A Lock-Free B+tree *

Anastasia Braginsky
Dept. of Computer Science
Technion - Israel Institute of Technology
Haifa 32000, Israel
anastas@cs.technion.ac.il

Erez Petrank
Dept. of Computer Science
Technion - Israel Institute of Technology
Haifa 32000, Israel
erez@cs.technion.ac.il

ABSTRACT

Lock-free data structures provide a progress guarantee and are known for facilitating scalability, avoiding deadlocks and livelocks, and providing guaranteed system responsiveness. In this paper we present a design for a lock-free balanced tree, specifically, a B$^+$tree. The B$^+$tree data structure has an important practical applications, and is used in various storage-system products. As far as we know this is the first design of a lock-free, dynamic, and balanced tree, that employs standard compare-and-swap operations.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures—*trees, distributed data structures*; D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*

General Terms

Algorithms, Design, Theory

Keywords

Concurrent Data Structures, Progress Guarantee, Lock-Freedom, B$^+$tree, Parallel Programming

1. INTRODUCTION

The growing popularity of parallel computing is accompanied by an acute need for data structures that execute efficiently and provide guaranteed progress on parallel platforms. Lock-free data structures provide a progress guarantee: if the program threads are run sufficiently long, then at least one of them must make progress. This ensures that the program as a whole progresses and is never blocked. Although lock-free algorithms exist for various data structures, lock-free balanced trees have been considered difficult to construct and as far as we know a construction for a lock-free balanced tree is not known.

In recent decades, the B-tree has been the data structure of choice for maintaining searchable, ordered data on disk. Traditional B-trees are effective in large part because they minimize the number of disk blocks accessed during a search. When using a B-tree on the computer memory, a reasonable choice is to keep a node on a single cache line. However, some studies show that a block size that is a (small) factor of the processor's cache line can deliver better performance if cache pre-fetching is employed by the hardware [14, 5]. Further details about the B-Tree structure and the B$^+$tree variant appear in Subsection 2.1.

This paper presents the first lock-free, linearizable, dynamic B$^+$tree implementation supporting searches, insertions, and deletions. It is dynamic in the sense that there is no (static) limit to the number of nodes that can be allocated and put in the tree. The construction employs only reads, writes, and (single-word) CAS instructions. Searches are not delayed by rebalancing operations. The construction employs the lock-free chunk mechanism proposed in [4]. The chunk mechanism provides a lock-free linked list that resides on a consecutive chunk of memory and maintains lower- and upper-bound on the number of elements. The chunks are split or joined with other chunks to maintain the bounds in the presence of insertions and deletions. This lock-free chunk mechanism fits naturally with a node of the B$^+$tree that is split and joined, keeping the number of elements within given bounds, and thus maintaining the balance of the tree.

Our construction follows some basic design decisions that reduce the complexity of the algorithm. First, a node marked by the need to join or split is frozen, and no more operations are allowed on it. It is never resurrected, and one or two nodes are allocated to replace it. This eliminates much of the difficulty with threads waking up after a long idle period and encountering an old node that has been split or joined. In general, a node begins its lifespan as an infant, proceeds to become a normal node, and remains so until frozen for a split or a join, after which it is eventually reclaimed. This monotonic progress, reflected in the node's state, simplifies the design. The replacement of old nodes with new ones is challenging as data may be held in both the old and the new nodes simultaneously. To allow lock-freedom, we let the search operation dive into old nodes as well as new ones. But to ensure linearizability, we only allow new nodes to be modified after the replacement procedure is completed. Ad-

*Supported by THE ISRAEL SCIENCE FOUNDATION (grant No. 283/10).

SPAA'12, June 25–27, 2012, Pittsburgh, Pennsylvania, USA.

ditionally, we take special care in the selection of a neighboring node to join with, to ensure that it cooperates correctly. Finally, we enforce the invariant that two join nodes always have the same parent. Our construction follows important lock-free techniques that have been previously used. In particular, we mark pointers to signify deletion following Harris [9], we assign nodes with states similarly to Ellen et al. [7]. We also propose new techniques that might be useful for future work, e.g., we use a gradual state transition for a node by gradually moving it from the *normal* to the *frozen* state, by marking its fields one by one as frozen.

This design of the lock-free B$^+$tree is meant to show the feasibility of a lock-free balanced tree. It is quite complex and we have not added (even straightforward) optimizations. We implemented the design (as is) in C and ran it against an implementation of a standard lock-based B$^+$tree [15]. The results show that the lock-based version wins when no contention exists or the contention is very low. However, as contention kicks in, the lock-free B$^+$tree behaves much better than the lock-based version. The lock-free tree is highly scalable and allows good progress even when many threads are executing concurrently. Similarly to the lock-free algorithm of the linked-list, a wait-free variant of the search method (denoted *contains*) can be defined here in the same manner. Again, to keep it simple, we do not spell it out.

In addition to implementing and measuring the algorithm, we also provide the full proof for the correctness of this design in the full version of this paper [3] with respect to linearizability [11] and (bounded) lock-freedom [10, 13]. Note that a balanced tree has a better worst-case behavior compared to regular trees. Ignoring concurrency, each operation has a worst-case complexity of $O(\log n)$ in contrast to a worst-case complexity of $O(n)$ for an imbalanced tree. Furthermore, in the presence of concurrent threads, we prove that progress must be made at worst-case within $O(T \log n + T^2)$ computational steps, where T is number of the concurrent running threads and n is number of keys in the B$^+$tree. (This means bounded lock-freedom with bound $O(T \log n + T^2)$.) Such guarantee can only be achieved with balanced trees, as computing a similar bound on the worst-case time to make progress in a non-balanced tree would yield $O(Tn)$[1].

Previous work on lock-free trees include Fraser's construction [8] of a lock-free balanced tree that builds on a transactional memory system. Our work does not require any special underlying system support. Fraser also presents a construction of a lock-free tree that uses multiple-word CAS [8], but this construction offers no balancing and at worst may require a linear complexity for the tree operations. Recently, Ellen *et al.* [7] presented a lock-free tree using a single-word CAS, but their tree offers no balancing. Bender *et al.* [2] described a lock-free implementation of a cache-oblivious B-tree from LL/SC operations. Our construction uses single-word CAS operations. Moreover, a packed-memory cache-oblivious B-tree is not equivalent to the traditional B$^+$tree data structure. First, it only guarantees amortized time complexity (even with no contention), as the data is kept in an array that needs to be extended occasionally by copying the entire data structure. Second, it does not keep the shallow structure and is thus not suitable for use with file systems. Finally, a full version of [2] has not yet appeared and some details of lock-free implementation are not specified.

