A Simple Optimistic Skip Tree Algorithm

Michael Spiegel[1] and Paul F. Reynolds, Jr.

Department of Computer Science
School of Engineering, University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, Virginia 22904-4740
[1]mspiegel@cs.virginia.edu, Tel: +1-804-241-9123

Abstract

The skip list has proven to be a relatively effective data structure for concurrent applications. However the skip list design exhibits poor spatial locality of reference, and thus performs poorly when it encounters the so-called memory wall. The memory wall has been identified as one of the fundamental challenges to distributed computing. This paper introduces a simple optimistic dense skip tree algorithm as an alternative to the concurrent skip list, with a design that exhibits high spatial locality of reference. The optimistic dense skip tree algorithm implements a concurrent cache-conscious randomized data structure. A simple lock-based concurrent algorithm is inspired by the simple optimistic skip list algorithm of Herlihy et al. [1]. The optimistic dense skip tree algorithm outperforms both the optimistic skip list and the best known lock-free skip list algorithms across a broad range of tests. Its simple design supports straightforward proofs that the concurrent skip tree algorithm implements a linearizable set, that it is deadlock free, and that searching can be performed without acquiring any locks. A series of benchmarks shows the optimistic dense skip tree to outperform the optimistic skip list by 14-98% and the lock-free skip list 26-92%, when the working set size exceeds the cache size. For these reasons, the skip tree is a superior alternative to the skip list for concurrent applications that must contend with the deleterious effects of the memory wall.

This paper is a regular submission. It is eligible for the best student paper award.

1 Introduction

The skip list is a randomized data structure that is balanced probabilistically in order to yield expected $O(\log n)$ complexity costs for common operations [2]. The local balancing of the skip list has made it an attractive data structure in a variety of distributed applications such as constant-degree routing networks [3], peer-to-peer filesystems [4, 5], and concurrent lock-free sorted sets, sorted maps, and priority queues [6–9]. Herlihy et al. published the simple optimistic skip list algorithm that performs well under the most common search patterns, yet is 'simple' because the algorithm is easier to reason about than competing lock-free skip list algorithms [1]. In this spirit, we introduce the simple optimistic dense skip tree algorithm. The dense skip tree is a variation on the skip list that we have designed to probabilistically achieve a cache-conscious access of data [10]. It obeys the same asymptotic properties as the concurrent skip list, and shows significant performance improvements over the skip list when the working set size is larger than the cache capacity.

The term "memory wall" was coined over fifteen years ago to describe the growing disparity between microprocessor and memory performance [11, 12]. The memory wall has been identified as one of the fundamental challenges to distributed computing. One report on the landscape of parallel computing found that out of thirteen broad classes of computational problems, the memory wall was the major obstacle to good performance for almost half of these classes [13]. Although the memory wall was initially applied to differences in both bandwidth and latency, the flat performance of memory latency has come to dominate the memory wall. Over the past two decades, while processor performance has doubled every 18 months, memory latency has improved only by about 7% a year [14]. Furthermore in the time that memory bandwidth doubles, latency improves by no more than a factor of 1.2 to 1.4 [15].

The simple optimistic dense skip tree algorithm implements a concurrent abstract set. The optimistic dense skip tree implementation is shown to be linearizable and deadlock-free, and the search operation is shown to be wait-free. The expected number of cache misses when searching on a dense skip tree is shown to be less than the expected number of cache misses when searching on a skip list. A series of synthetic benchmarks is constructed to measure the performance of the optimistic dense skip tree algorithm, as compared to both the optimistic skip list algorithm and the lock-free skip list algorithm available in the Java™ SE 6 concurrency library. At peak performance, the optimistic dense skip tree outperforms the optimistic skip list by 14-98% and the lock-free skip list 26-92%, when the working set size exceeds the cache size. As the problems associated with the memory latency bottleneck are expected to escalate in a many-core era, the need for concurrent cache-conscious data structures is fundamental.

2 Optimistic Skip Tree Algorithm

The dense skip tree data structure and the concurrent skip tree algorithm are described in this section. The concurrent dense skip tree algorithm implements a concurrent abstract set data type with identical semantics to the optimistic skip list algorithm [1]. Three methods are supported: `add`(v) adds v to the set and returns `true` iff v was not already in the set; `remove`(v) removes v from the set and returns `true` iff v was in the set; and `contains`(v) returns `true` iff v is in the set. The implementation is shown to be linearizable in Section 3. An object is said to be linearizable if all public methods have a linearization point at some step between their invocation and response, and all methods appear to occur instantly at their linearization point. It is also shown that the implementation is deadlock-free, and that the `contains` operation is wait-free.

