree whose nodes are equipped with two local fields *left-h* and *right-h* such that: $i.\textit{left-h} = 0$ (resp. $i.\textit{right-h} = 0$) for any node i with an empty left (resp. right) subtree.

Let $i.\textit{local-h}$ be the apparent local height of node i, $i.\textit{local-h} = 1 + \max(i.\textit{left-h}, i.\textit{right-h})$. Let c_i be the absolute difference of knowledge between the child i and its parent p:[4]

$$c_i = \begin{cases} |p.\textit{left-h} - i.\textit{local-h}|, & \text{if } i \text{ is the left child, } p.\ell = i \text{, or} \\ |p.\textit{right-h} - i.\textit{local-h}|, & \text{if } i \text{ is the right child, } p.r = i. \end{cases}$$

Let c be the maximal absolute difference for any child i, $\mathsf{c} = \mathrm{argmax}_{\forall i}\{c_i\}$. Let b_i be the absolute difference between the left height and the right height of node i, $b_i = |i.\textit{left-h} - i.\textit{right-h}|$. Let b be the maximal absolute difference between the left height and the right height for any node i, $\mathsf{b} = \mathrm{argmax}_{\forall i}\{b_i\}$.

We can now restate the worst convergence time result of Bougé et al.[25].

Lemma 4. Let τ be a height-relaxed search tree in any initial state. At most $6cn(n+1) + 3bn$ propagations/rotations on τ are necessary to obtain a balanced tree.

5.2 | The time complexity of rebalancing

Here we show that our tree is a height-relaxed search tree to conclude that its rebalancing takes quadratic time before showing that its propagation completes in linear time.

Lemma 5. A s-tree in a steady state is a height-relaxed search tree.

Proof. We show that the property of Definition 2 holds for Algorithm 1 when no insert/delete operations occur. A new node can happen to have an empty child if it was inserted as a leaf and no left or right child was inserted since then, or if one of its child is removed. First, consider the case where a node is inserted as a leaf. For any new node i, both its *left-h* and *right-h* are initially 0 as depicted at line 8. Hence, a new leaf node i is such that $i.\textit{left-h} = i.\textit{right-h} = 0$ at the time it gets inserted in the tree. When a new left (resp. right) child is added to this node, then only the corresponding $i.\textit{left-h}$ (resp. $i.\textit{right-h}$) gets updated, hence the result still holds when the insertion completes. If a child of node i is removed, then its left height *left-h* and right height *right-h* are accordingly updated before the end (line 63 guaranteeing that the property holds when the removal completes). In addition, when a rotation is performed the balance heights are also updated accordingly (line 188 of right_rotate) ensuring that the correct heights are exchanged and updated. □

Another difference between our model and the one of Bougé et al.[25] is that our algorithm is free from interferences. After our tree reaches a steady state, there is only one thread updating the tree. By contrast, the model of Bougé et al. requires multiple threads to rotate/propagate at different locations, so that one propagation (at the top of the tree) could be a waste of effort as another propagation (lower in the tree) may induce further changes higher up anyway. As a result, our tree simply needs a linear number of propagations while theirs requires a quadratic number of propagation.

Lemma 6. Let τ be a s-tree in a steady state, and let i be any node of τ. The left height $i.\textit{left-h}$ and right height $i.\textit{right-h}$ of node i is correct after n propagations, where n is the size of the tree.

Proof. The proof follows directly from the depth first search executed by the rotator thread. It is easy to observe that the correct imbalance ratio can be propagated recursively from the leaves to the root because we know by Lemma 5 that leaves maintain accurate information. Hence, once every node is visited once by the depth first search algorithm, which takes n node-to-node propagations, the accurate information is propagated to all nodes. □

A direct consequence of using ordered correct propagations indicates intuitively that only $O(n)$ ordered local rotations would also be sufficient to obtain a balanced tree in a recursive manner, however, a rotation at node i alters the depth of the subtree rooted in i.

[4] c_i is not defined if i is not a child, hence $c_{root} = \perp$.

Theorem 3. Let τ be a s-tree in a steady state. After $O(n)$ propagations and $O(n^2)$ rotations on τ we obtained a balanced tree.

