A General Technique for Non-blocking Trees

Trevor Brown and Faith Ellen
University of Toronto, Canada

Eric Ruppert
York University, Canada

Abstract

We describe a general technique for obtaining provably correct, non-blocking implementations of a large class of tree data structures where pointers are directed from parents to children. Updates are permitted to modify any contiguous portion of the tree atomically. Our non-blocking algorithms make use of the LLX, SCX and VLX primitives, which are multi-word generalizations of the standard LL, SC and VL primitives and have been implemented from single-word CAS [10].

To illustrate our technique, we describe how it can be used in a fairly straightforward way to obtain a non-blocking implementation of a chromatic tree, which is a relaxed variant of a red-black tree. The height of the tree at any time is $O(c + \log n)$, where n is the number of keys and c is the number of updates in progress. We provide an experimental performance analysis which demonstrates that our Java implementation of a chromatic tree rivals, and often significantly outperforms, other leading concurrent dictionaries.

Categories and Subject Descriptors E.1 [*Data*]: Data Structures—Distributed data structures

Keywords balanced binary search tree; non-blocking; chromatic tree; relaxed balance; red-black tree

1. Introduction

The binary search tree (BST) is among the most important data structures. Previous concurrent implementations of balanced BSTs without locks either used coarse-grained transactions, which limit concurrency, or lacked rigorous proofs of correctness. In this paper, we describe a general technique for implementing *any* data structure based on a down-tree (a directed acyclic graph of indegree one), with updates that modify any connected subgraph of the tree atomically. The resulting implementations are non-blocking, which means that some process is always guaranteed to make progress, even if processes crash. Our approach drastically simplifies the task of proving correctness. This makes it feasible to develop provably correct implementations of non-blocking balanced BSTs with fine-grained synchronization (i.e., with updates that synchronize on a small constant number of nodes).

As with all concurrent implementations, the implementations obtained using our technique are more efficient if each update to the data structure involves a small number of nodes near one another. We call such an update *localized*. We use *operation* to denote an operation of the abstract data type (ADT) being implemented by the data structure. Operations that cannot modify the data structure are called *queries*. For some data structures, such as Patricia tries and leaf-oriented BSTs, operations modify the data structure using a single localized update. In some other data structures, operations that modify the data structure can be split into several localized updates that can be freely interleaved.

A particularly interesting application of our technique is to implement *relaxed-balance* versions of sequential data structures efficiently. Relaxed-balance data structures decouple updates that rebalance the data structure from operations, and allow updates that accomplish rebalancing to be delayed and freely interleaved with other updates. For example, a chromatic tree is a relaxed-balance version of a red-black tree (RBT) which splits up the insertion or deletion of a key and any subsequent rotations into a sequence of localized updates. There is a rich literature of relaxed-balance versions of sequential data structures [22], and several papers (e.g., [24]) have described general techniques that can be used to easily produce them from large classes of existing sequential data structures. The small number of nodes involved in each update makes relaxed-balance data structures perfect candidates for efficient implementation using our technique.

Our Contributions

- We provide a simple template that can be filled in to obtain an implementation of any update for a data structure based on a down-tree. We prove that any data structure that follows our template for all of its updates will automatically be linearizable and non-blocking. The template

PPoPP '14, February 15–19, 2014, Orlando, Florida, USA.

http://dx.doi.org/10.1145/2555243.2555267

takes care of all process coordination, so the data structure designer is able to think of updates as atomic steps.

- To demonstrate the use of our template, we provide a complete, provably correct, non-blocking linearizable implementation of a chromatic tree [27], which is a relaxed-balanced version of a RBT. To our knowledge, this is the first provably correct, non-blocking balanced BST implemented using fine-grained synchronization. Our chromatic trees always have height $O(c + \log n)$, where n is the number of keys stored in the tree and c is the number of insertions and deletions that are in progress (Section 5.3).
- We show that sequential implementations of some queries are linearizable, even though they completely ignore concurrent updates. For example, an ordinary BST search (that works when there is no concurrency) also works in our chromatic tree. Ignoring updates makes searches very fast. We also describe how to perform successor queries in our chromatic tree, which interact properly with updates that follow our template (Section 5.5).
- We show experimentally that our Java implementation of a chromatic tree rivals, and often significantly outperforms, known highly-tuned concurrent dictionaries, over a variety of workloads, contention levels and thread counts. For example, with 128 threads, our algorithm outperforms Java's non-blocking skip-list by 13% to 156%, the lock-based AVL tree of Bronson et al. by 63% to 224%, and a RBT that uses software transactional memory (STM) by 13 to 134 times (Section 6).

2. Related Work

There are many lock-based implementations of search tree data structures. (See [1, 9] for state-of-the-art examples.) Here, we focus on implementations that do not use locks. Valois [32] sketched an implementation of non-blocking node-oriented BSTs from CAS. Fraser [17] gave a non-blocking BST using 8-word CAS, but did not provide a full proof of correctness. He also described how multi-word CAS can be implemented from single-word CAS instructions. Ellen et al. [15] gave a provably correct, non-blocking implementation of leaf-oriented BSTs directly from single-word CAS. A similar approach was used for k-ary search trees [11] and Patricia tries [28]. All three used the cooperative technique originated by Turek, Shasha and Prakash [31] and Barnes [4]. Howley and Jones [20] used a similar approach to build node-oriented BSTs. They tested their implementation using a model checker, but did not prove it correct. Natarajan and Mittal [25] give another leaf-oriented BST implementation, together with a sketch of correctness. Instead of marking nodes, it marks edges. This enables insertions to be accomplished by a single CAS, so they do not need to be helped. It also combines deletions that would otherwise conflict. All of these trees are not balanced, so the height of a tree with n keys can be $\Theta(n)$.

Tsay and Li [30] gave a general approach for implementing trees in a wait-free manner using LL and SC operations (which can, in turn be implemented from CAS, e.g., [3]). However, their technique requires every process accessing the tree (even for read-only operations such as searches) to copy an entire path of the tree starting from the root. Concurrency is severely limited, since every operation must change the root pointer. Moreover, an extra level of indirection is required for every child pointer.

Red-black trees [5, 18] are well known BSTs that have height $\Theta(\log n)$. Some attempts have been made to implement RBTs without using locks. It was observed that the approach of Tsay and Li could be used to implement wait-free RBTs [26] and, furthermore, this could be done so that only updates must copy a path; searches may simply read the path. However, the concurrency of updates is still very limited. Herlihy et al. [19] and Fraser and Harris [16] experimented on RBTs implemented using software transactional memory (STM), which only satisfied obstruction-freedom, a weaker progress property. Each insertion or deletion, together with necessary rebalancing is enclosed in a single large transaction, which can touch all nodes on a path from the root to a leaf.

Some researchers have attempted fine-grained approaches to build non-blocking balanced search trees, but they all use extremely complicated process coordination schemes. Spiegel and Reynolds [29] described a non-blocking data structure that combines elements of B-trees and skip lists. Prior to this paper, it was the leading implementation of an ordered dictionary. However, the authors provided only a brief justification of correctness. Braginsky and Petrank [8] described a B+tree implementation. Although they have posted a correctness proof, it is very long and complex.

In a balanced search tree, a process is typically responsible for restoring balance after an insertion or deletion by performing a series of rebalancing steps along the path from the root to the location where the insertion or deletion occurred. Chromatic trees, introduced by Nurmi and Soisalon-Soininen [27], decouple the updates that perform the insertion or deletion from the updates that perform the rebalancing steps. Rather than treating an insertion or deletion and its associated rebalancing steps as a single, large update, it is broken into smaller, localized updates that can be interleaved, allowing more concurrency. This decoupling originated in the work of Guibas and Sedgewick [18] and Kung and Lehman [21]. We use the leaf-oriented chromatic trees by Boyar, Fagerberg and Larsen [7]. They provide a family of local rebalancing steps which can be executed in any order, interspersed with insertions and deletions. Moreover, an amortized *constant* number of rebalancing steps per INSERT or DELETE is sufficient to restore balance for any sequence of operations. We have also used our template to

implement a non-blocking version of Larsen's leaf-oriented relaxed AVL tree [23]. In such a tree, an amortized *logarithmic* number of rebalancing steps per INSERT or DELETE is sufficient to restore balance.