In Section 2 we set up some preliminaries and present the B$^+$tree representation in the memory together with the basic B$^+$tree algorithms. In Section 3 we describe the B$^+$tree node's states and recall the lock-free chunk functionality from [4]. Balancing functions are presented in Section 4, and the implementation and results are described in Section 5. In Section 6 we describe the linearization points. We conclude in Section 7. In the full version of this paper [3] more details and the entire pseudo-code can be found. In addition, the full correctness, linearizability and bounded lock-freedom proof is presented in [3] as well.

2. BACKGROUND AND DATA STRUCTURE

This section presents the underlying data structures used to implement the lock-free B$^+$tree, starting with a review of the lock-free chunk mechanism presented in [4].

A *chunk* is a (consecutive) block of memory that contains *entries*. Each entry contains a key and a data field, and the entries are stored in the chunk as a key-ordered linked list. A chunk consumes a fixed amount of space and has two parameters, determining the minimum and maximum entries that may reside in it. The chunk supports set operations such as *search*, *insert* and *delete*. When an insert of a new entry increases the number of entries above the maximum, a *split* is executed and two chunks are created from the original chunk. Similarly, when a deletion violates the minimum number of entries, the chunk mechanism *joins* this chunk and another chunk, obtained from the data structure using the chunks (in particular the B$^+$tree). Therefore, the B$^+$tree implements a method that the chunk can call to obtain a partner to join with. A different B$^+$tree method is called by the chunk mechanism when the split or join are completed to ask that the tree replaces the frozen nodes with new ones. The chunk also supports an additional *replace* operation that allows replacing the data of an entry with a new value atomically without modifying the entry's location in the list. This operation is useful for switching a descendant without modifying the key associated with it[2]. All operations are lock-free.

2.1 The B$^+$tree

A B$^+$tree [6] is a balanced tree used to maintain a set of *keys*, and a mapping from each key to its associated *data*. Each node of the tree holds entries; each entry has a key and an auxiliary data. In contrast to a B-tree, only the leaves in a B$^+$tree hold the keys and their associated data. The data of the keys in the internal nodes is used to allow navigating through the tree. Thus, data in an internal node of the tree contains pointers to descendants of the internal node. The B$^+$tree structure simplifies the tree insertions and deletions and is commonly used for concurrent access. In our variant of a B$^+$tree, key repetition is not allowed.

Each internal node consists of an ordered list of entries containing keys and their associated pointers. A tree search starts at the root and chooses a descendant according to the values of the keys, the convention being that the entry's key provides the upper bound on the set of keys in its subtree.

[1]Actually, we do not know how to show a lock-free bound which is lower than $O(T^2 n)$ for non-balanced concurrent trees.

[2]The replace operation did not appear in the short conference version of [4] and is described in [3].

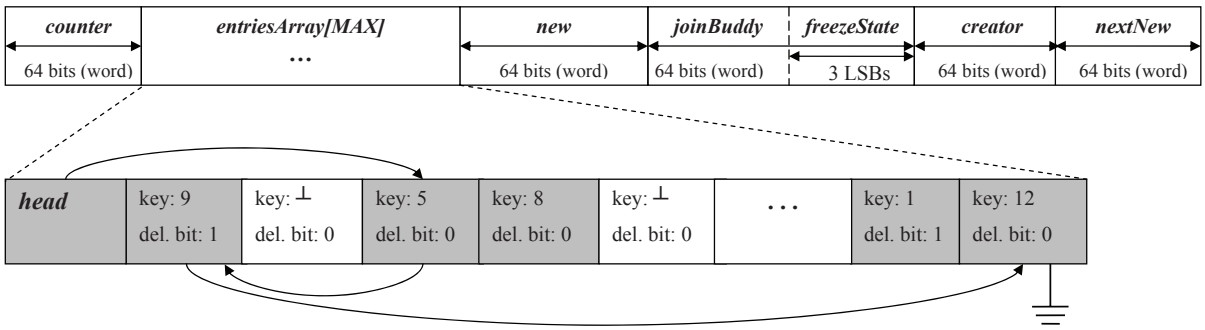

Figure 1: The structure of a chunk. The allocated grey entries present the ordered linked list.

Each node has a minimum and maximum number of possible entries in it. In our B$^+$tree the maximum is assumed to be even and is denoted d. The minimum is set to $d/2-3$. For $d \geq 10$ this ensures the balance of the tree, and specifically that the number of nodes to be read before reaching a leaf is bounded by a logarithm of the tree size. All insertions and deletions happen at leaves. When an insert violates the maximum allowed number of entries in the node, a split is performed on that node. When a deletion violates the minimum allowed number of entries, the algorithm attempts to join two nodes, resulting in borrowing entries from a neighboring node or merging the two nodes, if moving entries is not possible.

Splitting and joining leaves may, in turn, imply an insert or a delete to the parent, and such an update may roll up until the root. We ignore the minimum number of entries on the root, in order not to enforce a minimal number of entries in the tree. Note that splits and joins always create nodes with a legitimate number of entries. In practice, the minimum value is sometimes set to be smaller than $d/2-3$ to avoid frequent splits and joins.

2.2 The structure of the proposed B$^+$tree

For simplicity, our construction assumes the key and the data fit into a single word. This is the assumption of the chunk mechanism and it makes the allocation of a new entry easier. In practice, this means a word of 64 bits, with a key of 32 bits and data of 32 bits.[3] An architecture that provides a double-word compare-and-swap would allow using a full word for each of the fields, removing the restrictions, and simplifying the construction. The key values are taken from a finite set, bounded from above by a value that we denote ∞. The tree is represented by a pointer to the root node, initially set to an empty root-leaf node.

Our B$^+$tree node is built using the chunk structure of [4]. The chunk's maximum and minimum number of entries are set to d and $d/2-3$ to satisfy the B$^+$tree node requirement (except for the zero minimum bound on the root). In addition to a chunk, the tree node contains two additional fields to support its management: a *height* field indicating the distance from the leaves and a *root* flag indicating whether the node is a root.

We briefly review the fields of a chunk (Figure 1). A detailed discussion appears in [4]. The main part of the chunk is an array that contains the entries. The *counter* field counts the number of entries in a chunk. It is accurate during sequential execution and is always guaranteed to hold a lower bound on the real count, even in the presence of concurrent executions. The pointers *new*, *joinBuddy*, *nextNew* and *creator* point to nodes involved in the rebalancing, to be described below in Section 4. The split and join of a chunk requires a *freeze* of all operations on it, which imposes the *freeze state* of a chunk to be declared using *freezeState* field. The freezing mechanism is explained later, in Section 3.