Proof. By Lemma 6 the left-height and right-height are propagated to all nodes, so that no more left-height and right-height updates get propagated after n propagations. Then the result follows from the conjunction of Lemmas 4 and 5. □

As seen by the number of local rotations, it is unclear whether using local information for rotations, as we do here, is the most efficient way to balance the tree. Instead, one could think of using a global information, for example, to find the node with the key whose value is the median of all keys, rotating this node to make it the root, and repeating the process to its subtrees by following a top down approach. Such a process may require only $O(n)$ rotations to balance the tree, yet in the case of concurrent insert and delete operations, it might not be as efficient as our approach. The reason for this is simply that, by the time a node gets rotated to the root, concurrent operations might result in its key being far from the median. Two interesting observations are (1) that our s-tree is strictly balanced, so that the length difference between the longest and the shortest path is at most 1, and (2) that the tree may actually effectively rebalance itself even in a non-steady state by exploiting local heights.

6 | EXPERIMENTAL ANALYSIS

We evaluated our library in C by integrating it in (1) the Synchrobench micro-benchmark suite[22] to get a precise understanding of the performance causes and (2) the tree-based vacation reservation application of the STAMP suite[13] and whose runs sometimes exceed half an hour.

The machines used for our experiments are (1) a four AMD Opteron 12-core processor 6172 at 2.1 Ghz with 32 GB of RAM thus comprising 48 cores in total and (2) a 4-core 3.2GHz Intel Core i5-6500 machine with 16 GB of RAM. Unless specified otherwise we use the AMD machine for the experiments.

6.1 | Testbed choices

We evaluate our tree against well-engineered tree algorithms especially dedicated to transactional workloads. The red-black tree is a mature implementation developed and improved by expert programmers from Oracle Labs and others to show good performance of TM over the last decade[11,12,13,14,15,16,17,7]. The observed performance in these papers is that this tree is generally scalable when contention is low. To avoid the attempted update ratio that captures the number of calls to potentially updating operations (including non-updating ones) to be misleading, we consider the *effective* update ratios of synchrobench counting only modifications and ignoring the operations that fail (e.g., remove may fail in finding its parameter value thus failing in modifying the data structure).

As mentioned before one of the main refactoring of this red-black tree implementation is to avoid the use of sentinel nodes that would produce false-conflicts within transactions. This improvement could be considered a first step towards obtaining a speculation-friendly binary search tree, however, the modification and restructuring, which remain tightly coupled, prevent scalability to high levels of parallelism. The AVL tree we evaluate (as well as the aforementioned red-black tree) is the library implementation provided as part of STAMP.

To evaluate performance we ran the micro-benchmark and the vacation application with 1, 2, 4, 8, 16, 24, 32, 40, 48 application threads. The rotator thread is created at the start of each benchmark. For the micro-benchmark, we averaged the data over three runs of 10 seconds each (note that each of these executions commits about 4.10^7 transactions). For the vacation application, we averaged the data over three runs as well but we used the recommended default settings and some runs exceeded half an hour because of the amount of transactions used. We carefully verified that the variance was sufficiently low for the results to be meaningful.

6.2 | Biased workloads and restructuring effects

In this section, we evaluate the performance of our s-tree on an integer set micro-benchmark providing remove, insert, and contains operations, similarly to the benchmarks used to evaluate state-of-the-art TM algorithms[15,31,19]. We implemented two set libraries that we added to the synchrobench distribution: (1) our non-opt-s-tree and (2) a baseline tree that is similar but never rebalances the structure whatever modifications occur. Figure 3 depicts the performance obtained from four different binary search trees: the red-black tree (RBtree), our s-tree without optimizations (SFtree), the no-restructuring tree (NRtree) and the AVL tree (AVLtree).

The performance is expressed as the number of operations executed per microsecond. The update ratio varies between 5% and 20%. As we obtained similar results with 2^{10}, 2^{12} and 2^{14} elements, we only report the results obtained from an initialized set of 2^{12} elements. We tested the algorithms using two workloads: (1) The normal workload in which elements are inserted (resp. deleted) by selecting values uniformly at random from the value range. (2) The biased workload consists of inserting (resp. deleting) random values skewed towards high (resp. low) numbers in the

