Lock-Free Multiway Search Trees

Michael Spiegel Paul F. Reynolds, Jr.
University of Virginia, Charlottesville, VA 22903
mspiegel@cs.virginia.edu reynolds@cs.virginia.edu

***Abstract*—We propose a lock-free multiway search tree algorithm for concurrent applications with large working set sizes. Our algorithm is a variation of the randomized skip tree. We relax the ordering constraints among the nodes in the original skip tree definition. Optimal paths through the tree are temporarily violated by mutation operations, and eventually restored using online node compaction. Experimental evidence shows that our lock-free skip tree outperforms a highly tuned concurrent skip list under workloads of various proportions of operations and working set sizes. The max throughput of our algorithm is on average 41% higher than the throughput of the skip list, and 129% higher on the workload of the largest working set size and read-dominated operations.**

I. Introduction

The term 'memory wall' was coined over fifteen years ago to communicate the disparity between microprocessor speeds and memory latency and bandwidth in sequential processors [1]. Multicore architectures have increased the demand for low latency, high bandwidth memory access. A report on the landscape of parallel computing research identified the memory wall as a major obstacle to good performance for almost half of all classes of computational problems in the report [2]. A study of supercomputing applications at Sandia National Laboratories found that doubling memory latency leads to a 11% drop in the performance of floating-point applications and a 32% drop in the performance of integer applications [3].

The lock-free skip list implements a concurrent ordered set abstract data type. The skip list is scalable under thread contention. However, the design exhibits poor spatial locality of reference. A traversal that accesses N elements in the list also access at least N nodes. A multiway search tree would improve spatial locality by storing several elements in a single node.

In this paper we present a lock-free skip tree algorithm. Our algorithm implements a lock-free randomized multiway search tree. The lock-free skip tree has relaxed structural properties that allow atomic operations to modify the tree without invalidating the consistency of the data structure. Our skip-tree

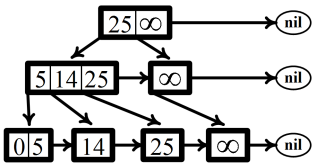


Figure 1: A skip tree with three levels. All nodes in the tree are optimal (as defined in Section III-D).

definition does not require that neighboring elements in the tree's interior serve as partitions on the tree. It maintains consistency by defining a *reachability* relation from the root of the tree to any potential element stored in the tree for all possible states of the tree (Section III-B).

Our specific contributions:

- We describe a redesign of the skip tree in order to support lock-free operations. Our design has two primary differences from the original skip tree definition. First, the introduction of 'link' pointers allows nodes to split independently of their parent nodes (Section III-C). Second, we relax the requirement that routing nodes behave as partitions on the tree (Section III-D).
- We describe a practical lock-free algorithm for the concurrent skip tree. The algorithm supports lock-free `add` and `remove` operations and wait-free `contains` operations. The algorithm is shown to be linearizable (Section IV).
- We show that our lock-free implementation outperforms a highly-tuned skip list, a relaxed balance AVL tree, and a B^{link}-tree across different thread counts, operation mixes, and machine architectures when the working set size cannot be contained in cache (Section V).

II. Background

Many data structures have been designed with concurrent, lock-free, cache conscious, or randomized properties. Our skip tree algorithm shares some structural properties with these data structures. Each of the related data structures shares one, two,

or three, but not all four properties of the lock-free skip tree design.

The skip list is a linked list data structure with express lanes of randomized heights that provides an expected $O(\log n)$ cost for basic operations [4]. Membership of the skip list and our algorithm is determined only by the bottommost linked list level. As the upper levels of the data structures provide hints to speed up searching, they can be modified by lazy updates to produce a concurrent ordered data structure [5, 6].

Skip trees are randomized multiway search trees that share properties with B-trees and skip lists. The construction of the deterministic skip list [7] is based on transforming the skip list to a multiway search tree. Messeguer [8] defines the skip tree as a randomized tree in which element membership is determined at the bottommost level of the tree. All paths in the tree are of identical length, and nodes with zero elements are used to preserve the invariant. Neighboring elements in the tree's interior serve as minimum and maximum bounds for the subtree of their shared child reference.

The ideal-cache model is an abstract memory model that consists of a two-level memory hierarchy. Cache conscious algorithms are optimized for the ideal-cache model and contains parameters that can be tuned to the cache size (C) and line length (L). The B-tree and the skip tree are cache conscious algorithms. Cache oblivious algorithms are optimal when the cache size and line length are unknown parameters [9]. A cache oblivious B-tree design proposed by Bender et al. [10] uses a random distribution for the height of elements. It supports `contains` and `add` operations, but no `remove` operations. The cache oblivious design performs at best $(\log_2 e)(\log_L N)$ memory transfers per search operation under the ideal-cache model [11].

Lock-free data structures implement concurrent objects without the use of mutual exclusion. Lock-based resource protection schemes can suffer from performance concerns such as priority inversion, lock convoying, and lock contention and liveness concerns such as starvation and deadlock. The earliest designs for lock-free linked list algorithms are credited to Valois [12]. The design of Michael [13], based on earlier work by Harris [14], forms the basis for the lock-free skip list algorithm in the `java.util.concurrent` library and the lock-free linked list levels of our skip tree design. The hallmark of the Michael-Harris algorithm is the marking of 'link' references of deleted nodes to avoid conflicts with concurrent insertions.