2.3 Memory Management

To avoid some of the ABA problems, lock-free algorithms typically rely on garbage collection or use the hazard pointer mechanism of Michael [12]. To simplify the current presentation, we assume the existence of garbage collection for the nodes. This means that nodes are never reused unless they become unreachable from all threads. An extension of the same scheme to a use of hazard pointers is possible.[4]

2.4 The Basic B$^+$tree Operations

The B$^+$tree interface methods: *SearchInBtree()*, *InsertToBtree()*, and *DeleteFromBtree()* are quite simple. The code of the basic B$^+$tree operations appear in Algorithm 1. An insert, delete, or search operation first finds the leaf with the relevant key range, after which the appropriate chunk operation is run on the leaf's chunk. It either simply succeeds or a more complicated action of a split or a join begins. Some care is needed when the suitable leaf is a new one (an infant), whose insertion into the B$^+$tree is not yet complete. In that case, we must help finish the insertion of the new node before continuing to perform the operation on it. Further explanations on the freezing of a node, on the infant state, etc. appear in Section 3.

Two important methods support the general use of the B$^+$tree. The first one is the *FindLeaf()* method that is used for finding a leaf whose associated range of values contains a given key. The second widely used supporting method is *FindParent()*. When a split or a merge occurs, we may need to find the parent of the current node in order to modify its

[3]Since a data field cannot hold a full pointer, we assume a translation table, or some base pointer to which the 32-bit address is added to create the real memory address. In the first case, this limits the number of nodes to 2^{32} nodes, and in the second case, it limits the entire tree space to 4GB, which is not a harsh constraint.

[4]In the implementation we measured, we implemented hazard pointers inside the chunk and did not reclaim full nodes at all during the execution.

Algorithm 1 Search, Insert, and Delete – High Level Methods.

```
(a) Bool SearchInBtree (key, data) {
1: Node* node = FindLeaf(key);
2: return SearchInChunk(&(node→chunk), key, data);
 }
(b) Bool InsertToBtree (key, data) {
1: Node* node = FindLeaf(key);
2: if (node→freezeState == INFANT)
3:    helpInfant(node);                    // Help infant node
4: return InsertToChunk(&(node→chunk), key, data);
 }
(c) Bool DeleteFromBtree (key, data) {
1: Node* node = FindLeaf(key);
2: if (node→freezeState == INFANT)
3:    helpInfant(node);                    // Help infant node
4: return DeleteInChunk(&(node→chunk), key);
 }
```

pointers.[5] Furthermore, we may need to find an adjacent node as a partner for a merge, when a node gets too sparse. The *FindLeaf()* and *FindParent()* methods are presented in detail in [3].

3. SPLITS AND JOINS WITH FREEZING

Before it is split or joined, a node's chunk must be frozen. The complete details appear in [4]. The freezing is executed by the chunk mechanism when its size limits are violated. This happens obliviously to the containing data structure, in this case, the B$^+$tree. Here we provide an overview on the chunk's freeze required to understand the B$^+$tree algorithm. To freeze a node, i.e., to freeze the chunk in it, all the chunk's entries are marked *frozen* (one by one) by setting a designated bit in each entry. After all the entries are marked frozen, no changes can occur on this node. A thread that discovers that a node needs to be frozen, or that a freeze has already begun, helps finish freezing the node. However, search operations do not need to help in freeze and can progress on the frozen nodes. Since changes may occur before all entries are marked frozen, the final state of the frozen node may not require a split or a join at the end of the freeze. Still a frozen node is never resurrected. After the freeze has been marked and the node can no longer be modified, a decision is made on whether it should be split, or joined with a neighboring node, or just copied into a single new node. If a join is required, then a neighboring node is found by the B$^+$tree. This communication between the chunk and the B$^+$tree is implemented using a predetermined method *FindJoinSlave()* that the tree supplies and the chunk mechanism uses. Then the neighboring chunk is frozen too. To recover from the node freeze, one or two nodes are allocated, and the live entries in the frozen node (or nodes) are copied into the new node (or nodes). Thereafter, a B$^+$tree method *CallForUpdate()* is called to let the tree replace the frozen nodes with the new ones. We focus in what follows on issues specific to the B$^+$tree, i.e., finding a neighbor, replacing the frozen nodes with the new ones in the B$^+$tree, and maybe rolling up more splits or joins.

Each tree node has a *freezeState* field, holding one of eight possible freeze states. Three bits are used to store the state. The freeze state is also a communication link between the B$^+$tree and the chunk mechanism, and so it can be read and updated both by the B$^+$tree and by the chunk. When a new node is created to replace a frozen node, and until it is properly inserted into the B$^+$tree, its freeze state is marked as INFANT. No insertions or deletions are allowed on an infant node until the node's freeze state becomes NORMAL. Any thread that attempts an operation on such a node must first help move this node from the INFANT to the NORMAL state. A node that is properly inserted into the B$^+$tree and can be used with no restrictions has a NORMAL freeze state. When an insert or a delete operation violates the maximum or minimum number of entries, a freeze of that node is initiated and its freeze state becomes FREEZE. After the freezing process stabilizes and the node can no longer be modified, a decision is reached about which action should be taken with this node. This decision is then marked in its freeze state as explained below.

When neither split nor join is required (because concurrent modifications have resulted in a legitimate number of entries), the freeze state of the node becomes COPY, and the node is simply copied into a newly allocated node. By the end of the copy, the parent's pointer into the old node is replaced (using the chunk's replace operation) with the pointer to the new node, and the new node becomes NORMAL. When a split is required, the node's frozen state changes to SPLIT and all its live entries are copied into two new INFANT nodes. These nodes are then inserted into the tree in place of the frozen node, after which they can become NORMAL. A join is more complicated since a neighbor must be found and *enslaved* for the purpose of the join. Since only three bits are required to store the freeze state, we can use the freeze state to also store a pointer to a join buddy and modify the state and the pointer together atomically.[6] The join process starts by looking for a neighbor that can be enslaved for the join and then the freeze state of the join initiator is changed into REQUEST_SLAVE together with a pointer to a potential join buddy in the *joinBuddy* word. Thus, the freeze state is actually modified into a pair ⟨REQUEST_SLAVE, *slave*⟩. At the enslaved node, its state is then modified from NORMAL into the pair ⟨SLAVE_FREEZE, *master*⟩, where *master* is a pointer to the node that initiated the join. (Upon failure, we try to resolve the contention and try again.) When the connection between the join initiator (the master) and the join slave is finalized, the freeze state of the master is modified into ⟨JOIN, *slave*⟩, where *slave* points to the determined join buddy node. The node that is typically chosen for a join is the immediate left sibling of the current node, except for the leftmost node, which chooses its immediate right sibling for the join. A special boundary condition appears when the two leftmost children of a node try to enslave each other. In order to break the symmetry in this case, we take special care to identify this situation and then choose the leftmost sibling among the two to be the slave. Figure 2 presents the state transition diagram for the *freezeState* field.