There is also a node-oriented relaxed AVL tree by Bougé et al. [6], in which an amortized *linear* number of rebalancing steps per INSERT or DELETE is sufficient to restore balance. Bronson et al. [9] developed a highly optimized fine-grained locking implementation of this data structure using optimistic concurrency techniques to improve search performance. Deletion of a key stored in an internal node with two children is done by simply marking the node and a later insertion of the same key can reuse the node by removing the mark. If all internal nodes are marked, the tree is essentially leaf-oriented. Crain et al. gave a different implementation using lock-based STM [12] and locks [13], in which *all* deletions are done by marking the node containing the key. Physical removal of nodes and rotations are performed by one separate thread. Consequently, the tree can become very unbalanced. Drachsler et al. [14] give another fine-grained lock-based implementation, in which deletion physically removes the node containing the key and searches are non-blocking. Each node also contains predecessor and successor pointers, so when a search ends at an incorrect leaf, sequential search can be performed to find the correct leaf. A non-blocking implementation of Bougé's tree has not appeared, but our template would make it easy to produce one.

3. LLX, SCX and VLX Primitives

The load-link extended (LLX), store-conditional extended (SCX) and validate-extended (VLX) primitives are multi-word generalizations of the well-known load-link (LL), store-conditional (SC) and validate (VL) primitives and they have been implemented from single-word CAS [10]. The benefit of using LLX, SCX and VLX to implement our template is two-fold: the template can be described quite simply, and much of the complexity of its correctness proof is encapsulated in that of LLX, SCX and VLX.

Instead of operating on single words, LLX, SCX and VLX operate on Data-records, each of which consists of a fixed number of mutable fields (which can change), and a fixed number of immutable fields (which cannot). LLX(r) attempts to take a snapshot of the mutable fields of a Data-record r. If it is concurrent with an SCX involving r, it may return FAIL, instead. Individual fields of a Data-record can also be read directly. An SCX(V, R, fld, new) takes as arguments a sequence V of Data-records, a subsequence R of V, a pointer fld to a mutable field of one Data-record in V, and a new value new for that field. The SCX tries to atomically store the value new in the field that fld points to and *finalize* each Data-record in R. Once a Data-record is finalized, its mutable fields cannot be changed by any subsequent SCX, and any LLX of the Data-record will return FINALIZED instead of a snapshot.

Before a process invokes SCX or VLX(V), it must perform an LLX(r) on each Data-record r in V. The last such LLX by the process is said to be *linked* to the SCX or VLX, and the linked LLX must return a snapshot of r (not FAIL or FINALIZED). An SCX(V, R, fld, new) by a process modifies the data structure only if each Data-record r in V has not been changed since its linked LLX(r); otherwise the SCX fails. Similarly, a VLX(V) returns TRUE only if each Data-record r in V has not been changed since its linked LLX(r) by the same process; otherwise the VLX fails. VLX can be used to obtain a snapshot of a set of Data-records. Although LLX, SCX and VLX can fail, their failures are limited in such a way that we can use them to build non-blocking data structures. See [10] for a more formal specification of these primitives.

These new primitives were designed to balance ease of use and efficient implementability using single-word CAS. The implementation of the primitives from CAS in [10] is more efficient if the user of the primitives can guarantee that two constraints, which we describe next, are satisfied. The first constraint prevents the ABA problem for the CAS steps that actually perform the updates.

Constraint 1: Each invocation of SCX(V, R, fld, new) tries to change fld to a value new that it never previously contained.

The implementation of SCX does something akin to locking the elements of V in the order they are given. Livelock can be easily avoided by requiring all V sequences to be sorted according to some total order on Data-records. However, this ordering is necessary only to guarantee that SCXs continue to succeed. Therefore, as long as SCXs are still succeeding in an execution, it does not matter how V sequences are ordered. This observation leads to the following constraint, which is much weaker.

Constraint 2: Consider each execution that contains a configuration C after which the value of no field of any Data-record changes. There is a total order of all Data-records created during this execution such that, for every SCX whose linked LLXs begin after C, the V sequence passed to the SCX is sorted according to the total order.

It is easy to satisfy these two constraints using standard approaches, e.g., by attaching a version number to each field, and sorting V sequences by any total order, respectively. However, we shall see that Constraints 1 and 2 are *automatically* satisfied in a natural way when LLX and SCX are used according to our tree update template.

Under these constraints, the implementation of LLX, SCX, and VLX in [10] guarantees that there is a linearization of all SCXs that modify the data structure (which may include SCXs that do not terminate because a process crashed, but *not* any SCXs that fail), and all LLXs and VLXs that return, but do not fail.

We assume there is a Data-record $entry$ which acts as the entry point to the data structure and is never deleted. This

Data-record points to the root of a down-tree. We represent an empty down-tree by a pointer to an empty Data-record. A Data-record is *in the tree* if it can be reached by following pointers from $entry$. A Data-record r is *removed from the tree* by an SCX if r is in the tree immediately prior to the linearization point of the SCX and is not in the tree immediately afterwards. Data structures produced using our template *automatically* satisfy one additional constraint:

Constraint 3: A Data-record is finalized when (and only when) it is removed from the tree.

Under this additional constraint, the implementation of LLX and SCX in [10] also guarantees the following three properties.

- If $\text{LLX}(r)$ returns a snapshot, then r is in the tree just before the LLX is linearized.
- If an $\text{SCX}(V, R, fld, new)$ is linearized and new is (a pointer to) a Data-record, then this Data-record is in the tree immediately after the SCX is linearized.
- If an operation reaches a Data-record r by following pointers read from other Data-records, starting from $entry$, then r was in the tree at some earlier time during the operation.

These properties are useful for proving the correctness of our template. In the following, we sometimes abuse notation by treating the sequences V and R as sets, in which case we mean the set of all Data-records in the sequence.

The memory overhead introduced by the implementation of LLX and SCX is fairly low. Each node in the tree is augmented with a pointer to a descriptor and a bit. Every node that has had one of its child pointers changed by an SCX points to a descriptor. (Other nodes have a NIL pointer.) A descriptor can be implemented to use only three machine words after the update it describes has finished. The implementation of LLX and SCX in [10] assumes garbage collection, and we do the same in this work. This assumption can be eliminated by using, for example, the new efficient memory reclamation scheme of Aghazadeh et al. [2].

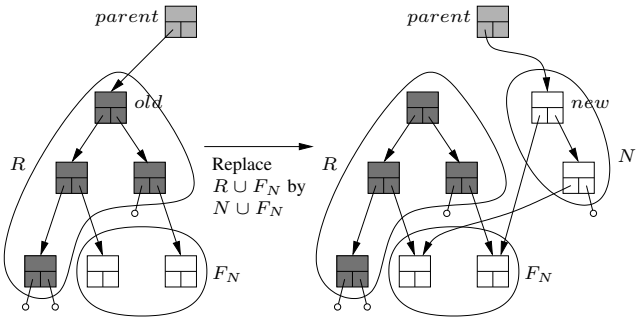

Figure 1. Example of the tree update template. R is the set of nodes to be removed, N is a tree of new nodes that have never before appeared in the tree, and F_N is the set of children of N (and of R). Nodes in F_N may have children. The shaded nodes (and possibly others) are in the sequence V of the SCX that performs the update. The darkly shaded nodes are finalized by the SCX.

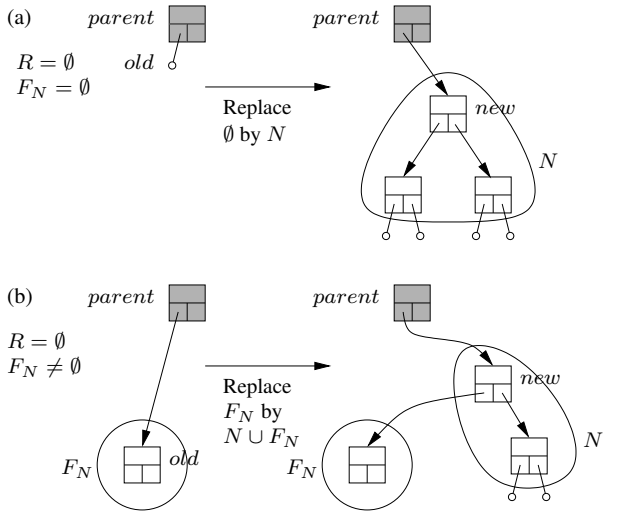

Figure 2. Examples of two special cases of the tree update template when no nodes are removed from the tree. (a) Replacing a NIL child pointer: In this case, $R = F_N = \emptyset$. (b) Inserting new nodes in the middle of the tree: In this case, $R = \emptyset$ and F_N consists of a single node.