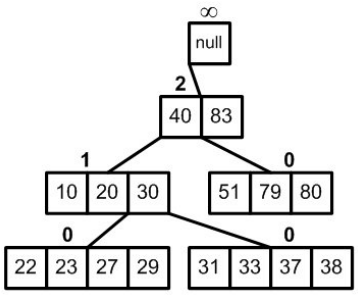

Figure 1: A dense skip tree

```
1  class Node<T> {
2    final Object[] items;
3    final Node<T>[] children;
4    final int height;
5    final Lock lock;
6    volatile boolean linked;
7  }
```

Figure 2: Node type declaration

Definition 1. A *dense skip tree* [10] is defined to be a multiway search tree, such that the following properties hold for all nodes:

(D1) Each node contains k keys in sorted order and possibly-null references to $k+1$ children for $k \geq 1$.
(D2) Each node contains a height h for $h \geq 0$.
(D3) Each key is assigned a random height at insertion.
(D4) A node with height h has children with heights that are less than h.
(D5) The left subtree of any key contains only keys of lesser value.
(D6) The right subtree of any key contains only keys of greater value.

The adjective 'dense' is used to differentiate our design from the original skip tree data structure [16]. The related work section contains a discussion of the differences between the skip tree and the dense skip tree (Section 5). For the remainder of the paper, the terms 'dense skip tree' and 'skip tree' will be used interchangeably.

An example of a dense skip tree is shown in Figure 1. The height of a node is the number listed above the node. A leaf node is defined as a node with height 0. The root node is defined as the node with the maximum height. The root node shown in the figure is defined to be a sentinel node. The sentinel node consists of one null item reference and two children references. The sentinel item reference is the only null item reference in the data structure. A node is augmented with two additional components for concurrent operations (Figure 2). A lock must be acquired in order to write to the node. A boolean `linked` flag is used to indicate whether a node belongs to the skip tree. The `linked` flag is initially true and is allowed to transition only from true to false. Reading from a node is a lock-free operation.

The `contains`, `add`, and `remove` operations all employ the `find` method to assist in their operation (Figure 3). The `find` method takes four arguments: a target item, a target height, an array of Node references, and an array of integer offsets into the arrays of the Node references. The third and fourth arguments to the `find` method are used to return additional values; they are used in a write-only context by the method. The method traces a path through the skip tree and records the path in the third and fourth arguments. The `path` array consists of all those nodes that could potentially store the target item. The offsets array consists of the offset that is used to index the array `path[i].children` in order to reach the node `path[i+1]`. The `path` consists of nodes in descending height order, beginning with the sentinel node (line 10) and terminating with a null reference (line 26). The return value is an index into the `path` array. Its value is dependent on whether the target item is found. If the target item is found, then the return value is the index of the node that contains the item. If the target item is located, then the `contains` operation returns true, the `add` operation returns false, and the `remove` operation performs the deletion and returns

```
 8  int find(T v, int height, Node<T>[] path, int[] offsets) {
 9    path[0] = root; offsets[0] = 0;
10    Node<T> curr = root.children[0];
11    int level = 1, index, targetLvl = -1;
12    while (curr != null) {
13      path[level] = curr;
14      index = Arrays.binarySearch(curr.items, v);
15      if (index >= 0) {            // item located
16        offsets[level] = index;
17        return(level);
18      }
19      index = - index - 1;
20      offsets[level] = index;
21      if (targetLvl < 0 && curr.height <= height)
22        targetLvl = level;
23      curr = curr.children[index];
24      level++;
25    }
26    path[level] = null;
27    if (targetLvl < 0) targetLvl = level;
28    return(targetLvl);
29  }
```

Figure 3: The `find` helper method, used by the `contains`, `add`, and `remove` methods

true. If the target item is not found, then the return value of the `find` method is the index of the node in the `path` array with the tallest height that is less than or equal to the target height.

The `add` operation proceeds in four steps (Figure 4). First, the path of potential nodes is acquired via a call to the `find` method. Next, the operation determines if the item is already present in the skip tree. If the item is present, then it returns false (lines 39-42). The check for an existing item has two pre-conditions. The `path` array is terminated with a null reference, so it must be confirmed that the target node is not a null reference. The `offsets` array contains indices that are in the range $[0...k]$, which is one element larger than the number of items a node contains. If neither of these safety conditions is violated, then it is possible to perform equality testing to determine if the item is present. If the item is not present, the `add` operation employs an optimistic locking strategy. All nodes along the path that must be modified are locked in descending order. Only after a node has been locked can it then be validated. A node is considered valid if its `linked` flag is true, and if the next node along the path has not been replaced in the meantime (line 52). If validation fails, then all the locks are released in descending order, and the operation is retried.