[5]Note that attempting to maintain a list of parent pointers is difficult for a B-Tree as each parent has a large number of children nodes that need to be simultaneously updated when the parent is modified via a split or a merge

[6]An 8-alignment of a node can be assumed in modern systems and the three redundant least-significant bits can hold the freeze state

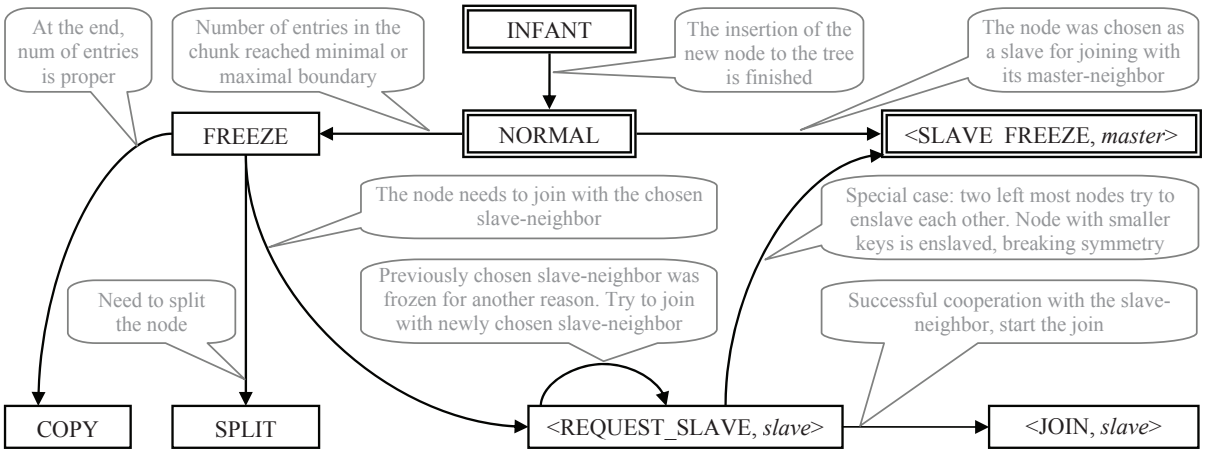

Figure 2: The state transitions of the freeze state of a node. The initial states are presented in the boxes with the double border.

4. BALANCING THE B$^+$tree

The basic methods for the tree operations have been discussed in Section 2.4. We now give a description of how to rebalance the tree following a split or a join of a node, and discuss the interface between the chunk mechanism and the tree operations. As explained above, upon violation of the node size constraints, the chunk mechanism freezes the node to prevent it from being modified, and then determines the required rebalancing action (split, join, or copy). In case of a join, the chunk mechanism invokes the B$^+$tree method *FindJoinSlave()*, which finds a join buddy. Later, the chunk mechanism creates new node(s) and copies the relevant information into them. When this part is completed, the chunk calls the B$^+$tree method *CallForUpdate()*. This method lets the B$^+$tree algorithm replace the frozen node (or nodes) with the newly created node(s) in the tree. The *CallForUpdate()* method actually redirects the calls according to whether a split, a copy, or a join occurred. We next examine each of these cases.

But before diving into the details, note that in general, upon creation of a node due to a split, a join, or a copy, the new node's freeze state is initiated to INFANT, its root flag is initiated to FALSE, its height value is copied from the original node's height value, and its counter is initiated to the exact number of entries copied into it. Also, the *creator* field of a new node is initiated to contain a pointer to the old node. the one that initiated the split, join, or copy operation.

4.1 Node Split

After the chunk mechanism executes a split, the original full node N is frozen, and N's *new* field points to the new node N_1 holding the lower half of the keys from the old node N. The field N_1.*nextNew* points to the second new node N_2 holding the higher half of the keys from the old node N. The two new nodes' freeze states are initiated to INFANT so that no updates can occur on these nodes until they are inserted into the tree. When the chunk split is completed, the *CallForUpdate()* method is invoked and in this case it proceeds by invoking the *InsertSplitNodes()* method, the code of which appear in Algorithm 2.

The *InsertSplitNodes()* method receives a pointer to N (n in the code), which is the frozen node whose split needs to be completed), and the *sepKey* parameter. The *sepKey* parameter holds the middle key that "separates" the two new nodes. The *sepKey* key and all lower keys have been placed in N_1 (*n1* in the code), and all keys higher than *sepKey* have been placed in N_2 (*n2* in the code).

Replacing the split node N starts by searching for its parent P in the tree, using *sepKey* for navigation in the tree. If the parent cannot be found, then the input node is no longer in the tree. This happens if the new node N_1 was properly inserted by some other thread, and the node N was disconnected in the process. In this case, the splitting process proceeds and attempts to insert N_2. Otherwise, and having found the parent, we modify it to point to the new node N_1. This is done by inserting a new entry to P (Line 6). The new entry contains the maximal key from N_1 as key and the pointer to N_1 as data. If the insert fails, it means that someone else has inserted this entry to the chunk and it is fine to continue. Therefore, we do not check if the insert succeeded. Note the possibility that the parent's chunk insert will create a split in the parent, which will recursively cause a split and roll it up the tree.

After the first new node is in place, we replace the pointer in the parent node, which points to the frozen node N, with the pointer to the second new node N_2 (Line 9). Again, this can only fail if another thread has done this earlier. The *ReplaceInChunk()* method finds the entry with key and data as in its third argument and replaces it with key and data as in its last argument. (The *combine()* method syntactically combines the key and the data values into a single word.) In order to invoke *ReplaceInChunk()* on the correct parent, we search for the parent (in the tree) of the split node, using the maximal key of that node for navigating in the tree. The second parent search may yield a different parent if the original parent was concurrently split or joined. After making the parent point to the two new nodes, it remains to set their state to NORMAL and return. The splitting process is complete.