Concurrent balanced binary search tree algorithms can exhibit poor tolerance of thread contention due to the balancing requirements of the algorithms. Bronson et al. [15] have developed a concurrent relaxed balance AVL tree algorithm. The balancing requirement is violated by mutating operations and then eventually restored by separate balancing requirements. An extension of the optimistic concurrent AVL tree algorithm uses copy-on-write to provide support for atomic clone and snapshot isolation during iteration.

Concurrent B-tree algorithms have been designed to support simultaneous transactions in a database management system. The B^{link}-tree was developed by Lehman and Yao [16] and refined by Sagiv [17] as a concurrent B-tree algorithm that requires only a single writer lock at any time for `add` and `remove` operations and no locks for `contains` operations.

III. Our Algorithm

The lock-free skip tree implements a linearizable ordered set data structure over some domain T. Three operations are supported: `contains`, `add`, and `remove`. The lock-free skip tree consists of several linked lists stacked on top of each other. Each node contains some number of elements, k for $k \geq 0$, that varies over time and is independent of the other nodes. The lowest level of the tree consists of elements of height 0, and is defined to be leaf level of the tree. Non-leaf nodes are referred to as routing nodes. A routing node contains k child references to nodes that are one level below in the tree. A leaf node or a routing node with zero elements is referred to as an empty node. It is forbidden to insert new elements into an empty node.

An element is assigned a random height, h, upon insertion into the tree. The random height is a non-negative integer. An element is considered to be a member of the skip tree if-and-only-if the element is a member of a leaf node in the tree. If the random height h is greater than zero, then a copy of the element is inserted at each level from the leaf level up to and inclusive of level h.

The root of the tree is defined to be the first node in the tree at the current highest level. An empty skip tree consists of one leaf node containing the value $+\infty$. Figure 2 shows examples of lock-free skip trees. To the left of the first node at each level is a number indicating the height of that level.

Any pair of adjacent elements share exactly one child reference. For example in Figure 2 the child reference of elements 3 and $+\infty$ at level 1 is the node containing $\{6, +\infty\}$ at level 0. If A is the last element of a node and B is the first element of the successor node at the same level, then the shared child reference of A and B is the first reference of the

(a) insertion into an empty tree

(b) elements $\{1, 2, 3\}$ are deleted then $\{2, 3\}$ are reinserted.

Figure 2: A sequence of `add` and `remove` operations.

successor node. The first element of each level in the tree is assumed to be $-\infty$, although this value is not explicitly stored in the tree.

In order to explain the structural properties of a lock-free skip tree it will be useful to define the *tail set* of a node n at level i. The tail set of n is the set that contains n and all nodes subsequent to n at level i. The tail set of a node n is written as $tail(n)$.

Definition 1. A lock-free *skip tree* consists of a set of nodes. Each node contains a sorted list of elements, a reference to the next node in the linked list, and possibly a list of child references. The elements stored within the nodes are members of some domain set `T`. The lock-free skip tree obeys the following properties:

(D1) Each level contains the element $+\infty$ at the last element of the last node. The element $+\infty$ appears exactly once for each level.

(D2) The leaf level does not contain duplicate elements.

(D3) For each level, given some $v \in$ `T` there exists exactly one pair of adjacent elements A and B such that $A < v \leq B$.

(D4) Given levels i and $i-1$ and some $v \in$ `T`, there exist exactly two pairs of adjacent elements $\{A_i, B_i\}$ and $\{A_{i-1}, B_{i-1}\}$ that satisfy property (D3). If *source* is the child reference between elements A_i and B_i, and *target* is the node that contains elements B_{i-1}, then $target \in tail(source)$.

(D5) For each node, given some $v \in$ `T` such that v is greater than or equal to all the elements of the node, then v will always be greater than or equal to all elements of the node in all possible futures.

Theorem 1. Each level of the skip tree is a sequence of elements in non-decreasing order.

Proof: Assume there exists a pair of consecutive elements A and B occurring in the sequence that is the first pair of elements occurring in decreasing order, i.e. $B < A$. Let C and D be the first pair of consecutive elements in increasing order that is subsequent to A and B. It is known that the pair C and D exists because property (D1) states that each level terminates with $+\infty$ and $+\infty$ appears exactly once for each level. Therefore each level terminates with a pair of adjacent elements in increasing order. As C and D are defined as the first pair of consecutive elements in increasing order that are subsequent to A and B, then it follows that $C \leq B$. The two inequalities may be combined to show that $C < A$.

Select some $v \in$ `T` such that $C < v \leq D$ and $v < A$. By combining the observation that $-\infty < v < A$ and the assumption of the proof that A and B are the first pair of consecutive elements in decreasing order, there must exist a pair of consecutive elements α and β in between elements $-\infty$ and A, inclusively, such that $\alpha < v \leq \beta$. However this violates property (D3) of the skip tree which requires exactly one pair of adjacent elements C and D such that $C < v \leq D$. Therefore the sequence cannot contain a pair of consecutive elements in decreasing order. □

Corollary 1. The leaf level of the skip tree is a sequence of elements in increasing order.