If all the locked nodes pass the validation phase, then the final stage of the `add` operation is to perform the addition of the new item into the skip tree. The final stage is shown in the `performInsertion` helper method (Figure 9). Insertion consists of two phases. First, the new item must be added to the tree. Either a target node exists, in which case a new node is created which is an augmented copy of the target node with the new item inserted (lines 115-119). Otherwise, a new node must be created that consists of a single item (lines 121-122). Once a new node has been created, the children nodes along the path below it must be split. Splitting can be viewed as an "unzipping" process whereby a single path is unzipped into two paths, using the new item as a dividing barrier that separates the two paths. The `splitLeft` and `splitRight` methods return either a new node that consists of all the items and children that are either left or right of an index, respectively, or return null if the number of items that satisfy their respective conditions is zero. Note that because `splitLeft` and `splitRight` can potentially return null, this implies that the two

```
30 boolean add(T v) {
31   final int height = randomLevel();
32   final Node<T>[] path = new Node[MAX_LEVEL + 2];
33   final int[] offsets = new int[MAX_LEVEL + 2];
34   while (true) {
35     int findLevel = find(v, height, path, offsets);
36     int parentLevel = findLevel - 1;
37     Node<T> parent = path[parentLevel], child = path[findLevel];
38     int offset = offsets[findLevel];
39     if ((child != null) && (offset < child.items.length)
40         && (child.items[offset].equals(v))) {
41         return(false);
42     }
43     int highestLocked = -1;
44     try {
45       Node<T> current = parent;
46       boolean valid = true;
47       for(int level = parentLevel; valid && (current != null); level++) {
48         current.lock();
49         highestLocked = level;
50         offset = offsets[level];
51         Node<T> next = path[level + 1];
52         valid = current.linked && current.children[offset] == next;
53         current = next;
54       }
55       if (!valid) continue;
56       offset = offsets[parentLevel];
57       parent.children[offset] = performInsertion(v, height,
58             findLevel, path, offsets);
59       return true;
60     } finally {
61       for (int level = parentLevel; level <= highestLocked; level++) {
62         path[level].unlock();
63       }
64     }
65   }
66 }
```

Figure 4: The `add` method

```
67 boolean contains(T v) {
68   Node<T>[] path = new Node[MAX_LEVEL + 2];
69   int[] offsets = new int[MAX_LEVEL + 2];
70   int findLevel = find(v, 0, path, offsets);
71   Node<T> child = path[findLevel];
72   int offset = offsets[findLevel];
73   return((child != null) && (offset < child.items.length)
74     && (child.items[offset].equals(v)));
75 }
```

Figure 5: The `contains` method

```
76  boolean remove(T v) {
77    final Node<T>[] path = new Node[MAX_LEVEL + 2];
78    final int[] offsets = new int[MAX_LEVEL + 2];
79    while(true) {
80      int cLevel = find(v, 0, path, offsets);
81      int pLevel = cLevel - 1;
82      Node<T> parent = path[pLevel], child = path[cLevel];
83      int pOffset = offsets[pLevel], cOffset = offsets[cLevel];
84      if ((child == null) ||
85          (cOffset == child.items.length) ||
86          !(child.items[cOffset].equals(v))) return(false);
87      boolean done = false;
88      try {
89        parent.lock();
90        child.lock();
91        if (!(parent.linked && child.linked &&
92          parent.children[pOffset] == child)) {
93          continue;
94        }
95        aquireJoinLocks(child, cOffset);
96        Node<T> newChild = performDeletion(child, cOffset);
97        parent.children[pOffset] = newChild;
98        releaseJoinLocks(parent, child, cOffset);
99        done = true;
100       return(true);
101     } finally {
102       if (!done) {
103         parent.unlock();
104         child.unlock();
105       }
106     }
107   }
108 }
```

Figure 6: The `remove` method

paths do not necessarily have the same number of nodes. When the splitting process has completed, then the `performInsertion` method will return the new child node. The new left and right paths have been attached to the new child node.

The `contains` and `remove` operations are variations of the `add` operation. The `contains` method is a wait-free method. It consists of a call to the `find` method, followed by an equality test that is identical to the equality test employed by the `add` operation (Figure 5).

The `remove` operation also begins with a call to the `find` method, followed by the same test for equality as `contains` or `add` operations (Figure 6). If the target item is found, then the child node containing the item and its parent node are locked in descending order. Once the parent and child are locked, a validation step must occur to ensure that both nodes are linked into the tree, and that the parent is still pointing to the child. If validation fails, then the remove operation retries. Otherwise the target item is removed, creating a new child node if-and-only-if the child node contains two items or more. Finally, the left and right paths underneath the child node must be joined into one new path. The next node along the left path is always the rightmost child of that node (if it exists), and similarly the next node along the right path is always the leftmost child. Locking the left and right paths does not require validation. Since the `remove` operation has acquired the lock on the target child, and locks are acquired in descending order on the left and right paths, the linked fields along these paths must be valid when the child node is locked.

3 Correctness and Cache-Consciousness

This section sketches three correctness proofs for the skip tree algorithm, and a cache-conscious proof of the skip tree. It is shown that the algorithm implements a linearizable set, that it is deadlock free, and that the `contains` operation is wait-free. In addition, it is shown that the `contains` operation has a smaller expected number of cache misses as compared to the skip list algorithm.