If the original node N was determined to be the root, then a new root R with two new children N_1 and N_2 is created. Next, the B$^+$tree's root pointer is swapped from pointing

Algorithm 2 The split of a non-root node

```
void InsertSplitNodes (Node* n, sepKey) {                    // sepKey is the highest key in the low-values new node
 1: Entry* nodeEnt;                                          // Pointer to the parent's entry pointing to the node about to be split
 2: Node* n1 = n→new;                                        // Pointer to the new node that holds the lower keys
 3: Node* n2 = n→new→nextNew;                                // Pointer to the new node that holds the higher keys
 4: maxKey = getMaxKey(n);                                   // Get maximal key on the given frozen node
 5: if ((parent = FindParent(sepKey, n, &nodeEnt, NULL)) != NULL) {
 6:    InsertToChunk(parent→chunk, sepKey, n1);              // Can only fail if someone else completes it before we do
 7: }
 8: if ((parent = FindParent(maxKey, n, &nodeEnt, NULL)) != NULL) {
 9:    ReplaceInChunk(parent→chunk, nodeEnt→key,             // Can only fail if someone else completes it before we do
10:                   combine(nodeEnt→key, n), combine(nodeEnt→key, n2));
11: }
12: CAS(&(n1→⟨freezeState, joinBuddy⟩), ⟨INFANT, NULL⟩, ⟨NORMAL, NULL⟩);   // Update the states of the new nodes
13: CAS(&(n2→⟨freezeState, joinBuddy⟩), ⟨INFANT, NULL⟩, ⟨NORMAL, NULL⟩);   // from INFANT to NORMAL
14: return;
}
```

Algorithm 3 The code of finding a node partner for a join in the lock-free B$^+$tree.

```
Node* FindJoinSlave(Node* master) {
 1: Node* oldSlave = NULL;
 2: start: anyKey = master→chunk→head→next→key;                          // Obtain an arbitrary master key
 3: if ( (parent = FindParent(anyKey, master, &masterEnt, &slaveEnt)) == NULL) {   // If master is not in the B+tree;
 4:    return master→⟨*, joinBuddy⟩;                                     // thus its slave was found and is written in the joinBuddy
 5: }
 6: slave=slaveEnt→data;                                                 // Slave candidate found in the tree
 7: // Set master's freeze state to ⟨REQUEST_SLAVE, slave⟩; oldSlave is not NULL if the code is repeated
 8: if ( oldSlave == NULL ) expState = ⟨FREEZE, NULL⟩; else expState = ⟨REQUEST_SLAVE, oldSlave⟩;
 9: if ( !CAS(&(master→⟨freezeState, joinBuddy⟩), expState, ⟨REQUEST_SLAVE, slave⟩) ) {
10:    // Master's freeze state can be only REQUEST_SLAVE, JOIN or SLAVE_FREEZE if the roles were swaped
11:    if ( master→⟨freezeState,*⟩ == ⟨JOIN,*⟩ ) return master→⟨*, joinBuddy⟩;
12: }
13: slave = master→⟨*, joinBuddy⟩;                                       // Current slave is the one pointed by joinBuddy
14: // Check that parent is not in a frozen state and help frozen parent if needed
15: if ( (parent→⟨freezeState,*⟩ != ⟨NORMAL,*⟩) && (oldSlave == NULL) ) {
16:    Freeze(parent, 0, 0, master, NONE, &result); oldSlave = slave; goto start;
17: }
18: // Set slave's freeze state from ⟨NORMAL, NULL⟩ to ⟨SLAVE_FREEZE, master⟩
19: if ( !SetSlave(master, slave, anyKey, slave→chunk→head→next→key) ) {oldSlave = slave; goto start;}
20: CAS(&(master→⟨freezeState, joinBuddy⟩),⟨REQUEST_SLAVE, slave⟩,⟨JOIN, slave⟩);   // We got the slave, update the master
21: if (master→⟨freezeState,*⟩ == ⟨JOIN,*⟩) return slave; else return NULL;
}
```

to N to point to R. The details of the root's split code are relegated to [3].

4.2 Nodes Join

Establishing the master-slave relationship: We assume that the join is initiated by a sparse node N, denoted *master*. The chunk mechanism has frozen the node N and it has determined that this node has too few entries. To complete the join, the chunk lets the B$^+$tree find the *slave*. The B$^+$tree establishes a master-slave relationship and later the chunk mechanism joins the entries of both nodes. The B$^+$tree's *FindJoinSlave()* method is responsible for establishing the master-slave relationship and returning the slave for the given master. Its code is presented in Algorithm 3. The establishment of the master-slave relationship is described below.

The *FindJoinSlave()* method starts by calling the *FindParent()* method, which returns a pointer to the master's parent node together with the pointers to the master's and its potential slave's entries. The parent node search fails only if the node N has already been deleted from the tree, in which case a slave has already been determined and can be retrieved from the *joinBuddy* field of N (Line 4). Otherwise, the parent and a potential slave node M were returned by *FindParent()*. The left-side neighbor is returned for all nodes except the left-most node, for which a right-side neighbor is returned. In order to establish the relationship we first change N's freeze state from ⟨FREEZE, NULL⟩ to ⟨REQUEST_SLAVE, M⟩. (Recall that the *joinBuddy* field and the freeze state field are located in a single word.) If this is not the first try, the field may hold a previous slave pointer (*oldSlave*) that we could not enslave. In this case, we change the value of N's freeze state from ⟨REQUEST_SLAVE, *oldSlave*⟩ to ⟨REQUEST_SLAVE, M⟩, where M is the new potential slave. The CAS operation in Line 9 may fail if N's freeze state has already been promoted to JOIN or it has become SLAVE_FREEZE due to swapping of master-slave roles as explained below. In these cases N's final slave was already set in the *joinBuddy* field of N. The CAS operation in Line 9 may also fail if another slave was already chosen due to delay of this CAS operation. In this case, we just use that slave (Line 13).

Algorithm 4 Setting the slave's freeze state for a join in the lock-free B$^+$tree.

```
Bool SetSlave(Node* master, Node* slave, masterKey, slaveKey) {
 1: // Set slave's freeze state from ⟨NORMAL, NULL⟩ to ⟨SLAVE_FREEZE, master⟩
 2: while (!CAS(&(slave→⟨freezeState,joinBuddy⟩),⟨NORMAL,NULL⟩,⟨SLAVE_FREEZE,master⟩)){
 3:    // Help slave, different helps for frozen slave and infant slave
 4:    if (slave→⟨freezeState, *⟩ == ⟨INFANT, *⟩) { helpInfant(slave); return FALSE; }
 5:    elseif (slave→⟨freezeState, *⟩ == ⟨SLAVE_FREEZE,master⟩) break;          // Completed by someone else
 6:    else {                                  // The slave is under some kind of freeze, help and look for new slave
 7:       // Check for a special case: two leftmost nodes try to enslave each other, break the symmetry
 8:       if ( slave→⟨freezeState, *⟩ == ⟨REQUEST_SLAVE, master⟩ ) {
 9:          if (masterKey < slaveKey) {          // Executing master node is left sibling and should become a slave
10:             if ( (master→⟨freezeState,joinBuddy⟩ == ⟨SLAVE_FREEZE,slave⟩) || CAS(&(master→⟨freezeState,joinBuddy⟩),
11:                      ⟨REQUEST_SLAVE,slave⟩, ⟨SLAVE_FREEZE,slave⟩) ) return TRUE; else return FALSE;
12:          else                               // Current master node is right sibling and the other node should become a slave
13:             if ( (slave→⟨freezeState,joinBuddy⟩ == ⟨SLAVE_FREEZE,master⟩) || CAS( &(slave→⟨freezeState,joinBuddy⟩),
14:                      ⟨REQUEST_SLAVE,master⟩, ⟨SLAVE_FREEZE,master⟩) ) return TRUE; else return FALSE;
15:       } // end case of two leftmost nodes trying to enslave each other
16:       Freeze(slave, 0, 0, master, ENSLAVE, &result);          // Help an unrelated freeze activity
17:       return FALSE;
18:    } // end of investigating the enslaving failure
19: } // end of while
20: MarkChunkFrozen(slave→chunk); StabilizeChunk(slave→chunk);          // Slave enslaved successfully. Freeze the slave
21: return TRUE;
}
```

Lines 15 and 16 are important for keeping the master and the slave descendants of the same parent This is further discussed in Subsection 4.3.