Property (D2) of Definition 1 states the leaf level does not contain duplicate elements. Theorem 1 has shown that all levels contain a sequence of elements in non-decreasing order. Combining these two statements yields the conclusion that the leaf level is a sequence of elements in increasing order.

A. Class and field declarations

A node consists of a single atomic reference to a contents object. The contents object stores an array of items, an array of children, and a reference to the next node in the linked list. A node will not move up or down the tree after its creation. The array of items is always a non-null value. The array of children is null if-and-only-if the enclosing node is a leaf node. A search object is used to keep track of the position of some $v \in$ `T` in relation to a specific node. The search object contains a reference to some node, a reference to a contents object that represents a snapshot of the node, and the index of element v relative to the items stored in the contents object. A head node stores a node reference and an integer representing the height of the node. The skip tree declaration contains a single field, which is an atomic reference to the root of the tree.

Figure 3 shows the class and field declarations. A convention is adopted such that all final fields contain immutable state information. The `AtomicReferenceFieldUpdater` interface is used to enable atomic updates to the contents field of a node or to the root field of a tree. For space reasons we have omitted the declaration of the singleton class

```
1  class Node<T> {
2    volatile Contents<T> contents;
3  }
4  class Contents<T> {
5    final Object[] items;
6    final Node<T>[] children;
7    final Node<T> link;
8  }
9  class Search<T> {
10   final Node<T> node;
11   final Contents<T> contents;
12   final int index;
13 }
14 class HeadNode<T> {
15   final Node<T> node;
16   final int height;
17 }
18 private volatile HeadNode<T> root;
```

Figure 3: Declarations for a tree with key type `T`

that represents the value $+\infty$. The singleton class implements the `compareTo` method such that the method will always return a value of 1. It is assumed that the runtime system performs garbage collection. Among other things, the garbage collector prevents instances of the ABA problem [18] from occurring.

B. Tree traversal: `contains(v)`

Given some $v \in$ `T` it is possible to determine if v is a member of the set by traversing through the skip tree starting at the root. The *reachable* relation is defined to determine whether a node is connected to the root of the tree with some arbitrary number of intermediate reachable nodes. The `contains` operation will determine whether a leaf node containing v is reachable from the root node of the tree.

Definition 2. Given two nodes *source* and *sink*, define the relation *reachable* such that *sink* is reachable from *source* if-and-only-if $sink \in tail(source)$ or *sink* is reachable from *child* where *child* is a child reference of some member of $tail(source)$.

Let h be the current height of the tallest level, and let A_h and B_h be the neighboring elements at level h that satisfy property (D3), ie. $A < v \leq B$. Let n_h be the node that contains element B_h. Node n_h is reachable from the root, because all nodes at level h are in the tail set of the first node of level h. Let A_{h-1} and B_{h-1} be the neighboring elements at level $h-1$ that satisfy property (D3) and let n_{h-1} be the node that contains element B_{h-1}. Property (D4) requires that $n_{h-1} \in tail(n_h)$ which is sufficient to show that n_{h-1} is reachable from n_h. Combining the two reachability arguments it can be shown that n_{h-1} is reachable from the root node. For all i such that $i \leq h$, by induction n_i is reachable from the root node where n_i is the node at the i^{th} level of the tree containing the neighboring element B_i. n_0 is the leaf node containing the neighboring element B_0. Property (D2) states that the leaf level does not contain duplicate elements. Property (D3) states that $(A_0, B_0]$ is the only interval at the leaf level that contains the value v. Therefore v is a member of the set if-and-only-if $v \in n_0$.

```
19 boolean contains(T v) {
20   Node<T> node = root.node;
21   Contents<T> cts = node.contents;
22   int i = search(cts.items, v);
23   while(cts.children != null) {
24     if (-i - 1 == cts.items.length)
25       node = cts.link;
26     else if (i < 0)
27       node = cts.children[-i - 1];
28     else
29       node = cts.children[i];
30     cts = node.cts;
31     i = search(cts.items, v);
32   }     // end traverse routing nodes
33   while(true) {
34     if (-i - 1 == cts.items.length)
35       node = cts.link;
36     else if (i < 0) return false;
37     else return true;
38     cts = node.cts;
39     i = search(cts.items, v);
40   }     // end traverse leaf nodes
41 }
```

Figure 4: Determining whether `v` is in the set

The `contains` method implements the search algorithm described in the previous paragraph. Figure 4 shows the code for the method. Starting at the root, the node η_i is located such that η_i satisfies property (D3) for level i of the tree. The `search` method used on lines 22 and 39 is a shorthand for calling the `Arrays.binarySearch` method. The `search` method returns the index of the search key, if the key is contained in the array. Otherwise the method returns $-$(insertion point) $-$ 1, where the insertion point is the index of the search key if the key were contained in the array. The `contains` method travels through the routing nodes until it eventually reaches a leaf node. When the method can no longer travel to a successor node in the leaf level, then it terminates and returns whether v is a member of the leaf level.