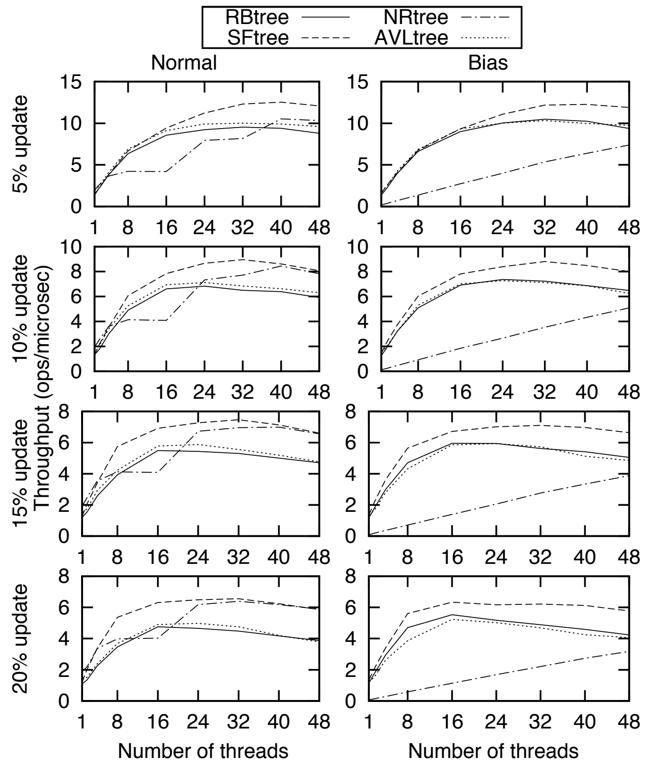


FIGURE 3 Comparing the AVL tree (AVLtree), the red-black tree (RBtree), the no-restructuring tree (NRtree) against the s-tree (SFtree) on an integer set micro-benchmark with from 5% **(top)** to 20% updates **(bottom)** under normal **(left)** and biased **(right)** workloads

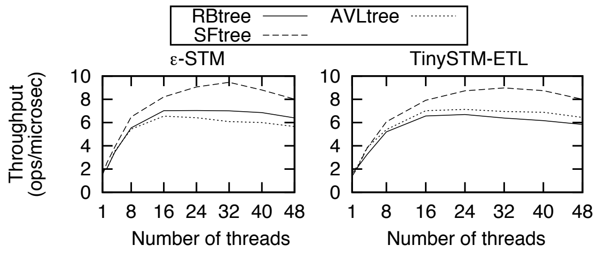
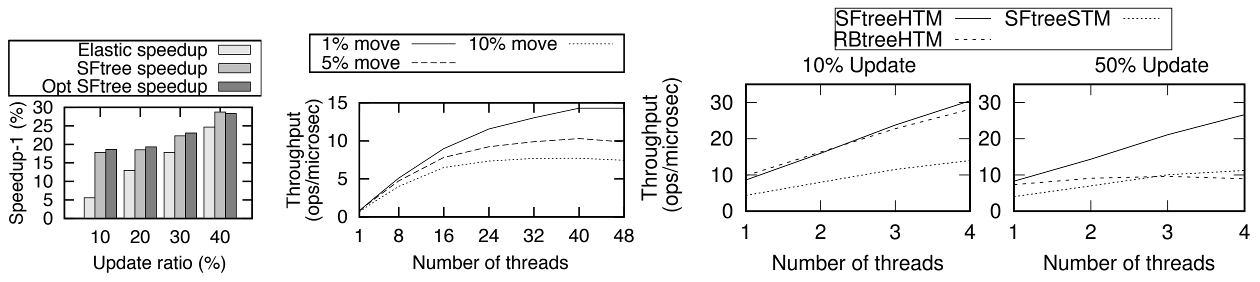


FIGURE 4 The speculation-friendly library running with **(left)** another TM library ($\mathscr{E}$-STM) and with **(right)** the previous TM library in a different configuration (TinySTM-ETL, i.e., with eager acquirement)

value range: the values always taken from a range of 2^{14} are skewed with a fixed probability by incrementing (resp. decrementing) with an integer uniformly taken within $[0..9]$.

On both the normal (uniformly distributed) and biased workloads, the s-tree scales well up to 32/40 threads. The no-restructuring tree performance drops to a linear shape under the biased workload as expected: as it does not rebalance, the complexity increases with the length of the longest path from the root to a leaf that, in turn, increases with the number of performed updates. Under biased workloads this non-restructuring tree tends to converge into a linked list shape which reduces the contention and translates into a throughput that grows linearly with the number of threads. In contrast, the s-tree can only be slightly unbalanced during short periods of time typically too short to affect the performance even under biased workloads. Given that s-tree does not physically remove nodes with two children, in benchmarks with high update rates we observe up to $\frac{1}{3}$ of the nodes as logically deleted but not removed, resulting in slightly longer traversal times (less than one node on average) as a trade-off for lower contention and complexity.

The s-tree improves the red-black tree and the AVL tree performance by up to $1.5\times$ and $1.6\times$, respectively. The s-tree is less prone to contention than AVL and red-black trees, which both share similar performance penalties due to contention.