4. Tree Update Template

Our tree update template implements updates that atomically replace an old connected subgraph in a down-tree by a new connected subgraph. Such an update can implement any change to the tree, such as an insertion into a BST or a rotation used to rebalance a RBT. The old subgraph includes all nodes with a field (including a child pointer) to be modified. The new subgraph may have pointers to nodes in the old tree. Since every node in a down-tree has indegree one, the update can be performed by changing a single child pointer of some node $parent$. (See Figure 1.) However, problems could arise if a concurrent operation changes the part of the tree being updated. For example, nodes in the old subgraph, or even $parent$, could be removed from the tree before $parent$'s child pointer is changed. Our template takes care of the process coordination required to prevent such problems.

Each tree node is represented by a Data-record with a fixed number of child pointers as its mutable fields (but different nodes may have different numbers of child fields). Each child pointer points to a Data-record or contains NIL (denoted by $\multimap$ in our figures). For simplicity, we assume that any other data in the node is stored in immutable fields. Thus, if an update must change some of this data, it makes a new copy of the node with the updated data.

Our template for performing an update to the tree is fairly simple: An update first performs LLXs on nodes in a contiguous portion of the tree, including $parent$ and the set

```
1  TEMPLATE(args)
2    follow zero or more pointers from entry to reach a node n_0
3    i := 0
4    loop
5      s_i := LLX(n_i)
6      if s_i ∈ {FAIL, FINALIZED} then return FAIL
7      s'_i := immutable fields of n_i
8      exit loop when CONDITION(s_0, s'_0, ..., s_i, s'_i, args)
             ▷ CONDITION must eventually return TRUE
9      n_{i+1} := NEXTNODE(s_0, s'_0, ..., s_i, s'_i, args)
             ▷ returns a non-NIL child pointer from one of s_0, ..., s_i
10     i := i + 1
11   end loop
12   if SCX(SCX-ARGUMENTS(s_0, s'_0, ..., s_i, s'_i, args)) then
         return RESULT(s_0, s'_0, ..., s_i, s'_i, args)
13   else return FAIL
```

Figure 3. Tree update template. CONDITION, NEXTNODE, SCX-ARGUMENTS and RESULT can be filled in with any locally computable functions, provided that SCX-ARGUMENTS satisfies postconditions PC1 to PC8.

R of nodes to be removed from the tree. Then, it performs an SCX that atomically changes the child pointer as shown in Figure 1 and finalizes nodes in R. Figure 2 shows two special cases where R is empty. An update that performs this sequence of steps is said to *follow* the template.

We now describe the tree update template in more detail. An update UP($args$) that follows the template shown in Figure 3 takes any arguments, $args$, that are needed to perform the update. UP first reads a sequence of child pointers starting from $entry$ to reach some node n_0. Then, UP performs LLXs on a sequence $\sigma = \langle n_0, n_1, \ldots \rangle$ of nodes starting with n_0. For maximal flexibility of the template, the sequence σ can be constructed on-the-fly, as LLXs are performed. Thus, UP chooses a non-NIL child of one of the previous nodes to be the next node of σ by performing some deterministic local computation (denoted by NEXTNODE in Figure 3) using any information that is available locally, namely, the snapshots of mutable fields returned by LLXs on the previous elements of σ, values read from immutable fields of previous elements of σ, and $args$. (This flexibility can be used, for example, to avoid unnecessary LLXs when deciding how to rebalance a BST.) UP performs another local computation (denoted by CONDITION in Figure 3) to decide whether more LLXs should be performed. To avoid infinite loops, this function must eventually return TRUE in any execution of UP. If any LLX in the sequence returns FAIL or FINALIZED, UP also returns FAIL, to indicate that the attempted update has been aborted because of a concurrent update on an overlapping portion of the tree. If all of the LLXs successfully return snapshots, UP invokes SCX and returns a result calculated locally by the RESULT function (or FAIL if the SCX fails).

The full version of this paper describes an optimization to the template which does not require invocations of UP to always begin at $entry$. For example, an operation may backtrack from n_0 to try again from a nearby node after an invocation of UP returns FAIL.

UP applies the function SCX-ARGUMENTS to use locally available information to construct the arguments V, R, fld and new for the SCX. The postconditions that must be satisfied by SCX-ARGUMENTS are somewhat technical, but intuitively, they are meant to ensure that the arguments produced describe an update as shown in Figure 1 or Figure 2. The update must remove a connected set R of nodes from the tree and replace it by a connected set N of newly-created nodes that is rooted at new by changing the child pointer stored in fld to point to new. In order for this change to occur atomically, we include R and the node containing fld in V. This ensures that if any of these nodes has changed since it was last accessed by one of UP's LLXs, the SCX will fail. The sequence V may also include any other nodes in σ.

More formally, we require SCX-ARGUMENTS to satisfy nine postconditions. The first three are basic requirements of SCX.

PC1: V is a subsequence of σ.

PC2: R is a subsequence of V.

PC3: The node $parent$ containing the mutable field fld is in V.

Let G_N be the directed graph $(N \cup F_N, E_N)$, where E_N is the set of all child pointers of nodes in N when they are initialized, and $F_N = \{y : y \notin N \text{ and } (x, y) \in E_N \text{ for some } x \in N\}$. Let old be the value read from fld by the LLX on $parent$.

PC4: G_N is a non-empty down-tree rooted at new.

PC5: If $old =$ NIL then $R = \emptyset$ and $F_N = \emptyset$.

PC6: If $R = \emptyset$ and $old \neq$ NIL, then $F_N = \{old\}$.

PC7: UP allocates memory for all nodes in N, including new.

Postcondition PC7 requires new to be a newly-created node, in order to satisfy Constraint 1. There is no loss of generality in using this approach: If we wish to change a child y of node x to NIL (to chop off the entire subtree rooted at y) or to a descendant of y (to splice out a portion of the tree), then, instead, we can replace x by a new copy of x with an updated child pointer. Likewise, if we want to delete the entire tree, then $entry$ can be changed to point to a new, empty Data-record.

The next postcondition is used to guarantee Constraint 2, which is used to prove progress.

PC8: The sequences V constructed by all updates that take place entirely during a period of time when no SCXs change the tree structure must be ordered consistently according to a fixed tree traversal algorithm (for example, an in-order traversal or a breadth-first traversal).

Stating the remaining postcondition formally requires some care, since the tree may be changing while UP performs its LLXs. If $R \neq \emptyset$, let G_R be the directed graph $(R \cup F_R, E_R)$, where E_R is the union of the sets of edges representing child pointers read from each $r \in R$ when it was last accessed by one of UP's LLXs and $F_R = \{y :$

$y \notin R$ and $(x, y) \in E_R$ for some $x \in R\}$. G_R represents UP's view of the nodes in R according to its LLXs, and F_R is the *fringe* of G_R. If other processes do not change the tree while UP is being performed, then F_R contains the nodes that should remain in the tree, but whose parents will be removed and replaced. Therefore, we must ensure that the nodes in F_R are reachable from nodes in N (so they are not accidentally removed from the tree). Let G_σ be the directed graph $(\sigma \cup F_\sigma, E_\sigma)$, where E_σ is the union of the sets of edges representing child pointers read from each $r \in \sigma$ when it was last accessed by one of UP's LLXs and $F_\sigma = \{y : y \notin \sigma$ and $(x, y) \in E_\sigma$ for some $x \in \sigma\}$. Since G_σ, G_R and G_N are not affected by concurrent updates, the following postcondition can be proved using purely sequential reasoning, ignoring the possibility that concurrent updates could modify the tree during UP.

PC9: If G_σ is a down-tree and $R \neq \emptyset$, then G_R is a non-empty down-tree rooted at old and $F_N = F_R$.

