A Provably Correct Scalable Concurrent Skip List

Maurice Herlihy[1,2], Yossi Lev[1,2], Victor Luchangco[2], and Nir Shavit[2]

[1] Computer Science Department, Brown University, Providence, RI 02912
[2] Sun Microsystems Laboratories, 1 Network Drive, Burlington, MA 01803

Abstract. We propose a new concurrent skip list algorithm distinguished by a combination of simplicity and scalability. The algorithm employs optimistic synchronization, searching without acquiring locks, followed by short lock-based validation before adding or removing nodes. It also logically removes an item before physically unlinking it. Unlike some other concurrent skip list algorithms, this algorithm preserves the skip list properties at all times, which facilitates reasoning about its correctness. Experimental evidence shows that this algorithm performs as well as the best previously known algorithm under most circumstances.

1 Introduction

Skip lists [7] are an increasingly important data structure for storing and retrieving ordered in-memory data. In this paper, we propose a new concurrent skip-list algorithm that appears to perform as well as the best existing concurrent skip list implementation under most conditions. The principal advantage of our implementation is that it is much simpler, and much easier to reason about.

The `ConcurrentSkipListMap`, written by Doug Lea based on work by Fraser and Harris [2] and released as part of the Java^{TM} SE 6 platform, is the best concurrent skip-list implementation that we are aware of. This algorithm is lock-free, and performs well in practice. The principal limitation of this implementation is that it is complicated. Certain interleavings can cause the usual skip list structure to be violated, sometimes transiently, and sometimes permanently. These violations do not affect performance or correctness, but they make it difficult to reason about the correctness of the algorithm. By contrast, the algorithm presented here preserves the skip list structure at all times. The algorithm is simple enough that we are able to provide a straightforward proof of correctness.

Our algorithm employs two complementary techniques. First, it is *optimistic*: methods traverse the list without acquiring locks. When a method discovers the items it is seeking, it locks the item and its predecessors, and then validates that the list is unchanged. Second, removing an item involves *logically* deleting it by marking it before it is *physically* removed (unlinked) from the list.

Experimental tests show that despite its simplicity, this algorithm performs as well as the lock-free Lea algorithm, except under conditions of extreme contention in multiprogrammed environments. In Section 6 we discuss some approaches for future work to address this issue.

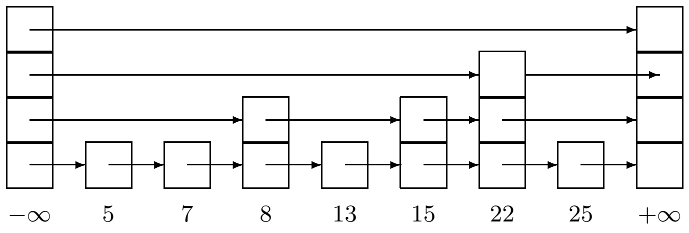


Fig. 1. A skip list with maximum height of 4. The number below each node (i.e., array of next pointers) is the key of that node, with $-\infty$ and $+\infty$ as the keys for the left and right sentinel nodes respectively.

2 Background

A skip list [7] is a linked list that is sorted by *key*, and in which nodes are assigned a random *height*, up to some maximum height, where the frequency of nodes of a particular height decreases exponentially with the height. A node in a skip list has not just one successor, but a number of successors equal to its height: each node stores a pointer to the next node in the list of each height up to its own. For example, a node of height 3 stores three "next pointers", one to the next node of height 1, one to the next node of height 2, and one to the next node of height 3. Figure 1 illustrates a skip list in which the keys are integers.

We think of a skip list as having several *layers* of lists, and we talk about the predecessor and successor of a node at each layer. The list at each layer, other than the bottom layer, is a sublist of the list at the layer beneath it. Because there are exponentially fewer nodes of greater heights, we can find a key quickly by searching first at higher layers, skipping over large numbers of shorter nodes and progressively working downward until a node with the desired key is found, or else the bottom layer is reached. Thus, the expected time complexity of skip-list operations is logarithmic in the length of the list.