(a) Elastic transaction speedup vs. s-tree speedup (b) Performance when reusing the s-tree (c) Comparing the s-tree using HTM and STM against the red-black tree using HTM at 10% updates (**left**) and 50% updates (**right**) on Intel

FIGURE 5 Elastic transaction comparison, HTM comparison and reusability

6.3 | Portability to other TM algorithms and HTM

The s-tree is an inherently efficient data structure that is portable to any TM system. It fulfills the TM interface standardized by the industrials and academics[30] and thus does not require the presence of explicit escape mechanisms like early release[11] or snap[32] to avoid extra TM bookkeeping (our uread optimization being optional). Nor does it require high-level conflict detection, like open nesting[33] transactional boosting[14]. Such improvements rely on explicit calls or user-defined abstract locks that require genuine refactoring to work with HTM[34]. To make sure that the obtained results are not biased by the underlying TM algorithm, we evaluated the trees on top of $\mathscr{E}$-STM[19], another TM library (on a 2^{16} sized tree where $\mathscr{E}$-STM proved efficient), on top of a different TM design from the one used so far: with eager acquirement, as well as on Intel's restricted HTM. It is interesting to note that elastic transaction is an improvement of the transaction model whereas the speculation-friendly tree is an improvement of the tree data structure for speculative execution. In particular, an elastic transaction relaxes some of the constraints imposed by the classic transaction model by tolerating benign conflicts between transactions that cannot violate the atomicity of the overlying abstraction (e.g., the dictionary abstraction) implemented by the data structure. It is clear that elastic transactions improve the performance over the classic transactions as was shown before[19]. What is more interesting is that the speedup obtained by replacing the tree by a speculation-friendly tree is higher than the speedup obtained by replacing the classic transactions by elastic transactions. This seems to indicate that reengineering data structures for speculative executions can lead to better performance improvement than reengineering the transaction model. To avoid false conflicts due to the cache level conflict detection of Intel's HTM we experimented with different padding sizes between and within the node structures, but found that this reduced performance except for small trees having only a few hundreds of nodes and with a high update rate. As a result we did not use padding in our experiments. This reduced performance is likely due to the larger amount of memory needed to be synchronized. Due to opt-s-tree needing a lightweight read operation we only implemented s-tree in HTM.

The obtained results, depicted in Figure 4 look similar to the ones obtained with TinySTM-CTL (Figure 3) in that the s-tree executes faster than other trees for all TM settings. This shows that the improvement of s-tree is potentially independent from the TM system used. In addition, to compare the improvement obtained using elastic transactions on red-black trees against the improvement of replacing the red-black tree by the s-tree is depicted in Figure 5(a). It shows that the elastic improvements (15% on average) is lower than the s-tree one (22% on average, be it optimized or not). Furthermore we see that both the optimized and non-optimized versions of the s-tree provide similar performance increases.

Figure 5(c) shows the results comparing the s-tree using Intel's HTM, the s-tree using STM, and the red-black tree (RBtree) using HTM. To ensure progress after 15 consecutive aborts of a transaction a global fallback lock is taken. At 10% updates we see that all trees scale well, with the HTM implementations more than twice as fast as the STM implementation. When increasing the update rate to 50% we see that the s-tree continues to perform well, while the RBtree no longer scales and actually performs worse than the STM version of the s-tree at 3 and 4 threads.

6.4 | Reusability for specific application needs

We illustrate the reusability of the s-tree by composing remove and insert from the existing interface to obtain a new atomic and deadlock-free move operation. Reusability is appealing to simplify concurrent programming by making it modular: a programmer can reuse a concurrent library without having to understand its synchronization internals. While reusability of sequential programs is straightforward, concurrent programs can generally be reused only if the programmer understands how each element is protected. For example, reusing a library can lead to deadlocks if shared data are locked in a different order than what is recommended by the library. Additionally, a lock-striping library may not conflict with a concurrent program that uses per-location locks even though they protect common locations, thus leading to inconsistencies.