4.1 Correctness and Progress

For brevity, we only sketch the main ideas of the proof here. The full version of this paper, containing a complete proof, is available from http://www.cs.utoronto.ca/~tabrown. Consider a data structure in which all updates follow the tree update template and SCX-ARGUMENTS satisfies postconditions PC1 to PC9. We prove, by induction on the sequence of steps in an execution, that the data structure is always a tree, each call to LLX and SCX satisfies its preconditions, Constraints 1 to 3 are satisfied, and each successful SCX atomically replaces a connected subgraph containing nodes $R \cup F_N$ with another connected subgraph containing nodes $N \cup F_N$, finalizing and removing the nodes in R from the tree, and adding the new nodes in N to the tree. We also prove no node in the tree is finalized, every removed node is finalized, and removed nodes are never reinserted.

We linearize each update UP that follows the template and performs an SCX that modifies the data structure at the linearization point of its SCX. We prove the following correctness properties.

C1: If UP were performed atomically at its linearization point, then it would perform LLXs on the same nodes, and these LLXs would return the same values.

This implies that UP's SCX-ARGUMENTS and RESULT computations must be the same as they would be if UP were performed atomically at its linearization point, so we obtain the following.

C2: If UP were performed atomically at its linearization point, then it would perform the same SCX (with the same arguments) and return the same value.

Additionally, a property is proved in [10] that allows some query operations to be performed very efficiently using only READs, for example, GET in Section 5.

C3: If a process p follows child pointers starting from a node in the tree at time t and reaches a node r at time $t' \geq t$, then r *was* in the tree at some time between t and t'. Furthermore, if p reads v from a mutable field of r at time $t'' \geq t'$ then, at some time between t and t'', node r was in the tree and this field contained v.

The following properties, which come from [10], can be used to prove non-blocking progress of queries.

P1: If LLXs are performed infinitely often, then they return snapshots or FINALIZED infinitely often.

P2: If VLXs are performed infinitely often, and SCXs are not performed infinitely often, then VLXs return TRUE infinitely often.

Each update that follows the template is wait-free. Since updates can fail, we also prove the following progress property.

P3: If updates that follow the template are performed infinitely often, then updates succeed infinitely often.

A successful update performs an SCX that modifies the tree. Thus, it is necessary to show that SCXs succeed infinitely often. Before an invocation of SCX(V, R, fld, new) can succeed, it must perform an LLX(r) that returns a snapshot, for each $r \in V$. Even if P1 is satisfied, it is possible for LLXs to always return FINALIZED, preventing any SCXs from being performed. We prove that any algorithm whose updates follow the template automatically guarantees that, for each Data-record r, each process performs at most one invocation of LLX(r) that returns FINALIZED. We use this fact to prove P3.

5. Application: Chromatic Trees

Here, we show how the tree update template can be used to implement an ordered dictionary ADT using chromatic trees. Due to space restrictions, we only sketch the algorithm and its correctness proof. All details of the implementation and its correctness proof are in the full version of the paper. The ordered dictionary stores a set of keys, each with an associated value, where the keys are drawn from a totally ordered universe. The dictionary supports five operations. If key is in the dictionary, GET(key) returns its associated value. Otherwise, GET(key) returns $\perp$. SUCCESSOR(key) returns the smallest key in the dictionary that is larger than key (and its associated value), or $\perp$ if no key in the dictionary is larger than key. PREDECESSOR(key) is analogous. INSERT$(key, value)$ replaces the value associated with key by $value$ and returns the previously associated value, or $\perp$ if key was not in the dictionary. If the dictionary contains key, DELETE(key) removes it and returns the value that was associated immediately beforehand. Otherwise, DELETE(key) simply returns $\perp$.

A RBT is a BST in which the root and all leaves are coloured black, and every other node is coloured either red or black, subject to the constraints that no red node has a red parent, and the number of black nodes on a path from the root to a leaf is the same for all leaves. These properties guarantee that the height of a RBT is logarithmic in the

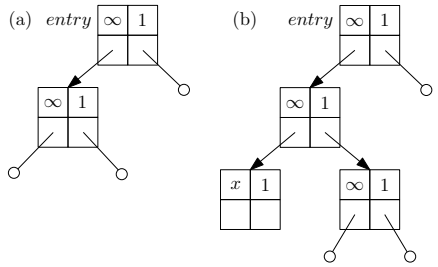

Figure 4. (a) empty tree, (b) non-empty tree.

number of nodes it contains. We consider search trees that are leaf-oriented, meaning the dictionary keys are stored in the leaves, and internal nodes store keys that are used only to direct searches towards the correct leaf. In this context, the BST property says that, for each node x, all descendants of x's left child have keys less than x's key and all descendants of x's right child have keys that are greater than *or equal to* x's key.

To decouple rebalancing steps from insertions and deletions, so that each is localized, and rebalancing steps can be interleaved with insertions and deletions, it is necessary to relax the balance properties of RBTs. A *chromatic tree* [27] is a relaxed-balance RBT in which colours are replaced by non-negative integer weights, where weight zero corresponds to red and weight one corresponds to black. As in RBTs, the sum of the weights on each path from the root to a leaf is the same. However, RBT properties can be violated in the following two ways. First, a red child node may have a red parent, in which case we say that a *red-red violation* occurs at this child. Second, a node may have weight $w > 1$, in which case we say that $w - 1$ *overweight violations* occur at this node. The root always has weight one, so no violation can occur at the root.

To avoid special cases when the chromatic tree is empty, we add sentinel nodes at the top of the tree (see Figure 4). The sentinel nodes and $entry$ have key ∞ to avoid special cases for SEARCH, INSERT and DELETE, and weight one to avoid special cases for rebalancing steps. Without having a special case for INSERT, we automatically get the two sentinel nodes in Figure 4(b), which also eliminate special cases for DELETE. The chromatic tree is rooted at the leftmost grandchild of $entry$. The sum of weights is the same for all paths from the root of the chromatic tree to its leaves, but not for paths that include $entry$ or the sentinel nodes.

Rebalancing steps are localized updates to a chromatic tree that are performed at the location of a violation. Their goal is to eventually eliminate all red-red and overweight violations, while maintaining the invariant that the tree is a chromatic tree. If no rebalancing step can be applied to a chromatic tree (or, equivalently, the chromatic tree contains no violations), then it is a RBT. We use the set of rebalancing steps of Boyar, Fagerberg and Larsen [7], which have a number of desirable properties: No rebalancing step increases the number of violations in the tree, rebalancing steps can be performed in any order, and, after sufficiently many rebalancing steps, the tree will always become a RBT. Furthermore, in any sequence of insertions, deletions and rebalancing steps starting from an empty chromatic tree, the amortized number of rebalancing steps is at most three per insertion and one per deletion.

5.1 Implementation

We represent each node by a Data-record with two mutable child pointers, and immutable fields k, v and w that contain the node's key, associated value, and weight, respectively. The child pointers of a leaf are always NIL, and the value field of an internal node is always NIL.

GET, INSERT and DELETE each execute an auxiliary procedure, SEARCH(key), which starts at $entry$ and traverses nodes as in an ordinary BST search, using READs of child pointers until reaching a leaf, which it then returns (along with the leaf's parent and grandparent). Because of the sentinel nodes shown in Figure 4, the leaf's parent always exists, and the grandparent exists whenever the chromatic tree is non-empty. If it is empty, SEARCH returns NIL instead of the grandparent. We define the *search path* for key at any time to be the path that SEARCH(key) would follow, if it were done instantaneously. The GET(key) operation simply executes a SEARCH(key) and then returns the value found in the leaf if the leaf's key is key, or $\perp$ otherwise.

At a high level, INSERT and DELETE are quite similar to each other. INSERT(key, $value$) and DELETE(key) each perform SEARCH(key) and then make the required update at the leaf reached, in accordance with the tree update template. If the modification fails, then the operation restarts from scratch. If it succeeds, it may increase the number of violations in the tree by one, and the new violation occurs on the search path to key. If a new violation is created, then an auxiliary procedure CLEANUP is invoked to fix it before the INSERT or DELETE returns.