After finding a potential slave, we attempt to set its freeze state to ⟨SLAVE_FREEZE, N⟩ and freeze it. This is done in the *SetSlave()* method presented in Algorithm 4 and explained in the next paragraph. If this action is not successful, the *FindJoinSlave()* method is restarted from scratch. After succeeding in setting the slave's freeze state, we change the master's state from ⟨REQUEST_SLAVE, M⟩, to ⟨JOIN, M⟩ to enable the actual join attempts.

The *SetSlave()* method attempts to CAS the freeze state of the slave M from ⟨NORMAL, NULL⟩ to ⟨SLAVE_FREEZE, N⟩. If the CAS of the freeze state in the slave is successful, we may proceed with the join. But M's freeze state isn't necessarily NORMAL: if it is not, then M is either still an infant or it is already frozen for some other reason. In the first case, *SetSlave()* helps M to become NORMAL and retries to set M's freeze state. In the second case, it helps to complete M's freeze. After finishing the freeze on M, M is frozen and is not suitable to serve as a slave. Therefore, failure is returned by *SetSlave()* and another slave must be found. A special case occurs when the potential slave M has a master freeze-state as well and is concurrently attempting to enslave N for a join. This case can only happen with the two leftmost nodes and, if special care is not taken, an infinite run may result, in which each of the two nodes repeatedly tries to enslave the other. In order to break the symmetry, we check explicitly for this case, and let the leftmost node among the two give up and become the slave, with a SLAVE_FREEZE state and a pointer to its master (which was originally meant to be enslaved for it). The *FindJoinSlave()* checks for this case in its last line. If it is successful in turning the freeze state of the master into JOIN, then all is well. Otherwise, and given that *SetSlave()* completed successfully, then it must be the case that the master has become a slave. In this case, no slave is returned, and the returned NULL value tells the calling method (in the chunk mechanism) to treat the master as the slave.

Finally, the *SetSlave()* completes by freezing the slave in Line 20, so that the join can continue. Two methods of the chunking mechanism are used. The method *MarkChunkFrozen()* marks all entries of a node frozen by setting a designated bit in each entry. After the entries are marked frozen, the *StabilizeChunk()* method ensures that no changes occur on this node. At this point the slave has been enslaved and frozen.

Merge: If the number of entries on the master and the slave is less than d, the chunk mechanism creates a new single chunk to replace the master and the slave. It then invokes the *CallForUpdate()* method to insert the new node into the tree. We denote this operation as *merge*. In this situation, the *CallForUpdate()* method invokes *InsertMergeNode()* whose code is presented in Algorithm 5. At this point, a master-slave relationship has already been established, both M and N have been frozen, and a new node N_1 has been created with the keys of both M and N merged.

The *InsertMergeNode()* method's input parameter is a pointer to the master, this master's slave can be found in the *joinBuddy* field on the master. The *InsertMergeNode()* method starts by checking which of the original nodes (master and slave) has higher keys. Denote this node by *highNode*. Note that the master and the slave are frozen and thus immutable. Next, *FindParent()* is invoked on *highNode*. If the parent is not found, then *highNode* has already been deleted and we can proceed with handling the old node with the lower keys, *lowNode*. Otherwise, we modify the parent's reference to *highNode*, to point to the new node (Line 9). Next, we handle the pointer to *lowNode* at the parent by attempting to delete it. Finally, we turn the new node's freeze status from *infant* to *normal*.

Special care is given to the root. We must avoid a root with a single descendant, which can occur when the two descendants of a root are merged. In this case, we make the merged node become the new root. If merged node parent is found to be root, the *MergeRoot()* method is invoked from *InsertMergeNode()* instead of deleting the pointer to *lowNode* at the parent. This is so, because deleting an entry

Algorithm 5 The merge of two old nodes to one new node

```
void InsertMergeNode (Node* master) {
 1: Node* new = master→new;                                                    // Pointer to the new node.
 2: Node* slave = master→⟨*, joinBuddy⟩;
 3: maxMasterKey = getMaxKey(master); maxSlaveKey = getMaxKey(slave);          // Both nodes are frozen
 4: if ( maxSlaveKey < maxMasterKey ) {                                        // Find low and high keys among master and slave
 5:    highKey = maxMasterKey; highNode = master; lowKey = maxSlaveKey; lowNode = slave;
 6: } else { highKey = maxSlaveKey; highNode = slave; lowKey = maxMasterKey; lowNode = master; }
 7: if ((parent = FindParent(highKey, highNode, &highEnt, NULL)) != NULL) {
 8:    highEntKey = highEnt→key;                                               // Change the highest key entry to point on new node
 9:    ReplaceInChunk(parent→chunk, highEntKey,                                // If replacing fails, the parent chunk was updated by a helper
10:                   combine(highEntKey, highNode), combine(highEntKey, new)); // continue anyway
11: } // If high node cannot be found continue to the low
12: if ((parent = FindParent(lowKey, lowNode, &lowEnt, NULL)) != NULL) {
13:    if (parent→root) MergeRoot(parent, new, lowNode, lowEnt→key);
14:    else DeleteInChunk(&(parent→chunk), lowEnt→key, lowNode);              // lowNode is the expected data
15: } // If also low node can no longer be found on the tree, then the merge was completed (by someone else).
16: CAS(&(new→⟨freezeState, joinBuddy⟩), ⟨INFANT, NULL⟩, ⟨NORMAL, NULL⟩); // Update the new node state from INFANT to NORMAL
17: return;
}
```

from the root may lead to having a single root descendant. (The *MergeRoot()* method is presented in [3].)