3.1 Linearizability

First, several observations are made about node initialization and node invariants. Nodes are initialized with an ordered list of keys, an ordered list of children, and a height. The `linked` field is initialized to true. For exposition, we assume that unused nodes are never reclaimed, and there is an inexhaustible supply of new nodes. The keys of a node never change (i.e., `keys` $= \{v_0, \ldots, v_k\}$ is stable), the height of a node never changes (i.e, `height` $= h$ is stable), and the `linked` field of a node is never set to true (i.e., `linked` is stable). The `children` field of a node is not stable, but can only be modified when the `lock` field of the node has been acquired. The `linked` field of a node can only be set to false when the `lock` is acquired. The "leads to" relationship is defined, in order to express a reachability relationship among keys in the tree.

Definition 2. Given a node n and key v, define the relation $\rightarrow$ such that $n \rightarrow v$ (read "node n leads to key v") iff v is one of the keys of n, or there exists a node m such that m is a child of n, and $m \rightarrow v$.

A key v is said to be a member of the skip tree iff the sentinel node leads to v. The `add` and `remove` methods lock a set of nodes that correspond to a path through the skip tree of those nodes that could potentially store the target item (lines 47-54 and 89-95). Let v_i denote the i^{th} key of some node, and c_i and c_{i+1} denote the i^{th} and $(i+1)^{st}$ children of that node, respectively. It can be shown that the skip tree algorithm preserves the following property:

Property 1. If $c_i \rightarrow v'$ then $v' < v_i$. If $c_{i+1} \rightarrow v'$ then $v' > v_i$.

Property 1 can be used to show that if `sentinel` $\rightarrow v$ in any reachable state of the algorithm, then `sentinel` $\rightarrow v$ in any subsequent state unless some node n in the tree is replaced by a copy of n such that k has been removed (line 144). In particular, if v' is a new key inserted into the skip tree, and the height of v' is less than the height of v, then $c_i \rightarrow v'$ iff $v' < v$ and $c_{i+1} \rightarrow v'$ iff $v' > v$. If the height of v' is greater than or equal to the height of v, and the intersection of $path(v)$ and $path(v')$ is non-null, then the node containing v will be split into two nodes. If the height of v' is greater than or equal to the height of v, and the intersection of the two paths is null, then the subtree containing v is unaffected by the insertion of v'.

Similarly, if `sentinel` $\rightarrow v$ does not hold in some reachable state of the algorithm, then it does not hold in any subsequent state unless a new node is added that is either a copy of an existing node with the addition of key v (line 115), or a singleton node that contains v (line 121). The linearization point of the `add` operation occurs when the new subtree that results from splitting a path is inserted into the tree, at line 58.

3.2 Deadlock Freedom and Wait-Freedom

The algorithm is deadlock-free because locks are always acquired in a top-down manner. More precisely, if a thread locks a node at height h, then it will not attempt to acquire a lock on a node with height less than or equal to h. The `contains` operation is wait-free because it does not acquire any locks and it never retries. The `contains` operation performs exactly one traversal of the tree, from the sentinel node to the bottom of the tree.

3.3 Cache-Consciousness Properties

In this section, an upper bound for the search cost of a skip tree is derived. This bound will be greater than a known upper bound for the search cost of a skip list. However, using these bounds it will be shown that an upper bound for the number of cache misses of a skip tree is less than the upper bound for the number of cache misses of a skip list. Let λ represent a skip list with random levels drawn from a geometric distribution with success probability p, and let $q = 1 - p$, and $Q = \frac{1}{q}$. Let τ represent a skip tree with random levels drawn from a geometric distribution with success probability p. If S_λ is an upper bound on the search cost of a skip list, then it has been shown that S_λ has expected value $Q \log_Q n + \frac{1}{p}$ [2]. Moreover this upper bound has been shown to be a good approximation for the true expected search cost of a skip list [17].

Lemma 1. For S_τ, an upper bound on the search cost of a skip tree, $E[S_\tau] = Q \log_Q n + \frac{Q}{p}$.

The total search cost of a skip tree is the sum of search costs for all nodes of the skip tree that are searched. We have previously shown that node sizes of a dense skip tree are distributed according to the geometric distribution $p^{s-1}q$, for $s \geq 1$ [10]. Let M_τ represent the expected maximum value from a sequence of n independent trials drawn from the random level generator of skip tree τ. M_τ has been shown to be at most $\log_Q n + \frac{1}{p}$ [2]. This upper bound for M_τ is used as an approximation for the expected number of nodes visited on a skip tree search. Then S_τ is the sum of M_τ independent trials with a geometric distribution of success rate q, plus an additional term to account for the shifted geometric distribution. Combining these properties yields the following result: $E[S_\tau] = E[NB(M_\tau, q) + M_\tau] = M_\tau \frac{p}{q} + M_\tau = \frac{1}{q} M_\tau = Q \log_Q n + \frac{Q}{p}$.