Detailed pseudocode for GET, SEARCH, DELETE and CLEANUP is given in Figure 5. (The implementation of INSERT is similar to that of DELETE, and its pseudocode is omitted due to lack of space.) Note that an expression of the form $P\ ?\ A : B$ evaluates to A if the predicate P evaluates to true, and B otherwise. The expression $x.y$, where x is a Data-record, denotes field y of x, and the expression $\&x.y$ represents a pointer to field y.

DELETE(key) invokes TRYDELETE to search for a leaf containing key and perform the localized update that actually deletes key and its associated value. The effect of TRYDELETE is illustrated in Figure 6. There, nodes drawn as squares are leaves, shaded nodes are in V, $\otimes$ denotes a node in R to be finalized, and $\oplus$ denotes a new node. The name of a node appears below it or to its left. The weight of a node appears to its right.

TRYDELETE first invokes SEARCH(key) to find the grandparent, n_0, of the leaf on the search path to key. If the grandparent does not exist, then the tree is empty (and

```
1  GET(key)
2    ⟨−, −, l⟩ := SEARCH(key)
3    return (key = l.k) ? l.v : NIL
4  SEARCH(key)
5    n0 := NIL; n1 := entry; n2 := entry.left
6    while n2 is internal
7      n0 := n1; n1 := n2
8      n2 := (key < n1.k) ? n1.left : n1.right
9    return ⟨n0, n1, n2⟩
10 DELETE(key)
11   do
12     result := TRYDELETE(key)
13   while result = FAIL
14   ⟨value, violation⟩ := result
15   if violation then CLEANUP(key)
16   return value
17 TRYDELETE(key)
18   ▷ If successful, returns ⟨value, violation⟩, where value is the
         value associated with key, or NIL if key was not in the
         dictionary, and violation indicates whether the deletion
         created a violation. Otherwise, FAIL is returned.
19   ⟨n0, −, −⟩ := SEARCH(key)
20   ▷ Special case: there is no grandparent of the leaf reached
21   if n0 = NIL then return ⟨NIL, FALSE⟩
22   ▷ Template iteration 0 (grandparent of leaf)
23   s0 := LLX(n0)
24   if s0 ∈ {FAIL, FINALIZED} then return FAIL
25   n1 := (key < s0.left.k) ? s0.left : s0.right
26   ▷ Template iteration 1 (parent of leaf)
27   s1 := LLX(n1)
28   if s1 ∈ {FAIL, FINALIZED} then return FAIL
29   n2 := (key < s1.left.k) ? s1.left : s1.right
30   ▷ Special case: key is not in the dictionary
31   if n2.k ≠ key then return ⟨⊥, FALSE⟩
32   ▷ Template iteration 2 (leaf)
33   s2 := LLX(n2)
34   if s2 ∈ {FAIL, FINALIZED} then return FAIL
35   n3 := (key < s1.left.k) ? s1.right : s1.left
36   ▷ Template iteration 3 (sibling of leaf)
37   s3 := LLX(n3)
38   if s3 ∈ {FAIL, FINALIZED} then return FAIL
39   ▷ Computing SCX-ARGUMENTS from locally stored values
40   w := (n1.k = ∞ or n0.k = ∞) ? 1 : n1.w + n3.w
41   new := new node with weight w, key n3.k, value n3.v, and
         children s3.left, s3.right
42   V := (key < s1.left.k) ? ⟨n0, n1, n2, n3⟩ : ⟨n0, n1, n3, n2⟩
43   R := (key < s1.left.k) ? ⟨n1, n2, n3⟩ : ⟨n1, n3, n2⟩
44   fld := (key < s0.left.k) ? &n0.left : &n0.right
45   if SCX(V, R, fld, new) then return ⟨n2.v, (w > 1)⟩
46   else return FAIL
47 CLEANUP(key)
48   ▷ Eliminates the violation created by an INSERT or DELETE of key
49   while TRUE
50     ▷ Save four last nodes traversed
51     n0 := NIL; n1 := NIL; n2 := entry; n3 := entry.left
52     while TRUE
53       if n3.w > 1 or (n2.w = 0 and n3.w = 0) then
54         ▷ Found a violation at node n3
55         TRYREBALANCE(n0, n1, n2, n3)          ▷ Try to fix it
56         exit loop          ▷ Go back to entry and search again
57       else if n3 is a leaf then return
58         ▷ Arrived at a leaf without finding a violation
59       if key < n3.k then
60         n0 := n1; n1 := n2; n2 := n3; n3 := n3.left
61       else n0 := n1; n1 := n2; n2 := n3; n3 := n3.right
```

Figure 5. GET, SEARCH, DELETE and CLEANUP.

it looks like Figure 4(a)), so TRYDELETE returns successfully at line 21. TRYDELETE then performs LLX(n_0), and uses the result to obtain a pointer to the parent, n_1, of the leaf to be deleted. Next, it performs LLX(n_1), and uses the result to obtain a pointer to the leaf, n_2, to be deleted. If n_2 does not contain key, then the tree does not contain key, and TRYDELETE returns successfully at line 31. So, suppose that n_2 does contain key. Then TRYDELETE performs LLX(n_2). At line 35, TRYDELETE uses the result of its previous LLX(n_1) to obtain a pointer to the sibling, n_3, of the leaf to be deleted. A final LLX is then performed on n_3. Over the next few lines, TRYDELETE computes SCX-ARGUMENTS. Line 40 computes the weight of the node new in the depiction of DELETE in Figure 4, ensuring that it has weight one if it is taking the place of a sentinel or the root of the chromatic tree. Line 41 creates new, reading the key, and value directly from n_3 (since they are immutable) and the child pointers from the result of the LLX(n_3) (since they are mutable). Next, TRYDELETE uses locally stored values to construct the sequences V and R that it will use for its SCX, ordering their elements according to a breadth-first traversal, in order to satisfy PC8. Finally, TRYDELETE invokes SCX to perform the modification. If the SCX succeeds, then TRYDELETE returns a pair containing the value stored in node n_2 (which is immutable) and the result of evaluating the expression $w > 1$.

DELETE can create an overweight violation (but not a red-red violation), so the result of $w > 1$ indicates whether TRYDELETE created a violation. If any LLX returns FAIL or FINALIZED, or the SCX fails, TRYDELETE simply returns FAIL, and DELETE invokes TRYDELETE again. If TRYDELETE creates a new violation, then DELETE invokes CLEANUP(key) (described in Section 5.2) to fix it before DELETE returns.

A simple inspection of the pseudocode suffices to prove that SCX-ARGUMENTS satisfies postconditions PC1 to PC9. TRYDELETE follows the template except when it returns at line 21 or line 31. In these cases, not following the template does not impede our efforts to prove correctness or progress, since TRYDELETE will not modify the data structure, and returning at either of these lines will cause DELETE to terminate.

We now describe how rebalancing steps are implemented from LLX and SCX, using the tree update template. As an example, we consider one of the 22 rebalancing steps, named RB2 (shown in Figure 6), which eliminates a red-red violation at node n_3. The other 21 are implemented similarly. To implement RB2, a sequence of LLXs are performed, starting with node n_0. A pointer to node n_1 is obtained from the result of LLX(n_0), pointers to nodes n_2 and f_3 are obtained from the result of LLX(n_1), and a pointer to node n_3 is obtained from the result of LLX(n_2). Since node n_3 is to be removed from the tree, an LLX is performed on it, too. If any of these LLXs returns FAIL or FINALIZED, then this update

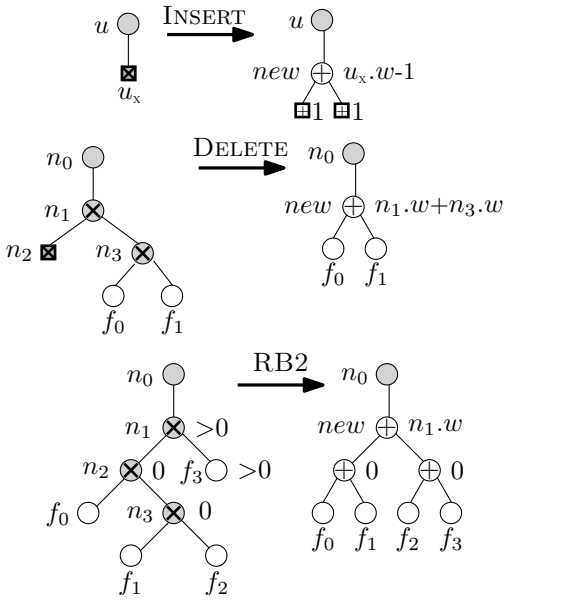

Figure 6. Examples of chromatic tree updates.

fails. For RB2 to be applicable, n_1 and f_3 must have positive weights and n_2 and n_3 must both have weight 0. Since weight fields are immutable, they can be read any time after the pointers to n_1, f_3, n_2, and n_3 have been obtained. Next, new and its two children are created. N consists of these three nodes. Finally, SCX(V, R, fld, new) is invoked, where fld is the child pointer of n_0 that pointed to n_1 in the result of LLX(n_0).