It is convenient to have *left sentinel* and *right sentinel* nodes, at the beginning and end of the lists respectively. These nodes have the maximum height, and initially, when the skip list is empty, the right sentinel is the successor of the left sentinel at every layer. The left sentinel's key is smaller, and the right sentinel's key is greater, than any key that may be added to the set. Searching the skip list thus always begins at the left sentinel.

3 Our Algorithm

We present our concurrent skip-list algorithm in the context of an implementation of a set object supporting three methods, `add`, `remove` and `contains`: `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. We show that our implementation is *linearizable* [5]; that is, every operation appears to take place atomically at some point (the

```
class Node {
  int key;
  int topLayer;
  Node** nexts;
  bool marked;
  bool fullyLinked;
  Lock lock;
};
```

Figure 1.2. A node

linearization point) between its invocation and response. We also show that the implementation is deadlock-free, and that the `contains` operation is *wait-free*; that is, a thread is guaranteed to complete a `contains` operation as long as it keeps taking steps, regardless of the activity of other threads.

Our algorithm builds on the lazy-list algorithm of Heller et al. [3], a simple concurrent linked-list algorithm with an optimistic fine-grained locking scheme for the `add` and `remove` operations, and a wait-free `contains` operation: we use lazy lists at each layer of the skip list. As in the lazy list, the key of each node is strictly greater than the key of its predecessor, and each node has a `marked` flag, which is used to make `remove` operations appear atomic. However, unlike the simple lazy list, we may have to link the node in at several layers, and thus might not be able to insert a node with a single atomic instruction, which could serve as the linearization point of a successful `add` operation. Thus, for the lazy skip list, we augment each node with an additional flag, `fullyLinked`, which is set to `true` after a node has been linked in at all its layers; setting this flag is the linearization point of a successful `add` operation in our skip-list implementation. Figure 1.2 shows the fields of a node.

A key is in the abstract set if and only if there is an unmarked, fully linked node with that key in the list (i.e., reachable from the left sentinel).

To maintain the skip-list structure—that is, that each list is a sublist of the list at lower layers—changes are made to the list structure (i.e., the `nexts` pointers) only when locks are acquired for all nodes that need to be modified. (There is one exception to this rule involving the `add` operation, discussed below.)

In the following detailed description of the algorithm, we assume the existence of a garbage collector to reclaim nodes that are removed from the skip list, so nodes that are removed from the list are not recycled while any thread might still access them. In the proof (Section 4), we reason as though nodes are never recycled. In a programming environment without garbage collection, we can use solutions to the repeat offenders problem [4] or hazard pointers [6] to achieve the same effect. We also assume that keys are integers from `MinInt+1` to `MaxInt-1`. We use `MinInt` and `MaxInt` as the keys for `LSentinel` and `RSentinel`, which are the left and right sentinel nodes respectively.

Searching in the skip list is accomplished by the `findNode` helper function (see Figure 1.3), which takes a key `v` and two maximal-height arrays `preds` and

```
int findNode(int v,
             Node* preds[],
             Node* succs[]) {
  int lFound = -1;
  Node* pred = &LSentinel;
  for (int layer = MaxHeight-1;
       layer >= 0;
       layer--) {
    Node* curr = pred->nexts[layer];
    while (v > curr->key)  {
      pred = curr; curr = pred->nexts[layer];
    }
    if (lFound == -1 && v == curr->key) {
      lFound = layer;
    }
    preds[layer] = pred;
    succs[layer] = curr;
  }
  return lFound;
}
```

Figure 1.3. The `findNode` helper function

`succs` of node pointers, and searches exactly as in a sequential skip list, starting at the highest layer and proceeding to the next lower layer each time it encounters a node whose key is greater than or equal to `v`. The thread records in the `preds` array the last node with a key less than `v` that it encountered at each layer, and that node's successor (which must have a key greater than or equal to `v`) in the `succs` array. If it finds a node with the sought-after key, `findNode` returns the index of the first layer at which such a node was found; otherwise, it returns -1. For simplicity of presentation, we have `findNode` continue to the bottom layer even if it finds a node with the sought-after key at a higher level, so all the entries in both `preds` and `succs` arrays are filled in after `findNode` terminates (see Section 3.4 for optimizations used in the real implementation). Note that `findNode` does not acquire any locks, nor does it retry in case of conflicting access with some other thread. We now consider each of the operations in turn.