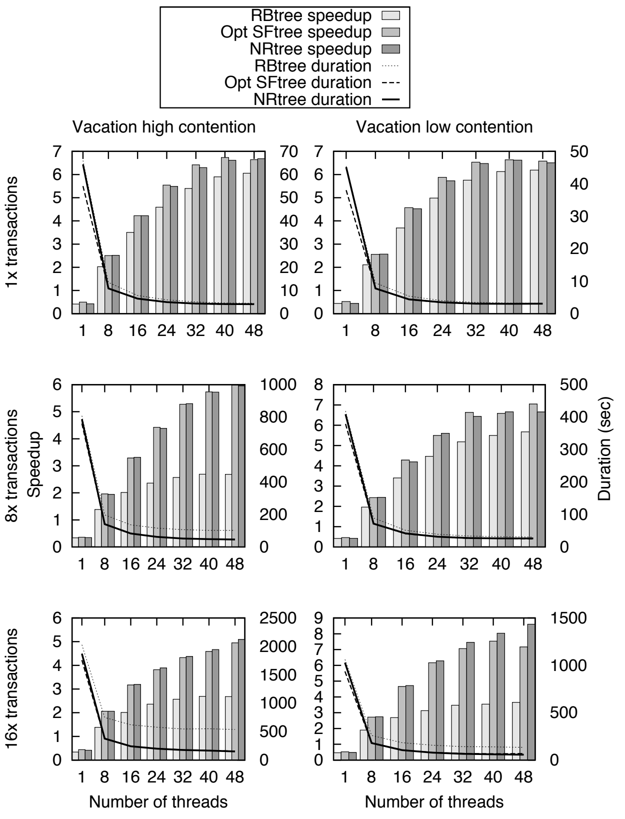


FIGURE 6 The speedup (over single-threaded sequential) and the corresponding duration of the vacation application built upon the red-black tree (RBtree), the opt-s-tree (Opt SFtree) and the no-restructuring tree (NRtree) on **(left)** high contention and **(right)** low contention workloads, and with **(top)** the default number of transaction, **(middle)** $8\times$ more transactions and **(bottom)** $16\times$ more transactions

Figure 5(b) indicates the performance on workloads comprising 90% of read-only operations (including contains and failed updates) and 10% move/insert/delete effective update operations (among which from 1% to 10% are move operations). The $\text{move}(k, k')$ is simply a transaction that checks that k is present while k' is absent to remove the pair $\langle k, * \rangle$ before inserting the pair $\langle k', v \rangle$ with some new value v, all done atomically. The performance decreases as more move operations execute, because a move protects more elements in the data structure than a simple insert or delete operation and during a longer period of time.

6.5 | The vacation travel reservation application

We experiment our optimized library tree with a travel reservation application from the STAMP suite [13], called vacation. This application is suitable for evaluating concurrent binary search trees as it represents a database with four tables implemented as tree-based key-value stores (cars, rooms, flights, and customers) accessed concurrently by client transactions.

Figure 6 depicts the execution time of the STAMP vacation application building on the Oracle red-black tree library (by default), our opt-s-tree, and the baseline no-restructuring tree. Given that the STAMP application nests several operations on the tree within a single transaction resulting in higher contention, we ran the these experiments with opt-s-tree as it provided the highest performance. We added the speedup obtained with each of these tree libraries over the performance of bare sequential code of vacation without synchronization. (A concurrent tree library outperforms the sequential tree when its speedup exceeds 1.) The chosen workloads are the two default configurations ("low contention" and "high contention") taken from the STAMP release, with the default number of transactions, $8\times$ more transactions than by default and $16\times$ more, to increase the duration and the contention of the benchmark without using more threads than cores.

Vacation executes always faster on top of our s-tree than on top of its built-in Oracle red-black tree. For example, the s-tree improves performance by up to $1.3\times$ with the default number of transactions and to $3.5\times$ with $16\times$ more transactions. The reason of this is twofold: (1) In contrast with the s-tree, if an operation on the red-black tree traverses a location that is being deleted then this operation and the deletion conflict. (2) Even though the Oracle red-black tree tolerates that the longest path from the root to a leaf can be twice as long as the shortest one, it triggers the rotation immediately after this threshold is reached. By contrast, our s-tree keeps checking the imbalance to potentially rotate in the background. In particular, we observed on 8 threads in the high contention settings that the red-black tree vacation triggered around $130,000$ rotations whereas the speculation-friendly vacation triggered only $50,000$ rotations.

Finally, we observe that vacation presents similarly good performance on top of the no-restructuring tree library. In rare cases, the s-tree outperforms the no-restructuring tree probably because the no-restructuring tree does not physically remove nodes from the tree, thus leading to a larger tree than the abstraction. Overall, their performance is comparable. With $16\times$ the default number of transactions, the contention is higher and rotations are more costly.