Borrow: If the keys of two join nodes cannot fit a single node, they are copied into two new nodes. This operation is called *borrow*. Due to lack of space, the details of the borrow operation are omitted here and can be found in the full version of this paper [3]. In a nutshell, the borrow case has four nodes involved: the master N, the slave M, the new node with the lower keys N_1 and the new node with the higher keys N_2. As in merge case, we start by finding the high and low keys' nodes, N_{high} and N_{low}, among the master and the slave. We then take the following steps: (1) Insert a reference to N_1 to the parent node (with the maximal key on the N_1 as the key); (2) Change the parent entry pointing to N_{high} to point to N_2; (3) Delete the parent entry pointing to N_{low}.

4.3 Two Invariants

Let us mention a couple of invariants that our algorithm maintains. These invariants may give some intuition on how the algorithm works and why it is correct.

Keys duplication. During the rebalancing operations described above, it sometimes happens that (for a short while) two duplicates of a key may become reachable from the root. However, at no point in the execution will a key be absent. For example, after the first new node is inserted to the parent as part of the split, there are keys that reside simultaneously in two different nodes: all keys in this first new node are also still available in the old split node, which is still in the tree. Similarly, as part of the merge, when an old frozen node with higher keys is replaced with the new node, there are keys that appear twice: all keys in the old frozen node with lower keys now also appear in the new node. Recall that a search in the B$^+$tree is allowed to navigate the tree and return the result, based on the data found on the frozen node.

This does not foil searches in the tree. When a key has duplicates available in two different reachable tree nodes the two nodes are immutable. One of these nodes must be frozen and the other must be an infant. Therefore, old searches may safely access keys in the old frozen node(s), and new searches can access the new infant node(s). None of these nodes can be modified until the rebalance operation terminates.

We should also note that the tree doesn't grow too big because of duplication. Another invariant is that there can only be two copies of a key in the tree. Thus, even though we may increase the size of the tree during the rebalancing operations, the increase will be at most by a factor of two. The factor-two increase is theoretical. In practice, the increase in the tree size is negligible.

Master-slave bond. We take special care to guarantee that the master and the slave keep the same parent up to the end of their join. Initially, the master and the slave are siblings and children of the same node P. However, the parent node P may then be split and the master and slave may then have different parent nodes. This may subsequently lead to an inconsistent tree in which a key does not represent the highest key in its subtree. Therefore, we enforce an invariant that the master and slave nodes must remain on the same parent. Namely, we do not allow the parent entries that point to a master and to its slave to be separated into different nodes due to a parent's split or borrow, until new nodes replace the frozen master and slave. Ensuring this variant is taken care of both during the parent split as well as during the children join. On the split side, we check whether the break point between the keys has two descendants that form a master and a slave. In case they do, we enforce placement of both on the same new node. However, the descendants may get into a master-slave relationship only after we make this check at the parent node. Therefore, on the merge side, i.e., the descendants' side, after declaring the intent of a master to enslave its neighbor (setting the master's state to REQUEST_SLAVE), we check that the master's parent is not in a frozen state. If it is, the descendant master helps the parent recover before continuing the descendants' join (Lines 15, 16). This ensures that the parent split (and borrow, in a similar way) does not occur obliviously and concurrently with its descendants' join. More about correctness and progress guarantees can be found in [3].

4.4 Extensions to the Chunk Mechanism

The chunk interface requires some minor modifications over [4] to properly serve the B$^+$tree construction in this

paper. Probably the most crucial modification arises from the need to deal with an ABA problem that arises during insertions and deletions of entries to the chunk of an internal node in the tree. The concern is that an insert or a delete may succeed twice due to a helper thread that remains idle for a while. Consider, for example, a merge and a subsequent delete of an entry at the parent node. Suppose that one thread executes the delete, but a second thread attempts this delete later, after the same key (with a different descendant) has been entered to the parent again. Thus, a delete should only succeed when the entry still points to the frozen node. As for inserts, we need to avoid reentering a pointer to a child node that has actually been frozen and deleted while the updating thread was stalled. To solve such problems, we add versioning to the *nextEntry* word in the chunk's linked-list. This eliminates the ABA problem, as a delayed CAS will fail and make us recheck the node that we attempt to insert or delete and discover that it has already been frozen. All extensions to the chunk mechanism are described in [3].

5. IMPLEMENTATION AND RESULTS

We have implemented the lock-free B$^+$tree presented in this paper as well as the lock-based B$^+$tree of [15] in the C programming language. The lock-free design in this paper can be optimized in many ways. However, we have implemented it as is with no further optimizations. The operations of the lock-based B$^+$tree progress in a top-down direction. During the descent through the tree, *lock-coupling* [1] is employed, i.e., a child is locked before its parent is unlocked. Exclusive locks on the nodes are used for insert and delete operations, and shared locks are used for search operations. Deadlock-freedom is guaranteed by a proactive approach to rebalancing that splits full nodes or joins sparse ones, while going down the path to the leaves.

We ran the experiments on the SUN FIRE machine with an UltraSPARC T1 8-core processor, each core running 4 hyper-threads, running Solaris 10. Overall, the eight cores, with quad hyper-threading simulates the concurrent execution of 32 threads. In both implementations the size of a B$^+$tree node was set to the machine's virtual page size, i.e., 8KB. In each test we start with a tree with N random keys in the range $[0, 2^{18}]$ already inserted to it, and during the test, we apply N operations on it. If the test runs T threads, then each executes N/T operations. The parameter N was varied among 10^4, 10^5 and 10^6. The operations consisted of insertions, deletions and searches in parallel, out of which 20% were insertions, 20% were deletions, and the remaining 60% were searches. All the threads start simultaneously at the beginning and we measure the time it takes to complete all operations by all threads.

The right graph of Figure 3 depicts the ratio between the time it took to complete the runs on the lock-free implementation as compared to the lock-based implementation. A result higher than 1 means that the lock-free implementation is slower. Clearly, the lock-free implementation outperforms the lock-based implementation when contention is not low. Note that contention increases as the tree gets smaller and as the number of threads increases. Also, the results show that the average cost of an operation increases as the tree gets larger, because rebalancing may ascend to higher levels. Such costs are heavier for the lock-free tree, but this overhead is offset by lock-freedom efficiency when contention kicks in. The right graph of Figure 3 depicts the speedup, which clearly shows that the lock-free algorithm is more scalable.

The weaker performance of the lock-free tree for low contention can be ameliorated by simple optimizations. For example, during the split, each thread helping the split copies the entries from the old node to a newly created private node and only one of these new nodes eventually replaces the old node and joins the tree. While threads can cooperate to perform copying, we decided to avoid it in this version because it complicates the design.

6. LINEARIZATION POINTS

When designing a concurrent data structure, it is important to spell out the linearization points for the different operations. This is done in this section. The B$^+$tree methods all have a similar pattern of operation: they traverse the B$^+$tree to find the relevant leaf node, and then call the appropriate chunking methods on the leaf's chunk. Thus the linearization points of the B$^+$tree are typically based on the linearization points defined for the chunk in [4].