3.1 The `add` operation

The `add` operation, shown in Figure 1.4, calls `findNode` to determine whether a node with the key is already in the list. If so (lines 59–66), and the node is not marked, then the `add` operation returns `false`, indicating that the key is already in the set. However, if that node is not yet fully linked, then the thread waits until it is (because the key is not in the abstract set until the node is fully linked). If the node is marked, then some other thread is in the process of deleting that node, so the thread doing the `add` operation simply retries.

```
bool add(int v) {
  int topLayer = randomLevel(MaxHeight);
  Node* preds[MaxHeight], succs[MaxHeight];
  while (true) {
    int lFound = findNode(v, preds, succs);
    if (lFound ≠ −1) {
      Node* nodeFound = succs[lFound];
      if (!nodeFound->marked) {
        while (!nodeFound->fullyLinked) {;}
        return false;
      }
      continue;
    }
    int highestLocked = −1;
    try {
      Node *pred, *succ, *prevPred = null;
      bool valid = true;
      for (int layer = 0;
           valid && (layer ≤ topLayer);
           layer++) {
        pred = preds[layer];
        succ = succs[layer];
        if (pred ≠ prevPred) {
          pred->lock.lock();
          highestLocked = layer;
          prevPred = pred;
        }
        valid = !pred->marked && !succ->marked &&
                pred->nexts[layer]==succ;
      }
      if (!valid) continue;

      Node* newNode = new Node(v, topLayer);
      for (int layer = 0;
           layer ≤ topLayer;
           layer++) {
        newNode->nexts[layer] = succs[layer];
        preds[layer]->nexts[layer] = newNode;
      }

      newNode->fullyLinked = true;
      return true;
    }
    finally { unlock(preds, highestLocked); }
  }
```

Figure 1.4. The add method

If no node was found with the appropriate key, then the thread locks and *validates* all the predecessors returned by `findNode` up to the height of the new node (lines 69–84). This height, denoted by `topNodeLayer`, is determined at the very beginning of the `add` operation using the `randomLevel` function.[3] Validation (lines 81–83) checks that for each layer $i \leq$ `topNodeLayer`, `preds[`i`]` and `succs[`i`]` are still adjacent at layer i, and that neither is marked. If validation fails, the thread encountered a conflicting operation, so it releases the locks it acquired (in the `finally` block at line 97) and retries.

If the thread successfully locks and validates the results of `findNode` up to the height of the new node, then the `add` operation is guaranteed to succeed because the thread holds all the locks until it fully links its new node. In this case, the thread allocates a new node with the appropriate key and height, links it in, sets the `fullyLinked` flag of the new node (this is the linearization point of the `add` operation), and then returns `true` after releasing all its locks (lines 86–97). The thread writing `newNode->nexts[`i`]` is the one case in which a thread modifies the `nexts` field for a node it has not locked. It is safe because `newNode` will not be linked into the list at layer i until the thread sets `preds[`i`]->nexts[`i`]` to `newNode`, *after* it writes `newNode->nexts[`i`]`.

3.2 The `remove` operation

The `remove` operation shown in Figure 1.5, likewise calls `findNode` to determine whether a node with the appropriate key is in the list. If so, the thread checks whether the node is "okay to delete" (Figure 1.6), which means it is fully linked, not marked, and it was found at its top layer.[4] If the node meets these requirements, the thread locks the node and verifies that it is still not marked. If so, the thread marks the node, which logically deletes it (lines 111–121); that is, the marking of the node is the linearization point of the `remove` operation.

The rest of the procedure accomplishes the "physical" deletion, removing the node from the list by first locking its predecessors at all layers up to the height of the deleted node (lines 124–138), and splicing the node out one layer at a time (lines 140–142). To maintain the skip-list structure, the node is spliced out of higher layers before being spliced out of lower ones (though, to ensure freedom from deadlock, as discussed in Section 4, the locks are acquired in the opposite order, from lower layers up). As in the `add` operation, before changing any of the deleted node's predecessors, the thread validates that those nodes are indeed still the deleted node's predecessors. This is done using the `weakValidate` function, which is the same as `validate` except that it does not fail if the successor

[3] This function is taken from Lea's algorithm to ensure a fair comparison in the experiments presented in Section 5. It returns 0 with probability $\frac{3}{4}$, i with probability $2^{-(i+2)}$ for $i \in [1, 30]$, and 31 with probability 2^{-32}.