C. Insertion: `add(v)`

When an element is inserted into the dense skip tree it is assigned a random height. The heights of the elements in a skip tree are distributed according to a geometric distribution:

$$\Pr(H = h) = q^h p \text{ where } p + q = 1 \quad (p \text{ and } q \text{ are constants})$$

An element is inserted into the skip tree through successive inserting and splitting of linked list levels of the tree. It is forbidden to insert new elements into an empty node. The node with zero elements acts as the 'marker' of the Michael-Harris algorithm that forbids concurrent updates and signals lazy elimination of the node.

To insert an element at height h, first the element must be inserted at level 0, then the linked list must be split at level 0, then the element must be inserted at level 1, then the linked list must be split at level 1, and this process continues until the element is inserted at level h.

Figure 5 shows the code for the `add` operation. The add operation consists of an initial call to `traverseAndTrack` and continues with alternating calls to `insertList` and `splitList`. A url for the complete code is given in Section VII. `traverseAndTrack` is a specialized version of the tree traversal operation. A path is traversed from the root to the leaves of the tree, while references to the nodes that are to be updated get stored in an array for later use.

The `insertList` method accepts four arguments: an element to insert ($v \in$ `T`), a hint at the correct node for insertion, a target height at which to insert, and a new child reference to insert along with the new element. If the target height is non-zero and the child node argument is **null**, then the previous `splitList` operation was unsuccessful. If the previous splitList operation was unsuccessful or if element v is located at the current level, then the method terminates. If the element v is greater than the largest element in the node, then the method moves forward in the linked list level. Otherwise it constructs a new contents object that contains the new element and child reference. The compare-and-swap operation is attempted using the node, the expected contents object, and the constructed contents object. If the compare-and-swap is unsuccessful then the contents of the node are re-read and the operation is retried.

The `splitList` method accepts two arguments: a partition element ($v \in$ `T`) and a hint at the correct node to split. The split operation will transform a single node into two nodes: a left partition node and a right partition node. The left partition node consists of all elements less than or equal to the partition element, and the right partition node consists of all elements greater than the partition element. The original node is transformed into the left partition node using a compare-and-swap operation. The right partition node is the return value of the method.

```
42  boolean add(T v) {
43    int height = randomLevel();
44    Search[] srchs=new Search[height+1];
45    traverseAndTrack(v, height, srchs);
46    boolean success;
47    success=insertList(v,srchs,null,0);
48    if(!success) return false;
49    for(int i = 0; i < height; i++) {
50      Node<T> right=splitList(v,srchs[i]);
51      insertList(v, srchs, right, i + 1);
52    }
53    return true;
54  }
55
56
57  void traverseAndTrack(T v,
58       int h, Search[] srchs) {
59    HeadNode<T> root = this.root;
60    if (root.height < h)
61      root = increaseRootHeight(h);
62    int height = root.height;
63    Node<T> node = root.node;
64    Search<T> res = null;
65    while(true) {
66      Contents<T> cts = node.contents;
67      int i = search(cts.items, v);
68      if (-i - 1 == cts.items.length) {
69        node = cts.link;
70      } else {
71        res = new Search<T>(node, cts, i);
72        if (height <= h)
73          srchs[height] = res;
74        if (height == 0)
75          return;
76        if (i < 0) i = -i - 1;
77        node = cts.children[i];
78        height--;
79      }
80    }
81  }
```

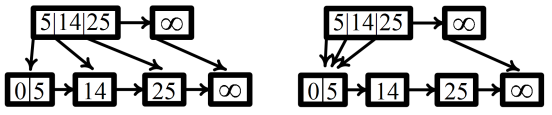
Figure 5: Inserting `v` into the set

D. Deletion & node compaction: `remove(v)`

Figure 6 shows the code for the `remove` operation. An element $v \in$ `T` is removed by searching for the presence of the element at the leaf level of the tree. If v is found, then construct a new array of elements minus v and update the node with a compare-and-swap. If the compare-and-swap is successful then return `true`. Otherwise retry the operation. If v is not found, then the `remove` operation returns `false`. The `moveForward` method is called when retrying a `remove` operation. The method accepts a node and an element, and returns a Search object containing the first node of the current linked list level with an element x such that $x \geq v$. The non-Java operator $\setminus$ used on Line 88 represents the set difference operation. Given an array (that does not hold duplicates) and an element in the array, the set difference operator returns a new array that does not contain the element.

```
82  boolean remove(T x) {
83    Search srch = traverseAndCleanup(v);
84    while(true) {
85      Node<T> node = srch.node;
86      Contents<T> cts = srch.contents;
87      if (srch.index < 0) return(false);
88      Object[] items = cts.items \ v;
89      Contents<T> update =
90       new Contents<T>(items,null,cts.link);
91      if (node.casContents(cts, update))
92        return(true);
93      srch = moveForward(node, v);
94    }
95  }
96
97  Search<T> traverseAndCleanup(T v) {
98    Node<T> node = root.node;
99    Contents<T> cts = node.contents;
100   Object[] items = cts.items;
101   int i = search(items, v);
102   T max = null;
103   while(cts.children != null) {
104     if (-i - 1 == items.length) {
105       if (items.length > 0)
106         max =(T) items[items.length-1];
107       node = cleanLink(node, cts);
108     } else {
109       if (i < 0) i = -i - 1;
110       cleanNode(node, cts, i, max);
111       node = cts.children[i];
112       max = null;
113     }
114     cts = node.cts;
115     items = cts.items;
116     i = search(items, v);
117   }     // end traverse routing nodes
118   while(true) {
119     if (i > -cts.items.length - 1)
120       return new Search<T>(node, cts, i);
121     node = cleanLink(node, cts);
122     cts = node.cts;
123     i = search(cts.items, v);
124   }     // end traverse leaf nodes
125 }
```

Figure 6: Deleting v from the set



(a) optimal references (b) suboptimal references

Figure 7: Optimal versus suboptimal child references.