If the SCX modifies the tree, then no node $r \in V$ has changed since the update performed LLX(r). In this case, the SCX replaces the directed graph G_R by the directed graph G_N and the nodes in R are finalized. This ensures that other updates cannot erroneously modify these old nodes after they have been replaced. The nodes in the set $F_R = F_N = \{f_0, f_1, f_2, f_3\}$ each have the same keys, weights, and child pointers before and after the rebalancing step, so they can be reused. $V = \langle n_0, n_1, n_2, n_3\rangle$ is simply the sequence of nodes on which LLX is performed, and $R = \langle n_1, n_2, n_3\rangle$ is a subsequence of V, so PC1, PC2 and PC3 are satisfied. Clearly, we satisfy PC4 and PC7 when we create new and its two children. It is easy to verify that PC5, PC6 and PC9 are satisfied. If the tree does not change during the update, then the nodes in V are ordered consistently with a breadth-first traversal of the tree. Since this is true for all updates, PC8 is satisfied.

One might wonder why f_3 is not in V, since RB2 should be applied only if n_1 has a right child with positive weight. Since weight fields are immutable, the only way that this can change after we check $f_3.w > 0$ is if the right child field of n_1 is altered. If this happens, the SCX will fail.

5.2 The rebalancing algorithm

Since rebalancing is decoupled from updating, there must be a scheme that determines when processes should perform rebalancing steps to eliminate violations. In [7], the authors suggest maintaining one or more *problem queues* which contain pointers to nodes that contain violations, and dedicating one or more *rebalancing processes* to simply perform rebalancing steps as quickly as possible. This approach does not yield a bound on the height of the tree, since rebalancing may lag behind insertions and deletions. It is possible to obtain a height bound with a different queue based scheme, but we present a way to bound the tree's height without the (significant) overhead of maintaining any auxiliary data structures. The linchpin of our method is the following claim concerning violations.

VIOL: If a violation is on the search path to key before a rebalancing step, then the violation is still on the search path to key after the rebalancing step, or it has been eliminated.

While studying the rebalancing steps in [7], we realized that most of them satisfy VIOL. Furthermore, any time a rebalancing step would violate VIOL another rebalancing step that satisfies VIOL can be applied instead. Hence, we always choose to perform rebalancing so that each violation created by an INSERT(key) or DELETE(key) stays on the search path to key until it is eliminated. In our implementation, each INSERT or DELETE that increases the number of violations cleans up after itself. It does this by invoking a procedure CLEANUP(key), which behaves like SEARCH(key) until it finds the first node n_3 on the search path where a violation occurs. Then, CLEANUP(key) attempts to eliminate or move the violation at n_3 by invoking another procedure TRYREBALANCE which applies one localized rebalancing step at n_3, following the tree update template. (TRYREBALANCE is similar to DELETE, and pseudocode is omitted, due to lack of space.) CLEANUP(key) repeats these actions, searching for key and invoking TRYREBALANCE to perform a rebalancing step, until the search goes all the way to a leaf without finding a violation.

In order to prove that each INSERT or DELETE cleans up after itself, we must prove that while an invocation of CLEANUP(key) searches for key by reading child pointers, it does not somehow miss the violation it is responsible for eliminating, even if a concurrent rebalancing step moves the violation upward in the tree, above where CLEANUP is currently searching. To see why this is true, consider any rebalancing step that occurs while CLEANUP is searching. The rebalancing step is implemented using the tree update template, and looks like Figure 1. It takes effect at the point it changes a child pointer fld of some node $parent$ from a node old to a node new. If CLEANUP reads fld while searching, we argue that it does not matter whether fld contains old or new. First, suppose the violation is at a node that is removed from the tree by the rebalancing step, or a child of such a node. If the search passes through old, it will definitely reach the violation, since nodes do not change after they are removed from the tree. If the search passes through

new, VIOL implies that the rebalancing step either eliminated the violation, or moved it to a new node that will still be on the search path through new. Finally, if the violation is further down in the tree, below the section modified by the concurrent rebalancing step, a search through either old or new will reach it.

Showing that TRYREBALANCE follows the template (i.e., by defining the procedures in Figure 3) is complicated by the fact that it must decide which of the chromatic tree's 22 rebalancing steps to perform. It is more convenient to unroll the loop that performs LLXs, and write TRYREBALANCE using conditional statements. A helpful technique is to consider each path through the conditional statements in the code, and check that the procedures CONDITION, NEXTNODE, SCX-ARGUMENTS and RESULT can be defined to produce this single path. It is sufficient to show that this can be done for each path through the code, since it is always possible to use conditional statements to combine the procedures for each path into procedures that handle all paths.

5.3 Proving a bound on the height of the tree

Since we always perform rebalancing steps that satisfy VIOL, if we reach a leaf without finding the violation that an INSERT or DELETE created, then the violation has been eliminated. This allows us to prove that the number of violations in the tree at any time is bounded above by c, the number of insertions and deletions that are currently in progress. Further, since removing all violations would yield a red-black tree with height $O(\log n)$, and eliminating each violation reduces the height by at most one, the height of the chromatic tree is $O(c + \log n)$.

5.4 Correctness and Progress

As mentioned above, GET(key) invokes SEARCH(key), which traverses a path from $entry$ to a leaf by reading child pointers. Even though this search can pass through nodes that have been removed by concurrent updates, we prove by induction that every node visited *was* on the search path for key at some time during the search. GET can thus be linearized when the leaf it reaches is on the search path for key (and, hence, this leaf is the only one in the tree that could contain key).

Every DELETE operation that performs an update, and every INSERT operation, is linearized at the SCX that performs the update. Other DELETE operations (that return at line 21 or 31) behave like queries, and are linearized in the same way as GET. Because no rebalancing step modifies the set of keys stored in leaves, the set of leaves always represents the set of dictionary entries.

The fact that our chromatic tree is non-blocking follows from P1 and the fact that at most $3i + d$ rebalancing steps can be performed after i insertions and d deletions have occurred (proved in [7]).

5.5 SUCCESSOR queries

SUCCESSOR(key) runs an ordinary BST search algorithm, using LLXs to read the child fields of each node visited, until it reaches a leaf. If the key of this leaf is larger than key, it is returned and the operation is linearized at any time during the operation when this leaf was on the search path for key. Otherwise, SUCCESSOR finds the next leaf. To do this, it remembers the last time it followed a left child pointer and, instead, follows one right child pointer, and then left child pointers until it reaches a leaf, using LLXs to read the child fields of each node visited. If any LLX it performs returns FAIL or FINALIZED, SUCCESSOR restarts. Otherwise, it performs a validate-extended (VLX), which returns TRUE only if all nodes on the path connecting the two leaves have not changed. If the VLX succeeds, the key of the second leaf found is returned and the query is linearized at the linearization point of the VLX. If the VLX fails, SUCCESSOR restarts.

5.6 Allowing more violations

Forcing insertions and deletions to rebalance the chromatic tree after creating only a single violation can cause unnecessary rebalancing steps to be performed, for example, because an overweight violation created by a deletion might be eliminated by a subsequent insertion. In practice, we can reduce the total number of rebalancing steps that occur by modifying our INSERT and DELETE procedures so that CLEANUP is invoked only once the number of violations on a path from $entry$ to a leaf exceeds some constant k. The resulting data structure has height $O(k + c + \log n)$. Since searches in the chromatic tree are extremely fast, slightly increasing search costs to reduce update costs can yield significant benefits for update-heavy workloads.