Lemma 2. The expected object reference cost of a skip tree is lower than the expected object reference cost of a skip list.

Let O_τ and O_λ denote an upper bound on the object reference cost of a skip tree and a skip list, respectively. Whereas the search cost is defined as the number of keys traversed during a

search operation, the object reference cost is defined as the number of references traversed during a search operation. The object reference cost will serve as an approximation for the number of cache misses on the algorithm. A skip list node stores exactly one key reference. Therefore $E[O_\lambda] = 2E[S_\lambda] = 2\left(Q \log_Q n + \frac{1}{p}\right)$. A skip tree node stores multiple key references. The number of nodes visited during a skip tree search has an upper bound of $E[M_\tau]$. Combining M_τ and S_τ yields $E[O_\tau] = E[S_\tau + M_\tau] = Q \log_Q n + \frac{Q}{p} + \log_Q n + \frac{1}{p} = E[O_\lambda] - \frac{Q}{p}\left(\log_Q n - 1\right)$. Thus $E[O_\tau] < E[O_\lambda]$ for $n > Q$.

4 Performance

Performance benchmarks of the optimistic skip tree were measured using scenarios adapted from the benchmarks used to evaluate the optimistic skip list [1]. Four scenarios were evaluated in various proportions of add, remove, and contains operations. In each experiment 5,000,000 randomly chosen operations were executed. Under one scenario the operations consist of 90% `contains`, 9% `add`, and 1% `remove` operations, and in another scenario the operations consist of 0% `contains`, 50% `add`, and 50% `remove` operations. The keys for each operation were selected uniformly at random from a specified range of either 200,000 or 2^{32} random integers. Keys that are designated for a `contains` or `remove` operation are pre-loaded into the data structure prior to the beginning of a test. The number of threads was varied across independent trials from 1 thread to 4,096 threads. Each independent trial was repeated 96 times. The results of these benchmarks are shown in Figures 7-8. Tic marks denote the mean of the repeated experiments and error bars denote standard deviation.

The optimistic skip tree was evaluated against the optimistic skip list implementation from Herlihy et al. [1] and the lock-free skip list implementation that is available from the Java SE 6 java.util.concurrent package developed by the JSR 166 concurrency utilities group [18]. The optimistic skip list implementation used is the optimized implementation of Herlihy et al. [1] that was used for their original benchmarks, which varies slightly from the algorithm that is presented in their paper.

Benchmarks were evaluated on an Intel Xeon X5460 with four cores at 3.16 GHz and 8 logical processors, and a Sun Fire T1000 with eight cores at 1.0 GHz and 32 logical processors. Each core of the Xeon processor has a 32 kB level-1 data cache and a 32 kB level-1 instruction cache. All cores share a 12 MB level-2 unified cache. Each core of the UltraSPARC T1 processor has a 8 kB level-1 data cache and a 16 kB instruction cache. The cores share a 3 MB level-2 unified cache. The benchmarks were executed on the 32-bit server version of the HotSpot Java Virtual Machine version 1.6.0 update 12. The random height generator for the optimistic skip list implementation and the lock-free skip list implementation were taken from the java.util.concurrent package, which returns 0 with probability $3/4$, i with probability $2^{-(i+2)}$ for $i \in [1, 30]$, and 31 with probability 2^{-32}. The random generator for the optimistic skip tree implementation is a geometric distribution with success probability $15/16$.

The first two scenarios were evaluated under a range of 200,000 integers and 5,000,000 operations (Figure 7). These scenarios are designed to measure data structure performance when the working set fits within the cache hierarchy. The range of keys limits the maximum size of the data structure. The final two scenarios have been evaluated with a range of all possible 32-bit integer values and 5,000,000 operations (Figure 8). The scenarios are designed to measure performance when the working set is largely outside of the cache hierarchy and inside main memory. Under these conditions there is a significant performance advantage of the optimistic skip tree as compared to the optimistic skip list and the lock-free skip list. At peak performance, the optimistic skip tree outperforms the lock-free skip list by 26-92% and the optimistic skip list by 14-98%. The optimistic skip tree exhibits