[4] A node found not in its top layer was either not yet fully linked, or marked and partially unlinked, at some point when the thread traversed the list at that layer. We could have continued with the `remove` operation, but the subsequent validation would fail.

```
bool remove(int v) {
  Node* nodeToDelete = null;
  bool isMarked = false;
  int topLayer = -1;
  Node* preds[MaxHeight], succs[MaxHeight];
  while (true) {
    int lFound = findNode(v, preds, succs);
    if (isMarked ||
        (lFound ≠ -1 && okToDelete(succs[lFound],lFound))){

      if (!isMarked) {
        nodeToDelete = succs[lFound];
        topLayer = nodeToDelete->topLayer;
        nodeToDelete->lock.lock();
        if (nodeToDelete->marked) {
          nodeToDelete->lock.unlock();
          return false;
        }
        nodeToDelete->marked = true;
        isMarked = true;
      }
      int highestLocked = -1;
      try {
        Node *pred, *succ, *prevPred = null;
        bool valid = true;
        for (int layer = 0;
             valid && (layer ≤ topLayer);
             layer++) {
          pred = preds[layer];
          succ = succs[layer];
          if (pred ≠ prevPred) {
            pred->lock.lock();
            highestLocked = layer;
            prevPred = pred;
          }
          valid = !pred->marked && pred->nexts[layer]==succ;
        }
        if (!valid) continue;

        for (int layer = topLayer; layer ≥ 0; layer--) {
          preds[layer]->nexts[layer] = nodeToDelete->nexts[layer];
        }
        nodeToDelete->lock.unlock();
        return true;
      }
      finally { unlock(preds,highestLocked); }
    }
    else return false;
  }
}
```

Figure 1.5. The `remove` method

```
bool okToDelete(Node* candidate, int lFound) {
  return (candidate->fullyLinked
          && candidate->topLayer==lFound
          && !candidate->marked);
}
```

Figure 1.6. The `okToDelete` method

```
bool contains(int v) {
  Node* preds[MaxHeight], succs[MaxHeight];
  int lFound = findNode(v, preds, succs);
  return (lFound ≠ −1
           && succs[lFound]->fullyLinked
           && !succs[lFound]->marked);
}
```

Figure 1.7. The `contains` method

is marked, since the successor in this case should be the node to be removed that was just marked. If the validation fails, then the thread releases the locks on the old predecessors (but not the deleted node) and tries to find the new predecessors of the deleted node by calling `findNode` again. However, at this point it has already set the local `isMarked` flag so that it will not try to mark another node. After successfully removing the deleted node from the list, the thread releases all its locks and returns `true`.

If no node was found, or the node found was not “okay to delete” (i.e., was marked, not fully linked, or not found at its top layer), then the operation simply returns `false` (line 148). It is easy to see that this is correct if the node is not marked because for any key, there is at most one node with that key in the skip list (i.e., reachable from the left sentinel) at any time, and once a node is put in the list (which it must have been to be found by `findNode`), it is not removed until it is marked. However, the argument is trickier if the node is marked, because at the time the node is found, it might not be in the list, and some unmarked node with the same key may be in the list. However, as we argue in Section 4, in that case, there must have been some time during the execution of the `remove` operation at which the key was not in the abstract set.

3.3 The `contains` operation

Finally, we consider the `contains` operation, shown in Figure 1.7, which just calls `findNode` and returns `true` if and only if it finds a unmarked, fully linked node with the appropriate key. If it finds such a node, then it is immediate from the definition that the key is in the abstract set. However, as mentioned above, if the node is marked, it is not so easy to see that it is safe to return `false`. We argue this in Section 4.