The `remove` operation can introduce empty nodes and suboptimal child references. Empty nodes are relatively straightforward to eliminate. As stated in the previous section, it is forbidden to insert new elements into an empty node. Therefore a link or child reference to an empty node can be replaced by a reference to the immediate successor of the empty node. The elimination of an empty node is shown in Figure 8a.

An optimal child reference is a child reference that cannot be replaced by a reference to its immediate successor without resulting in a violation of the skip tree properties. Figure 7 illustrates two skip trees with identical nodes and highlights the differences between optimal and suboptimal child references.

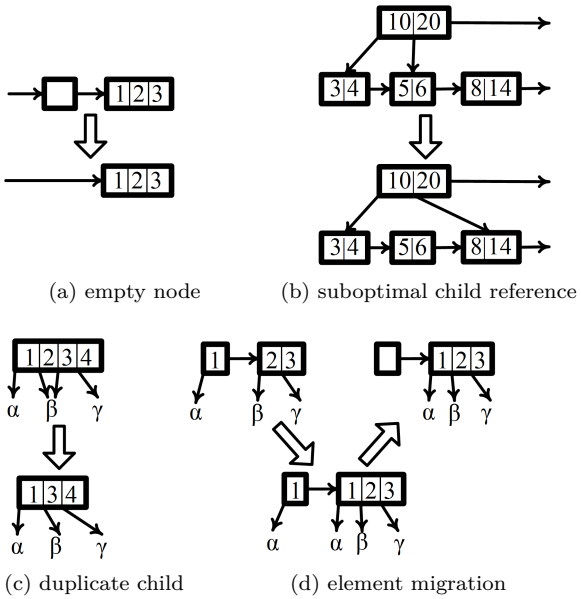
Four transformations can be applied to the tree to eliminate empty nodes and suboptimal child references. These transformations are shown in Figure 8. Node compaction can be applied in a lazy manner and is performed alongside the `remove` operation. An optimal node is defined as a node that does not benefit from any of the four node compaction operations.

Given a pair of adjacent elements A and B and their shared child reference to node n, the child reference is suboptimal if $\max(n) < A$. A suboptimal child reference can be repaired as shown in Figure 8b. To repair a child reference that is shared across two nodes, the max element of a node is stored when traveling across a linked-list level (Line 106).

As shown in Figure 8c, the node compaction operations can lead two neighboring child references to point to the same node. Given adjacent elements A, B, and C such that the child reference between A and B is equal to the child reference between B and C, then element B can be dropped from the node. It is unsafe to apply this transformation to a node that contains a single element. Assume element B from the previous example is the single element of some node n_B, and that element A is contained in node n_A and element C in node n_C. The intervals $\big(A, B\big]$ and $\big(B, C\big]$ satisfy property (D4) of Definition 1. If the leftmost child reference in n_C is a suboptimal child reference, and n_C is repaired while concurrently node n_B is removed, then the interval $\big(A, C\big]$ may no longer satisfy property (D4).

A copy and delete strategy is used to eliminate a node that contains a single element. This process is illustrated in Figure 8d. The inclusion of duplicate elements in routing nodes (Theorem 1) makes it possible to move the element to the successor node. Although element migration can be applied to any node that contains a single element, in practice it is applied only when child references α and β are pointing to the same node.

The deletion of the singleton element from a routing node can interfere with duplicate child elimination. Let us reuse the earlier example with elements A, B, and C stored in nodes n_A, n_B, and n_C. Thread 1 is performing node migration and has copied element B to node n_C. Thread 2 detects the duplicate copy of element B on node n_C and begins duplicate child elimination. Thread 1 finishes the node migration operation by removing the original



(a) empty node (b) suboptimal child reference

(c) duplicate child (d) element migration

Figure 8: Four types of node compaction

copy of element B from node n_B.

A third concurrent thread of execution attempts suboptimal child reference elimination on node n_C. The thread has read element B from node n_B before the element was deleted. As both copies of B have been eliminated, the correct max element that is a predecessor of n_C is less than B. B is too large to use as a max predecessor of node n_c and therefore the tree can reach an inconsistent state. This error is prevented by forbidding duplicate child elimination on the first pair of elements in a node. Nodes of two elements cannot be further compacted using duplicate child elimination. For these nodes, element migration is applied to the rightmost element of the node.

IV. Correctness

Liveness: We will show that `add` and `remove` are lock-free operations, and `contains` is a wait-free operation. Lock-free operations guarantee that at least one thread will complete in a finite number of steps. Wait-free operations guarantee that all threads will complete in a finite number of steps.