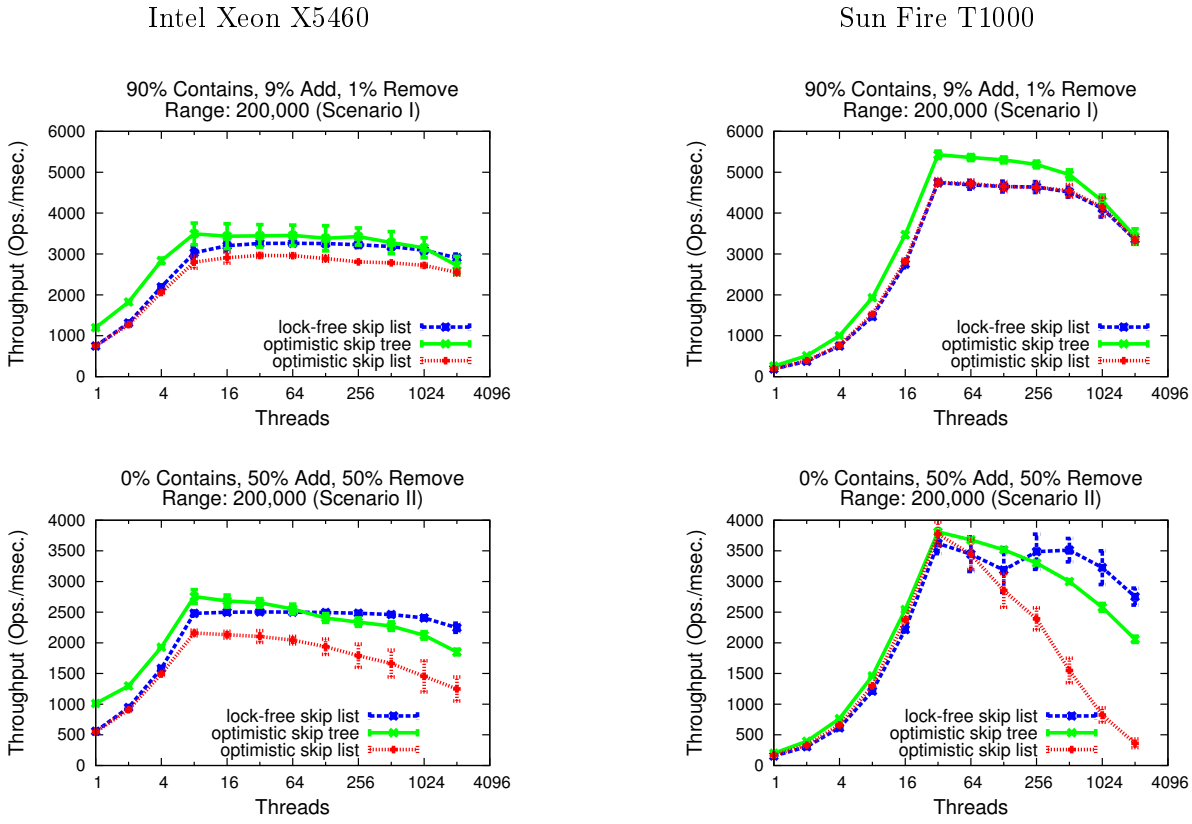

Figure 7: Throughput in operations per millisecond with a range of 200,000 integers and either 90% contains, 9% add, and 1% remove operations or 0% contains, 50% add, and 50% remove operations.

a degradation of performance under conditions of high thread contention. This behavior should be eliminated in a future lock-free concurrent skip tree algorithm.

5 Related Work

The sequential skip list [2] and the concurrent skip list [19] were invented by Bill Pugh. Several lock-free algorithms for concurrent skip lists have been published [6–8, 18]. As a response to these numerous lock-free implementations, a simple optimistic skip list algorithm was published by Herlihy et al. [1]. The simple optimistic skip list has a relatively straightforward proof of correctness, and performs as well as the lock-free skip list algorithms under many common search patterns. The skip tree was introduced by Messeguer [16] as a generalization of the skip list for concurrent operations. A sketch of the skip tree algorithm is shown without a discussion of the consistency model assumed by that algorithm. The dense skip tree is a redesign of the skip tree with the purpose of increasing spatial locality of reference. The dense skip tree is distinguished from the skip tree in that it contains no white nodes, which are nodes that store zero keys, and all paths in a dense skip tree from the root to the leaves are not of identical length.

Numerous techniques have been developed for the cache-conscious access of data within the framework of the multilevel memory hierarchy. These techniques include loop transformation [20, 21], data coloring [22], garbage collection [23], dynamic profiling [24], cache-conscious structure layout [25, 26], and cache-oblivious structure layout [27, 28]. These techniques are all deterministic in nature. Our randomized data structure is a novel addition to the set of deterministic techniques