7 | RELATED WORK

Aside from the optimistic synchronization context, various relaxed balanced trees have been proposed. The idea of decoupling the update and the rebalancing was originally proposed by Guibas and Sedgewick[35], proved correct by Kung and Lehman[36] and was applied to AVL trees by Kessels[37] and Nurmi et al.[38], and to red-black trees by Nurmi and Soisalon-Soininen[39], hence leading to chromatic trees. Manber and Ladner propose a lock-based tree whose rebalancing is the task of separate rotator threads running with a low priority[40]. Bougé et al.[25] propose to lock a constant number of nodes within local rotations. Recently, we presented the contention-friendly tree with lock-free contains and lock-based updates that builds upon this decoupling[41]. None of these solutions apply to speculative operations that may abort.

Ellen et al.[42] designed a lock-free binary search tree algorithm. The asymptotic complexity of this tree algorithm is however linear as it does not rotate, as opposed to balanced trees whose complexity is logarithmic. Our lock-free skip list also separates eager access from lazy adaptation to avoid contention hotspots[43].

Ballard[44] proposes a relaxed red-black tree insertion well-suited for transactions. When an insertion unbalances the red-black tree it marks the inserted node rather than rebalancing the tree immediately. Another transaction encountering the marked node must rebalance the tree before restarting. By contrast, our local rotation does not require the rotating transaction to restart, hence benefiting both insertions and removals. Bronson et al.[45] introduce an efficient object-oriented binary search tree. The algorithm uses underlying time-based TM principles to achieve good performance, however, its operations cannot be encapsulated within transactions.

Afek et al.[46] implemented a concurrent red-black tree, requiring the programmer to restrict the TM bookkeeping to some shared accesses hence sharing the limitations of[45]. Felber et al.[19] specify the elastic transactional model that ignores false conflicts but guarantees reusability. The evaluation of this model is performed on various data structures. Besides these relaxed model, it is well-known that different transactional memory implementations may affect the performance under specific workload[47], and in particular unit testing have shown significant performance discrepancies between implementation choices[48]. These approaches are orthogonal to ours as they aim at improving the performance of the underlying TM, independently from the data structure. Avni et al.[49] propose using HTM with Consistency Oblivious Programming (COP) where operations are split into a traversal phase without synchronization, followed by a validation phase performed within a transaction to ensure the correct node was reached, and finally a modification phase where rebalance operations are performed. Modifications are performed in a single HTM transaction. Given we did not implement opt-s-tree in HTM due to it requiring lightweight reads, one should be able to combine the approaches of COP and s-tree for improved performance. Siakavaras et al.[50] apply Read-Copy-Update (RCU) techniques to context of AVL trees in HTM by having update operations create a copy of the section of the tree they modify, then linking in the modified tree while keeping the original nodes in the read set to prevent conflicts. This allows read operations to be performed entirely without synchronization. Different to s-tree, modifications due to rotations may still result in large transactions resulting in a trade-off between fast reads and a possible increase in conflicts. Note that both[49] and[50] rely on the strong memory model semantics of Intel's HTM between transactional and non-transactional accesses, while the non-optimized version of s-tree does not.

Finally, it is well-known that the multicore era raises new questions on the design of scalable data structures[51]. More generally, several strategies have been explored to diminish the data structure contention on multicore platforms. First, one may violate atomicity and provide a weaker consistency criterion, hence Moir and Shavit[52] refer to quiescent consistency whereas[53] provide some variant of causal consistency. Second, one may limit concurrency by serializing all modifications. McKenney and Slingwine[54] proposed the read-copy-update to let read-only operations proceed without interfering, a technique reused recently by[55] to build a tree data structure that prohibits concurrent updates. Another serializing technique is the flat combining approach by[56] which pays off for abstractions whose sequential implementation generally outperforms their concurrent counterpart, like queues. Contention-friendliness guarantees atomicity (and thus the sequential specification) to make it easy to reason about the abstraction and exploits concurrency among all operations to outperform their sequential implementations.

8 | CONCLUDING REMARKS

In contrast with the traditional pessimistic synchronization, the optimistic synchronization allows the programmer to directly observe the impact of contention as part of the step complexity because conflicts potentially lead to subsequent speculative re-executions. We illustrated, with the first speculation-friendly data structure, how one can exploit this information to design an efficient multi-core library with both hardware and software TMs. Our s-tree decreases the inherent contention by relaxing the invariants of the structure while preserving the invariants of the abstraction. More generally, the transient relaxation of structural invariants help the data structure cope with the contention between multiple cores without affecting correctness. For future work we would like to explore the possibility of generalizing this idea to various data structures in transactional memory. For example[43] improves the performance of lock-free skip-lists by relaxing the height requirements of list items. For hash tables one could imagine relaxation of bucket size during the resize operation of the table.

Availability. The implementation of the speculation-friendly tree is publicly available online at https://github.com/gramoli/synchrobench.