The `add` and `remove` operations each perform a finite number of traversals through the tree. The `add` operation performs up to two passes through the tree. A first pass inserts an element at the appropriate levels and a second pass splits each level after the insertion pass has completed. A `remove` operation consists of one pass through the tree that performs node compaction and possibly deletes an element from the leaf level.

In all three cases, each node of the tree is visited at most once per pass. In all cases, each visit of a node performs at most one successful compare-and-swap operation. The failure of one compare-and-swap operation implies the success of a compare-and-swap operation from another concurrently executing operation. Therefore system-wide progress is guaranteed.

The `contains` operation performs one pass through the tree. Each node of the tree is visited at most once, and the contents of each node are read at most once per visit. The `contains` operation performs no conditional atomic operations. Therefore per-thread progress is guaranteed for `contains` operations.

Linearizability: To demonstrate the skip tree algorithm is linearizable [19] it is sufficient to define a linearization point for each operation and then show that operations produce equivalent results to a sequential execution in which the operations appear to occur instantly at the linearization point.

The `contains`, `add`, and `remove` operations are linearized with two possible actions. If the operation does not change the state of the abstract data type, then the operation has been linearized by a volatile read on the contents of a node. If the operation changes the state of the abstract data type, then the linearization point occurs at the success of a compare-and-swap operation.

The linearization point for `contains(`v`)` occurs on Line 30 when a single leaf node is traversed or on Line 38 otherwise. At any snapshot in time there is exactly one leaf node that contains either v, or the smallest element in the set that is greater than v, or $+\infty$ if all elements in the set are less than v.

The linearization point for `add(`v`)` occurs during the call to the `insertList` method at the leaf level (Line 47). The linearization point depends on whether the `add` operation is successful. A successful `add` operation is linearized on the compare-and-swap operation that inserts v into a leaf node. An unsuccessful `add` operation is linearized on the read of the contents of a leaf node that leads to the discovery of v inside the node.

An unsuccessful `remove(`v`)` operation is linearized at Line 114 or 122 at the volatile read of a leaf node that does not contain v and contains some element greater than v. A successful `remove` is linearized at Line 91 on the compare-and-swap that removes v from a leaf node.

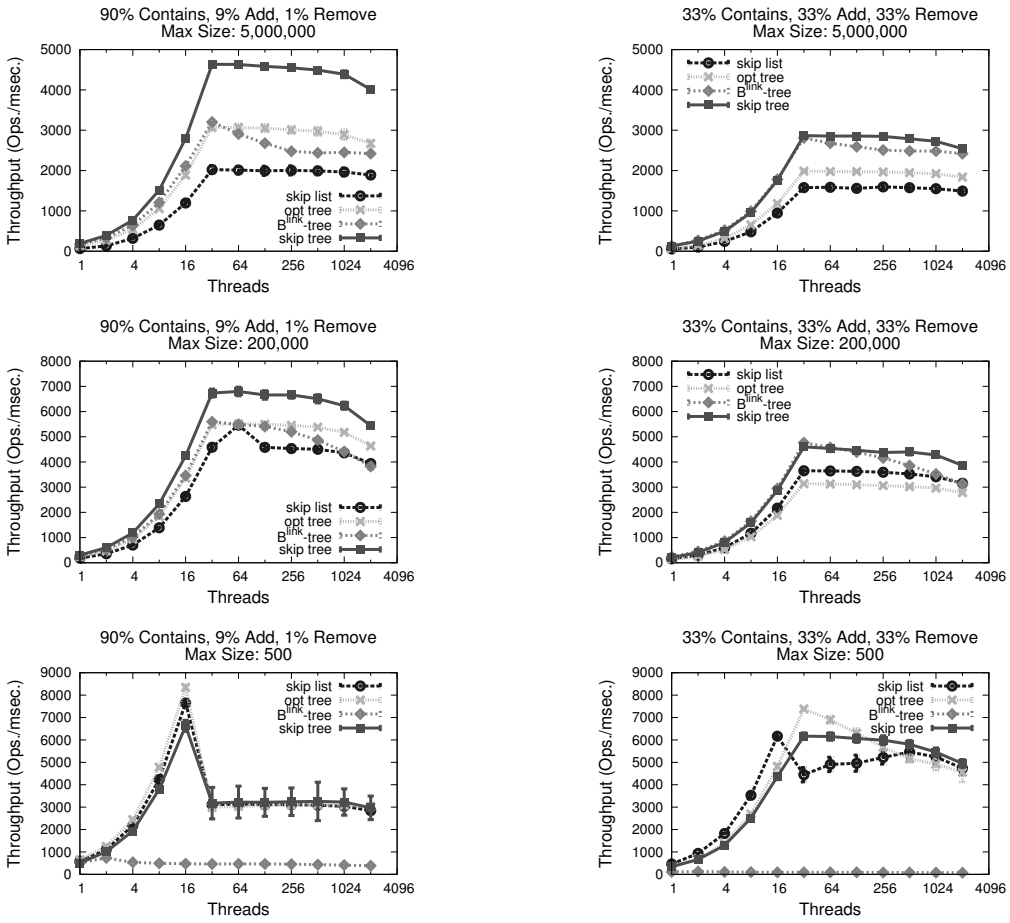


Figure 9: Sun Fire T1000 benchmarks. Intel Xeon results are available in the Supplemental Material.