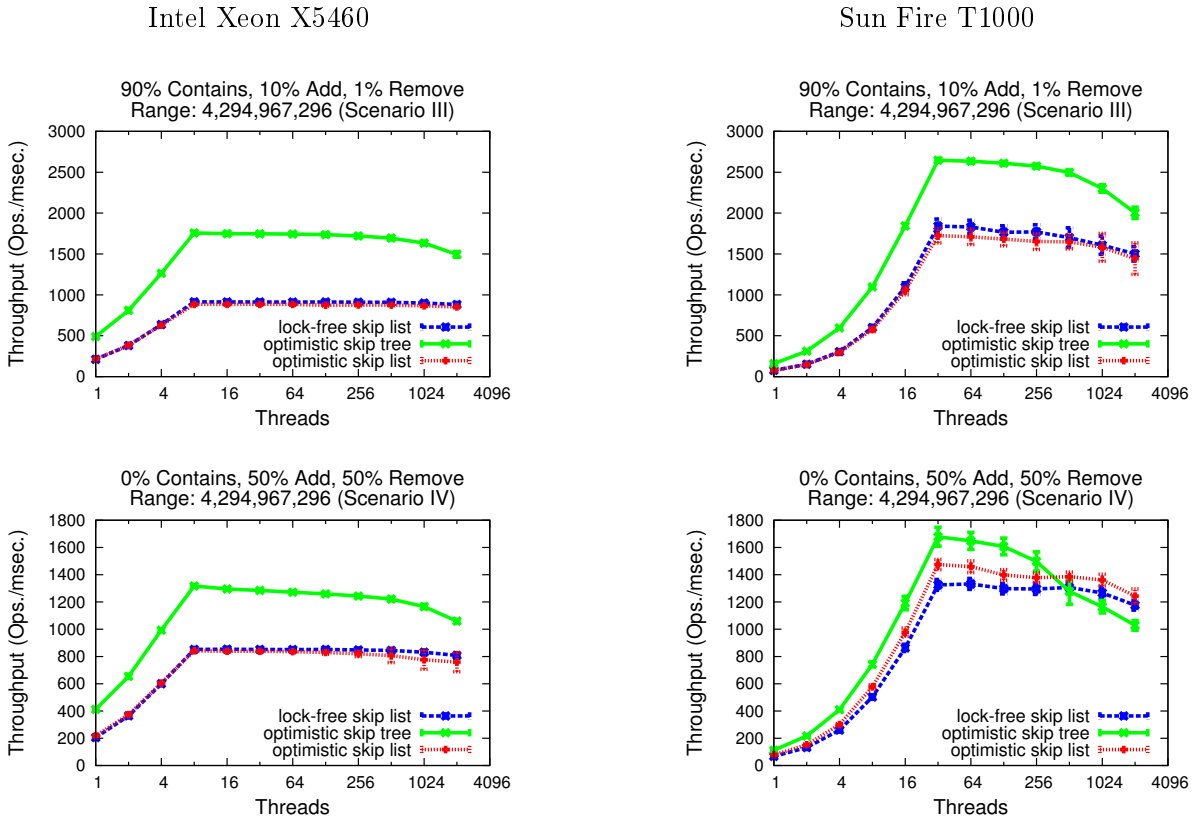

Figure 8: Throughput in operations per millisecond with a range of 2^{32} integers and either 90% contains, 9% add, and 1% remove operations or 0% contains, 50% add, and 50% remove operations.

that are used to reduce latency effects associated with the memory wall.

6 Conclusions

The simple optimistic dense skip tree algorithm has been shown to outperform both the optimistic skip list and the lock-free skip list algorithm when the working set size is larger than the cache capacity. It was shown that the optimistic skip tree algorithm implements a linearizable set, that it is deadlock free, and that the `contains` operation is wait-free. In addition, the `contains` operation has a smaller expected number of cache misses as compared to the skip list algorithms. The next logical phase is to develop a lock-free concurrent skip tree algorithm. It should exhibit the same peak performance behavior of the optimistic skip tree algorithm without performance degradation caused by lock contention. The concurrent skip tree has been shown to be a superior alternative to the concurrent skip list when the working set size exceeds the cache size.

Acknowledgments

We would like to thank Herlihy et al. [1] for access to their optimized optimistic skip list implementation. In addition, we wish to thank Jack Davidson and Wesley Weimer for access to their multicore servers to run our benchmarks.

Appendix

```
109 Node<T> performInsertion(T v, int height,
110       int level, Node<T>[] path, int[] offsets) {
111   Node<T> curr = path[level], newChild;
112   Node<T> lPred, rPred;
113   int lIndex, rIndex;
114   if (curr != null && curr.height == height) {
115     newChild = new Node<T>(curr, v, offsets[level]);
116     curr.linked = false;
117     lIndex = offsets[level];
118     rIndex = lIndex + 1;
119     curr = path[++level];
120   } else {
121     newChild = new Node<T>(v, height);
122     lIndex = 0; rIndex = 1;
123   }
124   lPred = rPred = newChild;
125   while(curr != null) {
126     Node<T> newLeft, newRight;
127     newLeft = curr.splitLeft(offsets[level]);
128     newRight = curr.splitRight(offsets[level]);
129     if (newLeft != null) {
130       lPred.children[lIndex] = newLeft;
131       lPred = newLeft; lIndex = lPred.items.length;
132     }
133     if (newRight != null) {
134       rPred.children[rIndex] = newRight;
135       rPred = newRight; rIndex = 0;
136     }
137     curr.linked = false; curr = path[++level];
138   }
139   return(newChild);
140 }
```