3.4 Implementation Issues

We implemented the algorithm in the Java^{TM} programming language, in order to compare it with Doug Lea's nonblocking skip-list implementation in the `java.util.concurrent` package. The array stack variables in the pseudocode are replaced by thread-local variables, and we used a straightforward lock implementation (we could not use the built-in object locks because our acquire and release pattern could not always be expressed using synchronized blocks).

The pseudocode presented was optimized for simplicity, not efficiency, and there are numerous obvious ways in which it can be improved, many of which we applied to our implementation. For example, if a node with an appropriate key is found, the `add` and `contains` operations need not look further; they only need to ascertain whether that node is fully linked and unmarked. If so, the `contains` operation can return `true` and the `add` operation can return `false`. If not, then the `contains` operation can return `false`, and the `add` operation either waits before returning `false` (if the node is not fully linked) or else must retry. The `remove` operation does need to search to the bottom layer to find all the predecessors of the node to be deleted, however, once it finds and marks the node at some layer, it can search for that exact node at lower layers rather than comparing keys.[5] This is correct because once a thread marks a node, no other thread can unlink it.

Also, in the pseudocode, `findNode` always starts searching from the highest possible layer, though we expect most of the time that the highest layers will be empty (i.e., have only the two sentinel nodes). It is easy to maintain a variable that tracks the highest nonempty layer because whenever that changes, the thread that causes the change must have the left sentinel locked. This ease is in contrast to the nonblocking version, in which a race between concurrent `remove` and `add` operations may result in the recorded height of the skip list being less than the actual height of its tallest node.

4 Correctness

In this section, we sketch a proof for our skip-list algorithm. There are four properties we want to show: that the algorithm implements a linearizable set, that it is deadlock-free, that the `contains` operation is wait-free, and that the underlying data structure maintains a correct skip-list structure, which we define more precisely below.

4.1 Linearizability

For the proof, we make the following simplifying assumption about initialization: Nodes are initialized with their key and height, their `nexts` arrays are initialized to all `null`, and their `fullyLinked` and `marked` fields are initialized to `false`.

[5] Comparing keys is expensive because, to maintain compatibility with Lea's implementation, comparison invokes the `compareTo` method of the `Comparable` interface.

Furthermore, we assume for the purposes of reasoning that nodes are never reclaimed, and there is an inexhaustible supply of new nodes (otherwise, we would need to augment the algorithm to handle running out of nodes).

We first make the following observations: The key of a node never changes (i.e., `key` $= k$ is stable), and the `marked` and `fullyLinked` fields of a node are never set to `false` (i.e., `marked` and `fullyLinked` are stable). Though initially `null`, `nexts[`i`]` is never written to `null` (i.e., `nexts[`i`]` $\neq$ `null` is stable). Also, a thread writes a node's `marked` or `nexts` fields only if it holds the node's lock (with the one exception of an `add` operation writing `nexts[`i`]` of a node before linking it in at layer i).

From these observations, and by inspection of the code, it is easy to see that in any operation, after calling `findNode`, we have `preds[`i`]->key` $<$ `v` and `succs[`i`]->key` $\geq$ `v` for all i, and `succs[`i`]->key` $>$ `v` for $i >$ `lFound` (the value returned by `findNode`). Also, for a thread in `remove`, `nodeToDelete` is only set once, and that unless that node was marked by some other thread, this thread will mark the node, and thereafter, until it completes the operation, the thread's `isMarked` variable will be `true`. We also know by `okToDelete` that the node is fully linked (and indeed that only fully linked nodes can be marked).

Furthermore, validation and the requirement to lock nodes before writing them ensures that after successful validation, the properties checked by the validation (which are slightly different for `add` and `remove`) remain true until the locks are released.