Search linearization point: The linearization point of the search operation is exactly the linearization point of the leaf's chunk search, as in [4]. In particular, if the leaf is not frozen, then the linearization point follows that of the underlying linked-list in the leaf's chunk, and if the leaf is frozen then the linearization point is set to be the point in which the chunk became frozen. As the freezing mechanism is not instantaneous, we need to define a point in the freezing process more accurately for the linearization point. We follow [4] and set the linearization point to be the point in the freeze process by which all the frozen bits have been set and also the internal list of the freezing node has been stabilized. Define this point as the *freezing point*. The freezing process of a chunk is explained more thoroughly in [4]. Formally, consider the linearization point of the search of the linked-list that is inside the chunk of the leaf (as defined by Harris [9]). If the chunk's linked-list search linearization point occurs before the freezing point, then that is also the linearization point of the overall tree search. If the chunk's linked-list linearization point happens after the freezing point, then we define the overall tree search linearization point to be the later point between the freezing point and the point in which the search started. The latter maximum makes sure that the linearization point happens during the execution of the search.

Justifying this choice for non-frozen node is straightforward. As for frozen nodes, we note that the frozen node may be replaced with a new node during the search execution and various actions may be applied on the new node. But at the freezing point, we know that the values of the frozen node exist only in the frozen node and are properly represented by the view of the frozen node.

The delicate case is when the search starts after the freezing point and still gets to the frozen leaf and completes the search there. In this case, since the search ends up in this leaf, we know that a new node that replaces this leaf (following the end of the freeze) has not yet been modified while the search traversed the tree, because the rebalancing operation has not yet terminated at that point. Therefore the new node has definitely not been modified when the search started, and the frozen values represent correctly the state of the tree at that point in time.

Insert and delete linearization points: Unlike the

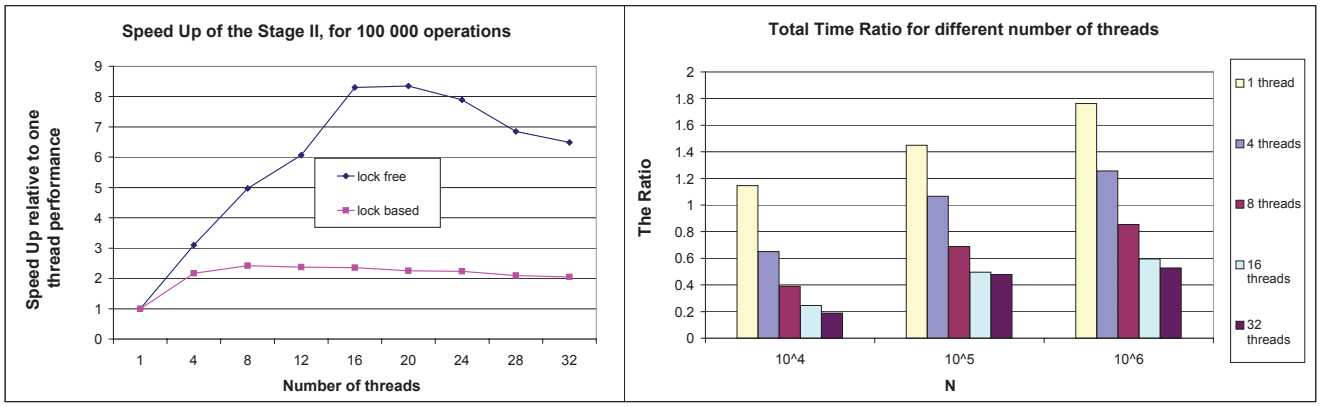

Figure 3: The empirical results.

analysis of the search operation, frozen nodes are not hazardous for the insert's and delete's initial tree traversing. If an insert or delete arrive at a frozen leaf, than the *InsertToChunk()* or the *DeleteInChunk()* methods will redirect the operation (after helping the frozen node) to a non-frozen leaf node. Intuitively, the insert operation is assumed to be finished when a leaf including the new key is reachable from the root via data pointers. Similarly, the delete operation is assumed to be finished when a leaf excluding an old key is reachable from the root via data pointers. In a worst-case, this may require more than just handling a freeze.

There are three cases possible here. First, if the insert or delete operation doesn't cause a rebalancing activity (split, merge, borrow, or copy), than the linearization point is simply determined to be the leaf's chunk linearization point. Second, if a rebalancing (by freezing) occurs and if the thread performing the insert or delete operation has its operation executed in the node that replaces the frozen node, then the linearization point of the operation becomes the linearization point of the insert operation of the new node to the parent of the frozen node (replacing the frozen node with the new one). Note that this definition may be recursive if the parent requires rebalancing for the insertion. The third case is when the result of this operation is not reflected in the node that replaces the frozen one. In this case, we again define the linearization point recursively, setting it to be the linearization point of the re-attempted operation on the new node that replaced the frozen one.

7. CONCLUSIONS

We presented a lock-free dynamic B^+tree, which builds on CAS synchronization. The construction is composed of a chunk mechanism that provides the low-level node implementation, including splitting and joining a node, and then a higher level mechanism which handles the operations at the tree level. The two mechanisms and their interface are lock-free. To the best of our knowledge, this is the first design of a lock-free balanced search tree for a general platform. Results indicate better handling of contention and higher scalability when compared to the lock-based version of the B^+tree. We have also proven the correctness (with respect to linearizability) of the algorithm and its lock-freedom property.

8. REFERENCES

[1] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 9, 1977.

[2] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent cache-oblivious b-tree. *SPAA*, 2005.

[3] A. Braginsky and E. Petrank. Lock-free B+tree (full version). http://www.cs.technion.ac.il/ erez/Papers/lfbtree-full.pdf.

[4] A. Braginsky and E. Petrank. Lock-free linked lists with improved locality. *ICDCN*, 2011.

[5] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefetching b+-trees: Optimizing both cache and disk performance. *SIGMOD*, 2002.

[6] D. Comer. The ubiquitous b-tree. *ACM Computing Surverys*, 11(2), 1979.

[7] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search tree. *PODC*, 2010.

[8] K. Fraser. Practical lock-freedom, 2004. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory.

[9] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. *DISC*, 2001.

[10] M. Herlihy. Wait-free synchronization. *TOPLAS*, 13(1):124–149, 1991.

[11] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990.

[12] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *TPDS*, 15(6):491–504, 2004.

[13] E. Petrank, M. Musuvathi, and B. Steensgaard. Progress guarantee for parallel programs via bounded lock-freedom. *PLDI*, pages 144–154, 2009.

[14] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. *VLDB*, 1999.

[15] O. Rodeh. B-trees, shadowing, and clones. *ACM Transactions on Storage Journal*, 2008.