Figure 9: The `performInsertion` helper method, used by the `add` method

```
141  Node<T> performDeletion(Node<T> child, int cOffset) {
142    Node<T> retval = null;
143    child.linked = false;
144    retval = child.remove(cOffset);
145    Node<T> left = child.children[cOffset];
146    Node<T> right = child.children[cOffset + 1];
147    if (left == null && right == null) return(retval);
148    if (left == null && retval == null) return(right);
149    if (right == null && retval == null) return(left);
150    if (left == null) {
151      retval.children[cOffset] = right; return(retval);
152    } else if (right == null) {
153      retval.children[cOffset] = left; return(retval);
154    }
155    int currOffset = cOffset, nextOffset;
156    Node<T> current = retval;
157    Node<T> next, nextLeft, nextRight;
158    do {
159      if (left.height > right.height) {
160        next = new Node<T>(left);
161        left.linked = false;
162        nextOffset = left.items.length;
163        nextLeft = left.children[left.items.length];
164        nextRight = right;
165      } else if (right.height > left.height) {
166        next = new Node<T>(right);
167        right.linked = false;
168        nextOffset = 0;
169        nextLeft = left;
170        nextRight = right.children[0];
171      } else {
172        next = new Node<T>(left, right);
173        left.linked = right.linked = false;
174        nextOffset = left.items.length;
175        nextLeft = left.children[left.items.length];
176        nextRight = right.children[0];
177      }
178      if (retval == null) retval = next;
179      else current.children[currOffset] = next;
180      current = next; currOffset = nextOffset;
181      left = nextLeft; right = nextRight;
182    } while(left != null && right != null);
183    if (left == null) current.children[currOffset] = right;
184    if (right == null) current.children[currOffset] = left;
185    return(retval);
186  }
```

Figure 10: The `performDeletion` helper method, used by the `remove` method

References

[1] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In Giuseppe Prencipe and Shmuel Zaks, editors, *Structural Information and Communication Complexity (SIROCCO), 14th International Colloquium*, volume 4474 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2007.

[2] William Pugh. Skip lists: A probablistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[3] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, 2002.

[4] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: a scalable overlay network with practical locality properties. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems*, volume 4, 2003.

[5] James Aspnes and Gauri Shah. Skip graphs. *ACM Transactions on Algorithms*, 3(4):384–393, 2007.

[6] Keir Fraser. *Practical Lock-Freedom.* PhD thesis, University of Cambridge, 2003.

[7] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 50–59, 2004.

[8] Håkan Sundell and Philippas Tsigas. Scalable and lock-free concurrent dictionaries. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1438–1445, 2004.

[9] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A provably correct scalable concurrent skip list. In *Proceedings of the 10th International Conference On Principles of Distributed Systems (OPODIS)*, 2006.

[10] Michael Spiegel and Paul F. Reynolds, Jr. The dense skip tree: A cache-conscious randomized data structure. Technical Report CS-2009-05, Department of Computer Science, University of Virginia, 2009.

[11] William A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23:20–24, March 1995.

[12] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers*, April 2004.

[13] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical report, University of California at Berkeley, 2006.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, fourth edition edition, 2007.

[15] David A. Patterson. Latency lags bandwidth. *Communications of the ACM*, 47:71–75, 2004.

[16] Xavier Messeguer. Skip trees, an alternative data structure to skip lists in a concurrent approach. *Theoretical Informatics and Applications*, 31(3):251–269, 1997.

[17] Thomas Papadakis, J. Ian Munro, and Patricio V. Poblete. Analysis of the expected search cost in skip lists. In *Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory*, pages 160–172, Bergen, Sweden, July 1990.

[18] Doug Lea, Joseph Bowbeer, Brian Goetz, David Holmes, and Tim Peierls. Java specification request (JSR) 166: Concurrency utilities. http://jcp.org/en/jsr/detail?id=166, September 2004.

[19] William Pugh. Concurrent maintenance of skip lists. Technical Report UMIACS-TR-90-80, University of Maryland, College Park, 1990.

[20] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, 1991.

[21] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. *ACM SIGOPS Operating Systems Review*, 28(28):252 – 262, December 1994.

[22] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, 1999.

[23] Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the 1st International Symposium on Memory Management*, 1998.

[24] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. In *Proceedings of the 8th International Conference on Architecture Support for Programming Languages and Operating Systems*, 1998.

[25] Jun Rao and Kenneth A. Ross. Making B+-trees cache conscious in main memory. In *Proceedings of the 2000 SIGMOD Conference*, 2000.

[26] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. Fractal prefetching B+ trees: Optimizing both cache and disk performance. In *Proceedings of the ACM 2002 SIGMOD Conference*, 2002.

[27] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35:341–358, 2005.

[28] Lars Arge, Gerth Stolting Brødal, and Rolf Fagerberg. *Handbook on Data Structures and Applications*, chapter 38 Cache-Oblivious Data Structures. CRC Press, 2004.