V. Performance

Performance analysis has been conducted with an experimental design that is popular in the concurrent data structures literature [13, 15, 20]. Synthetic workloads are created that vary in proportions of `contains`, `add`, and `remove` operations and in the number of unique elements stored by the data structure. Half of the workloads use a $\frac{9}{10}$:$\frac{9}{100}$:$\frac{1}{100}$ ratio of operations. The other half use a $\frac{1}{3}$:$\frac{1}{3}$:$\frac{1}{3}$ ratio of operations. 5,000,000 operations are executed in each independent trial, while the total throughput of the data structure as measured by the number of concurrently executing threads varies from 1 to 2,048.

The maximum size of the tree is determined through selection of random elements from a uniform distribution with a range of 500, or 200,000, or 2^{32} integers. For each scenario, an upper bound on the size of the tree is the minimum of the range of input integers (500, 200,000 or 2^{32}) and the number of operations performed (5 million). Each independent trial is repeated 64 times. Integers that are designated for a `contains` or `remove` operation are pre-loaded into the tree prior to the beginning of a trial.

Benchmarks were evaluated on a Sun Fire T1000 and an Intel Xeon L5430. The benchmarks were executed on the 32-bit server version of the HotSpot Java Virtual Machine version 1.6.0 update 16. Explicit parameters for the virtual machine are 2 GB heap size and 128 kB thread stack size. The Sun Fire has 8 UltraSPARC T1 cores at 1.0 GHz and 32 logical processors. The cores share a 3 MB level-2 unified cache. The operating system version on the Sun Fire T1000 is Solaris 10.

The Xeon L5430 has 4 cores at 2.66 GHz and 8 logical processors. Each pair of cores shares a 6 MB level-2 unified cache. The operating system distribution is CentOS release 5.3 with Linux kernel 2.6.29-2.

We compare four implementations of linearizable concurrent ordered sets:

- skip-list - the `ConcurrentSkipListSet` in the `java.util.concurrent` library. Written by

members of the JCP JSR-166 Expert Group.
- skip-tree - our lock-free skip tree algorithm.
- opt-tree - the optimistic relaxed balance AVL tree algorithm of Bronson et al. [15].
- B^{link}-tree - a concurrent B-tree algorithm developed by Lehman and Yao [16] and refined by Sagiv [17].

Figure 9 shows the results of the synthetic benchmarks on the Sun Fire T1000. Tic marks denote the mean of the repeated experiments and error bars denote the standard deviation. In all graphs, a higher value on the vertical axis denotes improved performance.

The skip-tree structure is controlled by a single parameter, q, the failure rate of underlying geometric distribution. The B^{link}-tree is also controlled by a single parameter, M, the minimum node size. Parameter variations for each of the six scenarios were conducted. The parameter value with the best average performance was selected ($q = 1/32$, $M = 128$). The parameter variations and all benchmark results on the Intel Xeon L5430 are available in the supplemental material.

Using a maximum set size of 5,000,000 elements, our algorithm outperforms the skip-list by 129% on the read-dominated scenario and 81% on the write-dominated scenario. The opt-tree outperforms the skip-list by 51% and 88% in the two scenarios, and the B^{link}-tree by 58% and 77%. The skip tree exhibits the greatest improvement to system throughput under read-dominated scenarios with a large working set size. Averaged over all scenarios, the skip-tree outperforms the skip-list by 41% and the opt-tree outperforms the skip-list by 26%.

Averaged over all scenarios excluding the maximum range of 500 elements, the B^{link}-tree outperforms the skip-list by 41%. Averaged over all scenarios the B^{link}-tree performance drops to -4%. The B^{link}-tree performs wait-free searches when implemented for the memory-disk boundary where a page can be accessed from disk atomically. When implemented in main memory, the tree uses shared reader-writer locks [21, 22]. The reader-writer lock strategy is a bottleneck when there are only a handful of nodes in the data structure.

On the Xeon L5430, our algorithm outperforms the skip-list by 128% on the read-dominated scenario and 82% on the write-dominated scenario, using a maximum set size of 5,000,000 elements. The opt-tree outperforms the skip-list by 134% and 136% in the two scenarios, and the B^{link}-tree by 69% and 60%.

The opt-tree has the greatest improvement relative to the other data structures on the Intel processor. Averaged over all scenarios on the Xeon L5430, the skip-tree outperforms the skip-list by 41% and the opt-tree outperforms the skip-list by 56%. We are further investigating the possible causes for the improved performance of the opt-tree on the Xeon processor relative to the Sun Fire.