6. Experimental Results

We compared the performance of our chromatic tree (Chromatic) and the variant of our chromatic tree that invokes CLEANUP only when the number of violations on a path exceeds six (Chromatic6) against several leading data structures that implement ordered dictionaries: the non-blocking skip-list (SkipList) of the Java Class Library, the non-blocking multiway search tree (SkipTree) of Spiegel and Reynolds [29], the lock-based relaxed-balance AVL tree with non-blocking searches (AVL-D) of Drachsler et al. [14], and the lock-based relaxed-balance AVL tree (AVL-B) of Bronson et al. [9]. Our comparison also includes an STM-based red-black tree optimized by Oracle engineers (RB-STM) [19], an STM-based skip-list (SkipListSTM), and the highly optimized Java red-black tree, `java.util.TreeMap`, with operations protected by a global lock (RBGlobal). The STM data structures are implemented using DeuceSTM 1.3.0, which is one of the fastest STM implementations that does not require modifications to the Java virtual machine. We used DeuceSTM's offline instrumentation ca-

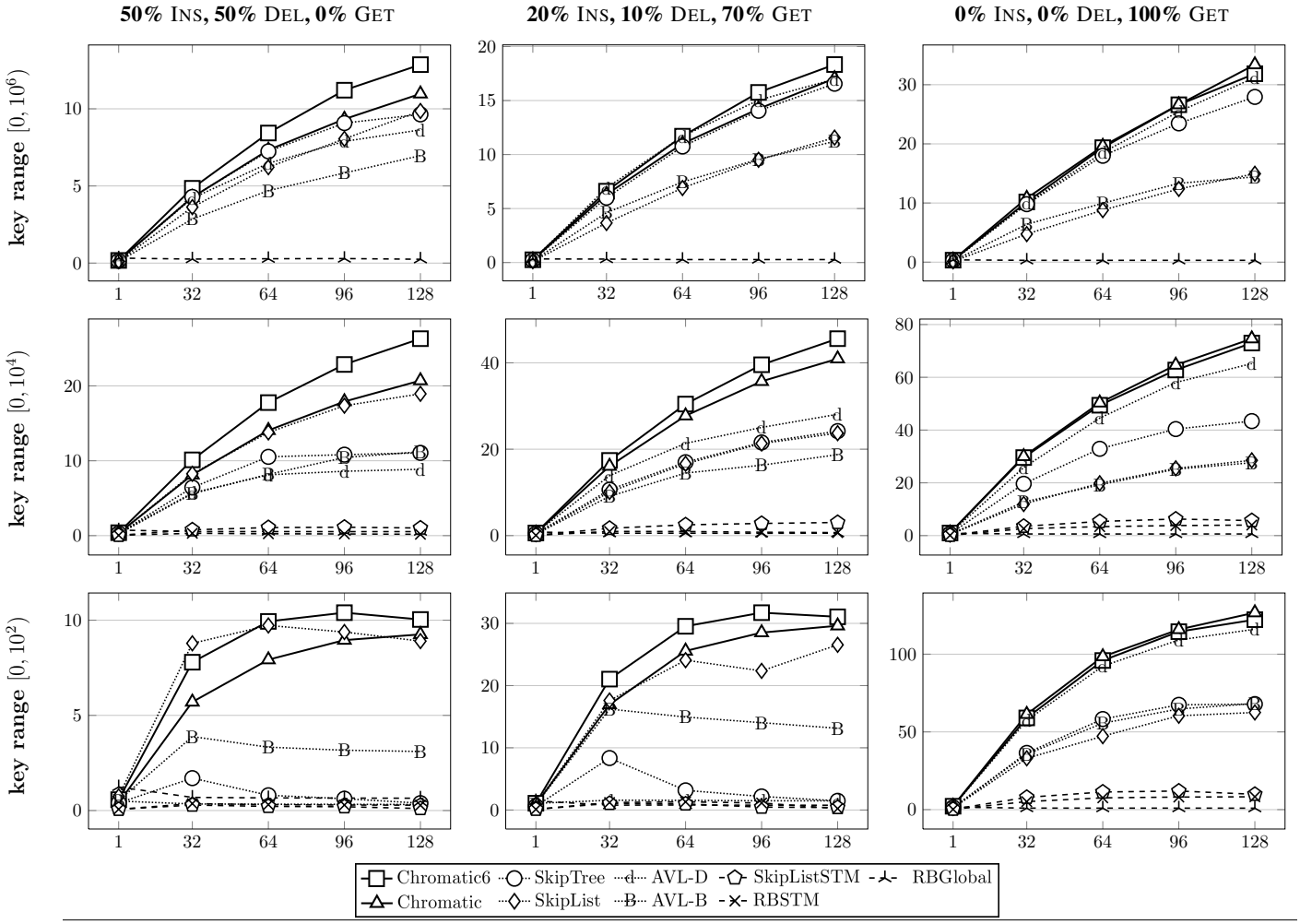

Figure 7. *Multithreaded* throughput (millions of operations/second) for 2-socket SPARC T2+ (128 hardware threads) on y-axis versus number of threads on x-axis.

pability to eliminate any STM instrumentation at running time that might skew our results. All of the implementations that we used were made publicly available by their respective authors. For a fair comparison between data structures, we made slight modifications to RBSTM and SkipListSTM to use generics, instead of hardcoding the type of keys as `int`, and to store values in addition to keys. Java code for Chromatic and Chromatic6 is available from http://www.cs.utoronto.ca/~tabrown.

We tested the data structures for three different operation mixes, 0i-0d, 20i-10d and 50i-50d, where xi-yd denotes $x\%$ INSERTS, $y\%$ DELETES, and $(100 - x - y)\%$ GETS, to represent the cases when all of the operations are queries, when a moderate proportion of the operations are INSERTS and DELETES, and when all of the operations are INSERTS and DELETES. We used three key ranges, $[0, 10^2), [0, 10^4)$ and $[0, 10^6)$, to test different contention levels. For example, for key range $[0, 10^2)$, data structures will be small, so updates are likely to affect overlapping parts of the data structure.

For each data structure, each operation mix, each key range, and each thread count in $\{1, 32, 64, 96, 128\}$, we ran five trials which each measured the total throughput (operations per second) of all threads for five seconds. Each trial began with an untimed prefilling phase, which continued until the data structure was within 5% of its expected size in the steady state. For operation mix 50i-50d, the expected size is half of the key range. This is because, eventually, each key in the key range has been inserted or deleted at least once, and the last operation on any key is equally likely to be an insertion (in which case it is in the data structure) or a deletion (in which case it is not in the data structure). Similarly, 20i-10d yields an expected size of two thirds of the key range since, eventually, each key has been inserted or deleted and the last operation on it is twice as likely to be an insertion as a deletion. For 0i-0d, we prefilled to half of the key range.

We used a Sun SPARC Enterprise T5240 with 32GB of RAM and two UltraSPARC T2+ processors, for a total of 16 1.2GHz cores supporting a total of 128 hardware threads. The Sun 64-bit JVM version 1.7.0_03 was run in server

mode, with 3GB minimum and maximum heap sizes. Different experiments run within a single instance of a Java virtual machine (JVM) are not statistically independent, so each batch of five trials was run in its own JVM instance. Prior to running each batch, a fixed set of three trials was run to cause the Java HotSpot compiler to optimize the running code. Garbage collection was manually triggered before each trial. The heap size of 3GB was small enough that garbage collection was performed regularly (approximately ten times) in each trial. We did not pin threads to cores, since this is unlikely to occur in practice.

Figure 7 shows our experimental results. Our algorithms are drawn with solid lines. Competing handcrafted implementations are drawn with dotted lines. Implementations with coarse-grained synchronization are drawn with dashed lines. Error bars are not drawn because they are mostly too small to see: The standard deviation is less than 2% of the mean for half of the data points, and less than 10% of the mean for 95% of the data points. The STM data structures are not included in the graphs for key range $[0, 10^6)$, because of the enormous length of time needed just to perform prefilling (more than 120 seconds per five second trial).