We can use these properties to derive the following fundamental lemma:

Lemma 1. *For a node* n *and* $0 \leq i \leq n$*`->topLayer`:*

$$n\texttt{->nexts[}i\texttt{]} \neq \texttt{null} \Longrightarrow n\texttt{->key} < n\texttt{->nexts[}i\texttt{]->key}$$

We define the relation $\rightarrow_i$ so that $m \rightarrow_i n$ (read "m leads to n at layer i") if m`->nexts[`i`]` $= n$ or there exists m' such that $m \rightarrow_i m'$ and m'`->nexts[`i`]` $= n$; that is, $\rightarrow_i$ is the transitive closure of the relation that relates nodes to their immediate successors at layer i. Because a node has (at most) one immediate successor at any layer, the $\rightarrow_i$ relation "follows" a linked list at layer i, and in particular, the layer-i list of the skip list consists of those nodes n such that `LSentinel` $\rightarrow_i n$ (plus `LSentinel` itself). Also, by Lemma 1, if $m \rightarrow_i n$ and $m \rightarrow_i n'$ and n`->key` $< n'$`->key` then $n \rightarrow_i n'$.

Using these observations, we can show that if $m \rightarrow_i n$ in any reachable state of the algorithm, then $m \rightarrow_i n$ in any subsequent state unless there is an action that splices n out of the layer-i list, that is, an execution of line 141. This claim is proved formally for the lazy-list algorithm in a recent paper [1], and that proof can be adapted to this algorithm. Because n must already be marked before being spliced out of the list, and because the `fullyLinked` flag is never set to `false` (after its initialization), this claim implies that a key can be removed from the abstract set only by marking its node, which we argued earlier is the linearization point of a successful `remove` operation.

Similarly, we can see that if `LSentinel` $\rightarrow_i n$ does *not* hold in some reachable state of the algorithm, then it does not hold in any subsequent state unless there is some execution of line 91 with $n =$ `newNode` (as discussed earlier, the previous line doesn't change the list at layer-i because `newNode` is not yet linked in then). However, the execution of that line occurs while `newNode` is being inserted and before `newNode` is fully linked. Thus, the only action that adds a node to a list at any level is the setting of the node's `fullyLinked` flag.

Finally, we argue that if a thread finds a marked node then the key of that node must have been absent from the list at some point during the execution of the thread's operation. There are two cases: If the node was marked when the thread invoked the operation, the node must have been in the skip list at that time because marked nodes cannot be added to the skip list (only a newly allocated node can be added to the skip list), and because no two nodes in the skip list can have the same key, no unmarked node in the skip list has that key. Thus, at the invocation of the operation, the key is not in the skip list. On the other hand, if the node was not marked when the thread invoked the operation, then it must have been marked by some other thread before the first thread found it. In this case, the key is not in the abstract set immediately after the other thread marked the node. This claim is also proved formally for the simple lazy list [1], and that proof can be adapted to this algorithm.

4.2 Maintaining the skip-list structure

Our algorithm guarantees that the skip-list structure is preserved at all times. By "skip-list structure", we mean that the list at each layer is a sublist of the lists at lower layers. It is important to preserve this structure, as the complexity analysis for skip lists requires this structure.

To see that the algorithm preserves the skip-list structure, note that linking new nodes into the skip list always proceeds from bottom to top, and while holding the locks on all the soon-to-be predecessors of the node being inserted. On the other hand, when a node is being removed from the list, the higher layers are unlinked before the lower layers, and again, while holding locks on all the immediate predecessors of the node being removed.

This property is not guaranteed by the lock-free algorithm. In that algorithm, after linking a node in the bottom layer, links the node in the rest of the layers from top to bottom. This may result in a state of a node that is linked only in its top and bottom layers, so that the list at the top layer is *not* a sublist of the list at the layer immediately beneath it, for example. Moreover, attempts to link in a node at any layer other than the bottom are not retried, and hence this state of nonconformity to the skip-list structure may persist indefinitely.

4.3 Deadlock freedom and wait-freedom

The algorithm is deadlock-free because a thread always acquires locks on nodes with larger keys first. More precisely, if a thread holds a lock on a node with key v then it will not attempt to acquire a lock on a node with key greater than or equal to v. We can see that this is true because both the `add` and `remove` methods acquire locks on the predecessor nodes from the bottom layer up, and