Acknowledgments. This research was supported under Australian Research Council's Discovery Projects funding scheme (project number 160104801) entitled "Data Structures for Multi-Core". Vincent Gramoli is the recipient of the Australian Research Council Discovery International Award.

References

1. Crain Tyler, Gramoli Vincent, Raynal Michel. A Speculation-friendly Binary Search Tree. *SIGPLAN Not.*. 2012;47(8):161–170.
2. Crain Tyler, Gramoli Vincent, Raynal Michel. A Speculation-friendly Binary Search Tree. In: PPoPP '12:161–170ACM; 2012; New York, NY, USA.
3. Herlihy Maurice, Moss J. Eliot B.. Transactional Memory: Architectural Support For Lock-Free Data Structures. In: :289–300; 1993.
4. Shavit Nir, Touitou Dan. Software Transactional Memory. In: :204–213; 1995.
5. Harris Tim, Larus James, Rajwar Ravi. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers; 2nd ed.2010.
6. Rajwar Ravi, Goodman James R.. Speculative lock elision: enabling highly concurrent multithreaded execution. In: :294–305; 2001.
7. Wang Amy, Gaudet Matthew, Wu Peng, et al. Evaluation of Blue Gene/Q hardware support for transactional memories. In: :127-136; 2012.
8. Pankratius Victor, Adl-Tabatabai Ali-Reza. A study of transactional memory vs. locks in practice. In: :43–52; 2011.
9. Rossbach Christopher J., Hofmann Owen S., Witchel Emmett. Is transactional programming actually easier?. In: :47–56; 2010.
10. Harris Tim, Marlow Simon, Peyton-Jones Simon, Herlihy Maurice. Composable memory transactions. In: :48–60; 2005.
11. Herlihy Maurice, Luchangco Victor, Moir Mark, Scherer William N.. Software transactional memory for dynamic-sized data structures. In: :92–101; 2003.
12. Dice Dave, Shalev Ori, Shavit Nir. Transactional Locking II. In: :194–208; 2006.
13. Cao Minh Chi, Chung JaeWoong, Kozyrakis Christos, Olukotun Kunle. STAMP: Stanford Transactional Applications for Multi-Processing. In: :35–46; 2008.
14. Herlihy Maurice, Koskinen Eric. Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects. In: :207–216; 2008.
15. Felber Pascal, Fetzer Christof, Riegel Torvald. Dynamic performance tuning of word-based software transactional memory. In: :237–246; 2008.
16. Yoo Richard M., Ni Yang, Welc Adam, Saha Bratin, Adl-Tabatabai Ali-Reza, Lee Hsien-Hsin S.. Kicking the tires of software transactional memory: why the going gets tough. In: :265–274; 2008.
17. Dragojevic Aleksandar, Felber Pascal, Gramoli Vincent, Guerraoui Rachid. Why STM can be more than a Research Toy. *Commun. ACM.* 2011;54(4):70–77.
18. Siakavaras Dimitrios, Nikas Konstantinos, Goumas Georgios, Koziris Nectarios. Performance analysis of concurrent red-black trees on HTM platforms. In: ; 2015.

19. Felber Pascal, Gramoli Vincent, Guerraoui Rachid. Elastic Transactions. *Journal of Parallel and Distributing Computing.* 2016;.

20. Adelson-Velskii Georgy, Landis Evgenii M.. An algorithm for the organization of information. In: :263–266; 1962.

21. Bayer Rudolf. Symmetric binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica 1.* 1972;1(4):290–306.

22. Gramoli Vincent. More Than You Ever Wanted to Know about Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In: :1–10ACM; 2015.

23. Dijkstra Edsger W., Lamport Leslie, Martin A. J., Scholten C. S., Steffens E. F. M.. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM.* 1978;21(11):966–975.

24. Mohan C.. Commit-LSN: a novel and simple method for reducing locking and latching in transaction processing systems. In: :406–418; 1990.

25. Bougé Luc, Gabarro Joaquim, Messeguer Xavier, Schabanel Nicolas. *Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries.* Research Report 1998-18, ENS Lyon; 1998.

26. Harris Tim. A Pragmatic Implementation of Non-blocking Linked-Lists. In: :300–314; 2001.

27. Herlihy Maurice, Shavit Nir. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc.; 2008.

28. Gramoli Vincent, Guerraoui Rachid. Democratizing Transactional Programming. In: :1–19; 2011.

29. Guerraoui Rachid, Kapalka Michal. On the correctness of transactional memory. In: :175–184; 2008.

30. Intel Corporation . *Intel Transactional Memory Compiler and Runtime Application Binary Interface.* 2009.

31. Dalessandro Luke, Spear Michael, Scott Michael L.. NOrec: streamlining STM by abolishing ownership records. In: :67–78; 2010.