Chromatic6 nearly always outperforms Chromatic. The only exception is for an all query workload, where Chromatic performs slightly better. Chromatic6 is prefilled with the Chromatic6 insertion and deletion algorithms, so it has a slightly larger average leaf depth than Chromatic; this accounts for the performance difference. In every graph, Chromatic6 rivals or outperforms the other data structures, even the highly optimized implementations of SkipList and SkipTree which were crafted with the help of Doug Lea and the Java Community Process JSR-166 Expert Group. Under high contention (key range $[0, 10^2)$), Chromatic6 outperforms every competing data structure except for SkipList in case 50i-50d and AVL-D in case 0i-0d. In the former case, SkipList approaches the performance of Chromatic6 when there are many INSERTs and DELETEs, due to the simplicity of its updates. In the latter case, the non-blocking searches of AVL-D allow it to perform nearly as well as Chromatic6; this is also evident for the other two key ranges. SkipTree, AVL-D and AVL-B all experience negative scaling beyond 32 threads when there are updates. For SkipTree, this is because its nodes contain many child pointers, and processes modify a node by replacing it (severely limiting concurrency when the tree is small). For AVL-D and AVL-B, this is likely because processes waste time waiting for locks to be released when they perform updates. Under moderate contention (key range $[0, 10^4)$), in cases 50i-50d and 20i-10d, Chromatic6 significantly outperforms the other data structures. Under low contention, the advantages of a non-blocking approach are less pronounced, but Chromatic6 is still at the top of each graph (likely because of low overhead and searches that ignore updates).

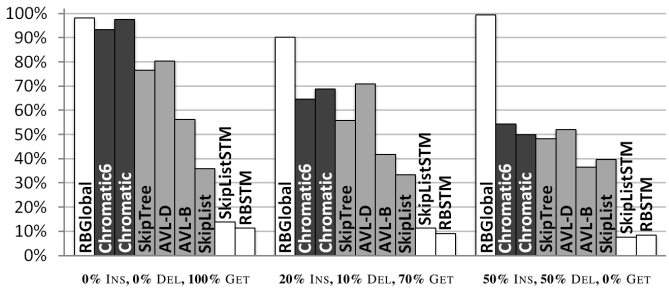

Figure 8. *Single threaded* throughput of the data structures relative to Java's sequential RBT for key range $[0, 10^6)$.

Figure 8 compares the single-threaded performance of the data structures, relative to the performance of the sequential RBT, `java.util.TreeMap`. This demonstrates that the overhead introduced by our technique is relatively small.

Although balanced BSTs are designed to give performance guarantees for worst-case sequences of operations, the experiments are performed using random sequences. For such sequences, BSTs without rebalancing operations are balanced with high probability and, hence, will have better performance because of their lower overhead. Better experiments are needed to evaluate balanced BSTs.

7. Conclusion

In this work, we presented a template that can be used to obtain non-blocking implementations of any data structure based on a down-tree, and demonstrated its use by implementing a non-blocking chromatic tree. To the authors' knowledge, this is the first provably correct, non-blocking balanced BST with fine-grained synchronization. Proving the correctness of a direct implementation of a chromatic tree from hardware primitives would have been completely intractable. By developing our template abstraction and our chromatic tree in tandem, we were able to avoid introducing any extra overhead, so our chromatic tree is very efficient.

Given a copy of [23], and this paper, a first year undergraduate student produced our Java implementation of a relaxed-balance AVL tree in less than a week. Its performance was slightly lower than that of Chromatic. After allowing more violations on a path before rebalancing, its performance was indistinguishable from that of Chromatic6.

We hope that this work sparks interest in developing more relaxed-balance sequential versions of data structures, since it is now easy to obtain efficient concurrent implementations of them using our template.

Acknowledgements

This work was supported by NSERC. We thank Oracle for providing the machine used for our experiments.

References

[1] Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, and R. Tarjan. CBTree: A practical concurrent self-adjusting search tree. In

Proc. 26th International Symposium on Distributed Computing, volume 7611 of *LNCS*, pages 1–15, 2012.

[2] Z. Aghazadeh, W. Golab, and P. Woelfel. Brief announcement: Resettable objects and efficient memory reclamation for concurrent algorithms. In *Proc. 32nd ACM Symposium on Principles of Distributed Computing*, pages 322–324, 2013.

[3] J. H. Anderson and M. Moir. Universal constructions for multi-object operations. In *Proc. 14th ACM Symposium on Principles of Distributed Computing*, pages 184–193, 1995.

[4] G. Barnes. A method for implementing lock-free data structures. In *Proc. 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.

[5] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

[6] L. Bougé, J. Gabarró, X. Messeguer, and N. Schabanel. Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries. Technical Report LSI-98-12-R, Universitat Politècnica de Catalunya, 1998. Available from http://www.lsi.upc.edu/dept/techreps/llistat_detallat.php?id=307.

[7] J. Boyar, R. Fagerberg, and K. S. Larsen. Amortization results for chromatic search trees, with an application to priority queues. *J. Comput. Syst. Sci.*, 55(3):504–521, Dec. 1997.

[8] A. Braginsky and E. Petrank. A lock-free B+tree. In *Proc. 24th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 58–67, 2012.

[9] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proc. 15th ACM Symposium on Principles and Practice of Parallel Programming*, pages 257–268, 2010.

[10] T. Brown, F. Ellen, and E. Ruppert. Pragmatic primitives for non-blocking data structures. In *Proc. 32nd ACM Symposium on Principles of Distributed Computing*, 2013. Full version available from http://www.cs.utoronto.ca/~tabrown.

[11] T. Brown and J. Helga. Non-blocking k-ary search trees. In *Proc. 15th International Conf. on Principles of Distributed Systems*, volume 7109 of *LNCS*, pages 207–221, 2011.

[12] T. Crain, V. Gramoli, and M. Raynal. A speculation-friendly binary search tree. In *Proc. 17th ACM Symp. on Principles and Practice of Parallel Programming*, pages 161–170, 2012.

[13] T. Crain, V. Gramoli, and M. Raynal. A contention-friendly binary search tree. In *Euro-Par*, pages 229–240, 2013.

[14] D. Drachsler, M. Vechev, and E. Yahav. Practical concurrent binary search trees via logical ordering. In these proceedings, 2014.

[15] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proc. 29th ACM Symposium on Principles of Distributed Computing*, pages 131–140, 2010. Full version available as Technical Report CSE-2010-04, York University.

[16] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. on Computer Systems*, 25(2):5, 2007.

[17] K. A. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, 2003.

[18] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[19] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proc. 22nd ACM Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

[20] S. V. Howley and J. Jones. A non-blocking internal binary search tree. In *Proc. 24th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 161–171, 2012.

[21] H. Kung and P. L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, 1980.

[22] K. S. Larsen. Amortized constant relaxed rebalancing using standard rotations. *Acta Informatica*, 35(10):859–874, 1998.

[23] K. S. Larsen. AVL trees with relaxed balance. *J. Comput. Syst. Sci.*, 61(3):508–522, Dec. 2000.

[24] K. S. Larsen, T. Ottmann, and E. Soisalon-Soininen. Relaxed balance for search trees with local rebalancing. *Acta Informatica*, 37(10):743–763, 2001.

[25] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In these proceedings, 2014.

[26] A. Natarajan, L. Savoie, and N. Mittal. Concurrent wait-free red black trees. In *Proc. 15th International Symposium on Stabilization, Safety and Security of Distributed Systems*, volume 8255 of *LNCS*, pages 45–60, 2013.

[27] O. Nurmi and E. Soisalon-Soininen. Chromatic binary search trees: A structure for concurrent rebalancing. *Acta Informatica*, 33(6):547–557, 1996.

[28] N. Shafiei. Non-blocking Patricia tries with replace operations. In *Proc. 33rd International Conference on Distributed Computing Systems*, pages 216–225, 2013.

[29] M. Spiegel and P. F. Reynolds, Jr. Lock-free multiway search trees. In *Proc. 39th International Conference on Parallel Processing*, pages 604–613, 2010.

[30] J.-J. Tsay and H.-C. Li. Lock-free concurrent tree structures for multiprocessor systems. In *Proc. International Conference on Parallel and Distributed Systems*, pages 544–549, 1994.

[31] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. In *Proc. 11th ACM Symposium on Principles of Database Systems*, pages 212–222, 1992.

[32] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. 14th ACM Symposium on Principles of Distributed Computing*, pages 214–222, 1995.