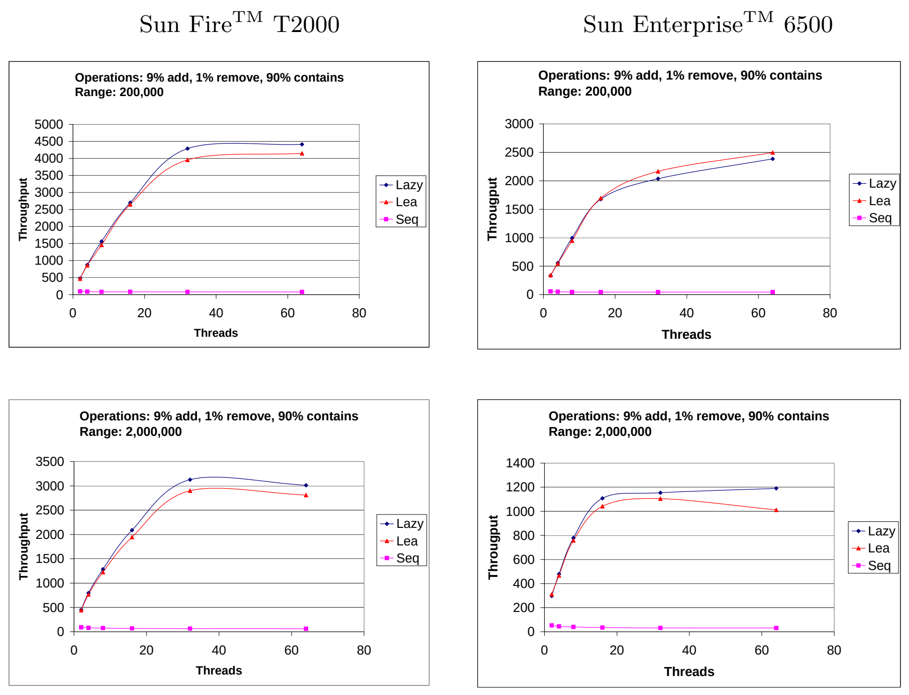


Fig. 8. Throughput in operations per millisecond of 1,000,000 operations, with 9% `add`, 1% `remove`, and 90% `contains` operations, and a range of either 200,000 or 2,000,000.

the key of a predecessor node is less than the key of a different predecessor node at a lower layer. The only other lock acquisition is for the node that a `remove` operation deletes. This is the first lock acquired by that operation, and its key is greater than that of any of its predecessors.

That the `contains` operation is wait-free is also easy to see: it does not acquire any locks, nor does it ever retry; it searches the list only once.

5 Performance

We evaluated our skip-list algorithm by implementing in the Java programming language, as described earlier. We compared our implementation against Doug Lea's nonblocking skip-list implementation in the `ConcurrentSkipListMap` class of the `java.util.concurrent` package, which is part of the Java$^{\text{TM}}$ SE 6 platform; to our knowledge, this is the best widely available concurrent skip-list implementation. We also implemented a straightforward sequential skip list, in which methods were `synchronized` to ensure thread safety, for use as a baseline in these experiments. We descibe some of the results we obtained from these experiments in this section.

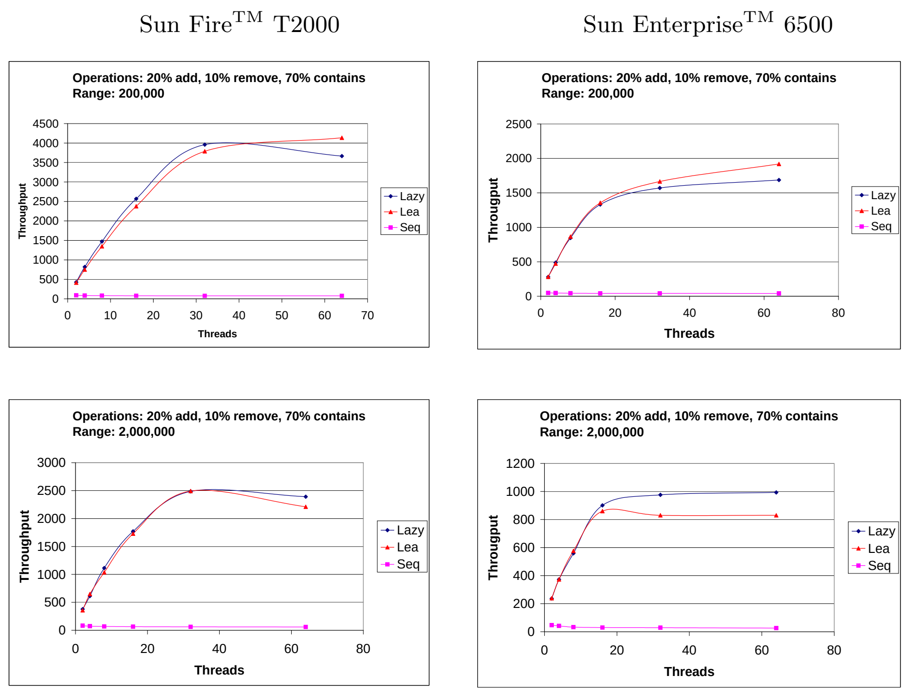
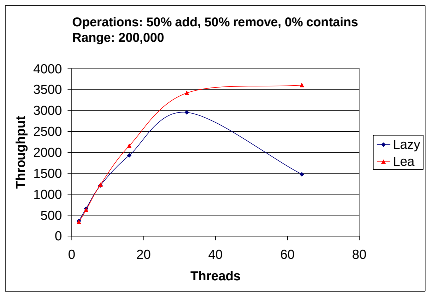