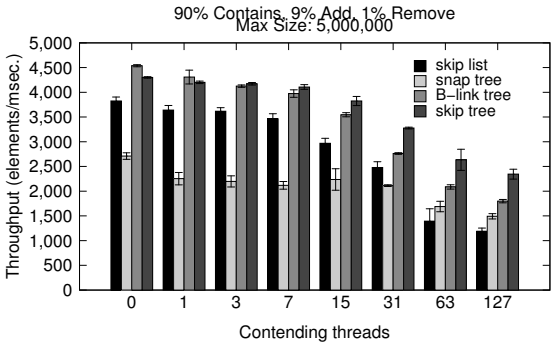


Figure 10: Iteration throughput of a single thread.

The performance of sequential iteration through the set is examined while concurrent operations are performed. The 90% `contains`, 9% `add`, and 1% `remove` workload is selected with a maximum set size of 5,000,000 elements. A single thread continuously iterates over the elements of the set. Competing threads perform `contains`, `add`, and `remove` operations. The throughput of the iterator thread is shown in Figure 10.

To measure the performance of sequential iteration, the opt-tree is replaced by the snap-tree, an extended algorithm that provides support for fast cloning and snapshots. Our experiments confirmed the original findings that the snap-tree outperforms the opt-tree for sequential iteration [15]. The skip tree shows a 18% improvement over the skip list with zero thread contention and a 97% improvement at the highest thread contention. The snap-tree shows a 29% decrease in performance over the skip list at zero thread contention and a 25% improvement at the highest thread contention.

VI. Conclusion

In this paper we have developed a lock-free multiway search tree algorithm for concurrent applications with large working set sizes. The lock-free skip tree has relaxed structural properties that allow atomic operations to modify the tree without invalidating the consistency of the data structure. Optimal paths through the tree are temporarily violated by mutation operations, and eventually restored using online node compaction.

Experimental evidence shows that our lock-free skip tree outperforms a highly tuned concurrent skip

list under workloads of various proportions of operations and working set sizes. The max throughput of our algorithm is on average 41% higher than the throughput of the skip list, and 129% higher on the workload of the largest working set size and read-dominated operations.

VII. Supplemental Material

Parameter variations, benchmark results on a Xeon processor, and an implementation of the skip tree are available from http://www.cs.virginia.edu/~ms6ep/ICPP2010.

References

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, pp. 20–24, March 1995.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley," University of California at Berkeley, Tech. Rep., 2006.

[3] R. Murphy, "On the effects of memory latency and bandwidth on supercomputer application performance," in *IEEE 10th International Symposium on Workload Characterization*, 2007.

[4] W. Pugh, "Skip lists: A probablistic alternative to balanced trees," *Communications of the ACM*, vol. 33, no. 6, pp. 668–676, June 1990.

[5] ——, "Concurrent maintenance of skip lists," University of Maryland, College Park, Tech. Rep. UMIACS-TR-90-80, 1990.

[6] K. Fraser, "Practical lock-freedom," Ph.D. dissertation, University of Cambridge, 2003.

[7] J. I. Munro, T. Papadakis, and R. Sedgewick, "Deterministic skip lists," in *Proceedings of the third annual ACM-SIAM symposium on Discrete Algorithms*, 1992.

[8] X. Messeguer, "Skip trees, an alternative data structure to skip lists in a concurrent approach," *Theoretical Informatics and Applications*, vol. 31, no. 3, pp. 251–269, 1997.

[9] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, 1999, pp. 285–397.

[10] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul, "Concurrent cache-oblivious B-trees," in *Proceedings of the Seventeenth ACM Symposium on Parallelism in Algorithms and Architectures*, 2005.

[11] M. A. Bender, G. S. Brodal, R. Fagerberg, D. Ge, S. He, H. Hu, J. Iacono, and A. Lopez-Ortiz, "The cost of cache-oblivious searching," in *Proceedings of the 44th Annual Symposium on Foundations of Computer Science*, 2003, pp. 271–280.

[12] J. D. Valois, "Lock-free linked lists using compare-and-swap," in *Proceedings of the 14th annual ACM symposium on Principles of distributed computing*, 1995.

[13] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the 14th annual ACM symposium on Parallel algorithms and architectures*, 2002.

[14] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the 15th International Conference on Distributed Computing*, 2001.

[15] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, "A practical concurrent binary search tree," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principals and Practice of Parallel Programming*, 2010.

[16] P. L. Lehman and S. B. Yao, "Efficient locking for concurrent operations on B-trees," *ACM Transactions on Database Systems*, vol. 6, no. 4, pp. 650–670, 1981.

[17] Y. Sagiv, "Concurrent operations on B-trees with overtaking," in *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1985.

[18] IBM, *IBM System/370 Extended Architecture, Principles of Operation*, 1983, publication no. SA22-7085.

[19] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," in *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, July 1990, pp. 463–492.

[20] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, "A simple optimistic skiplist algorithm," in *Structural Information and Communication Complexity, 14th International Colloquium*, ser. Lecture Notes in Computer Science, G. Prencipe and S. Zaks, Eds., vol. 4474. Springer, 2007, pp. 124–138.

[21] T. Johnson and D. Sasha, "The performance of current B-tree algorithms," *ACM Transactions on Database Systems*, vol. 18, pp. 51–101, 1993.

[22] V. Srinivasan and M. J. Carey, "Performance of B+ tree concurrency control algorithms," *VLDB Journal*, vol. 2, pp. 361–406, 1993.