32. Cole Christopher, Herlihy Maurice. Snapshots and software transactional memory. *Sci. Comput. Program..* 2005;58(3):310–324.

33. Moss J. Eliot B.. Open Nested Transactions: Semantics and Support. In: ; 2005. Poster presentation.

34. Chapman Keith, Hosking Antony L., Moss J. Eliot B.. Hybrid STM/HTM for nested transactions on OpenJDK. In: :660–676; 2016.

35. Guibas Leo J., Sedgewick Robert. A Dichromatic Framework for Balanced Trees. In: :8–21; 1978.

36. Kung H. T., Lehman Philip L.. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst..* 1980;5(3):354–382.

37. Kessels Joep L. W.. On-the-Fly Optimization of Data Structures. *Comm. ACM.* 1983;26(11):895–901.

38. Nurmi Otto, Soisalon-Soininen Eljas, Wood D.. Concurrency Control in Database Structures with Relaxed Balance. In: :170–176; 1987.

39. Nurmi Otto, Soisalon-Soininen Eljas. Uncoupling Updating and Rebalancing in Chromatic Binary Search Trees. In: :192–198; 1991.

40. Manbar Udi, Ladner Richard E.. Concurrency control in a dynamic search structure. *ACM Trans. Database Syst..* 1984;9(3):439–455.

41. Crain Tyler, Gramoli Vincent, Raynal Michel. A Fast Contention-Friendly Binary Search Tree. *Parallel Processing Letters (PPL).* 2016;26(03).

42. Ellen Faith, Fatourou Panagiota, Ruppert Eric, Breugel Franck. Non-blocking binary search trees. In: :131-140; 2010.

43. Crain Tyler, Gramoli Vincent, Raynal Michel. No Hot Spot Non-Blocking Skip List. In: IEEE , ed. *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS'13)*, :196–205; 2013.

44. Ballard Lucia. *Conflict Avoidance: Data Structures in Transactional Memory.* Undergraduate thesis, Brown University; 2006.

45. Bronson Nathan G., Casper Jared, Chafi Hassan, Olukotun Kunle. A practical concurrent binary search tree. In: :257–268; 2010.

46. Afek Yehuda, Avni Hillel, Shavit Nir. Towards Consistency Oblivious Programming. In: :65-79; 2011.

47. Harmanci Derin, Gramoli Vincent, Felber Pascal, Fetzer Christof. Extensible Transactional Memory Testbed. *Journal of Parallel and Distributed Computing - Special Issue on Transactional Memory (JPDC).* 2010;70(10):1053–1067.

48. Harmanci Derin, Felber Pascal, Gramoli Vincent, Fetzer Christof. TMunit: Testing Software Transactional Memories. In: ; 2009.

49. Improving htm scaling with consistency-oblivious programming. In: ; 2014.

50. Siakavaras D., Nikas K., Goumas G., Koziris N.. RCU-HTM: Combining RCU with HTM to Implement Highly Efficient Concurrent Binary Search Trees. In: :1-13; 2017.

51. Shavit Nir. Data structures in the multicore age. *Commun. ACM.* 2011;54(3):76–84.

52. Moir Mark, Shavit Nir. Concurrent Data Structures. In: 2007 (pp. 14–30).

53. Howard Philip W., Walpole Jonathan. A relativistic enhancement to software transactional memory. In: :15–15; 2011.

54. McKenney Paul. E., Slingwine J. D.. Read Copy Update: Using Execution History to Solve Concurrency Problems. In: :509–518; 1998.

55. Clements Austin T., Kaashoek M. Frans, Zeldovich Nickolai. Scalable address spaces using RCU balanced trees. In: :199–210; 2012.

56. Hendler Danny, Incze Itai, Shavit Nir, Tzafrir Moran. Flat combining and the synchronization-parallelism tradeoff. In: :355–364; 2010.

Update	0%	10%	20%	30%	40%	50%
AVL tree	29	415	711	1008	1981	2081
Oracle red-black tree	31	573	965	1108	1484	1545
s-tree	29	75	123	120	144	180

TABLE 1 Maximum number of transactional reads per operation on three 2^{12}-sized balanced search trees as the update ratio increases

How to cite this article: T. Crain, V. Gramoli, and M. Raynal (2018), A Speculation-Friendly Binary Search Tree, , .