Fig. 9. Throughput in operations per millisecond of 1,000,000 operations with 20% `add`, 10% `remove`, and 70% `contains` operations, and range of either 200,000 or 2,000,000.

We present results from experiments on two multiprocessor systems with quite different architectures. The first system is a Sun Fire^{TM} T2000 server, which is based on a single UltraSPARC® T1 processor containing eight computing cores, each with four hardware strands, clocked at 1200 MHz. Each four-strand core has a single 8-KByte level-1 data cache and a single 16-KByte instruction cache. All eight cores share a single 3-MByte level-2 unified (instruction and data) cache, and a four-way interleaved 32-GByte main memory. Data access latency ratios are approximately 1:8:50 for L1:L2:Memory accesses. The other system is an older Sun Enterprise^{TM} 6500 server, which contains 15 system boards, each with two UltraSPARC® II processors clocked at 400 MHz and 2 Gbytes of RAM for a total of 30 processors and 60 Gbytes of RAM. Each processor has a 16-KByte data level-1 cache and a 16-Kbyte instruction cache on chip, and a 8-MByte external cache. The system clock frequency is 80 MHz.

We present results from experiments in which, starting from an empty skip list, each thread executes one million (1,000,000) randomly chosen operations. We varied the number of threads, the relative proportion of `add`, `remove` and `contains` operations, and the range from which the keys were selected. The key for each operation was selected uniformly at random from the specified range.

Sun Fire^{TM} T2000



Sun Enterprise^{TM} 6500

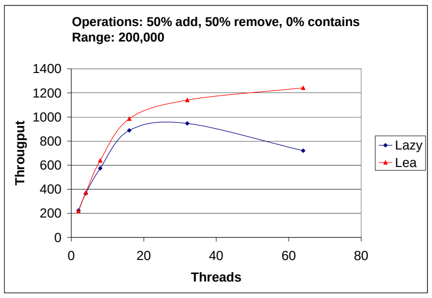


Fig. 10. Throughput in operations per millisecond of 1,000,000 operations, with 50% `add` and 50% `remove` operations, and a range of 200,000

In the graphs that follow, we compare the throughput in operations per millisecond, and the results shown are the average over six runs for each set of parameters.

Figure 8 presents the results of experiments in which 9% of the operations were `add` operations, 1% were `remove` operations, and the remaining 90% were `contains` operations, where the range of the keys was either two hundred thousand or two million. The different ranges give different levels of contention, with significantly higher contention with the 200,000 range, compared with the 2,000,000 range. As we can see from these experiments, both our implementation and Lea's scale well (and the sequential algorithm, as expected, is relatively flat). In all but one case (with 200,000 range on the older system), our implementation has a slight advantage.

In the next set of experiments, we ran with higher percentages of `add` and `remove` operations, 20% and 10% respectively (leaving 70% `contains` operations). The results are shown in Figure 9. As can be seen, on the T2000 system, the two implementations have similar performance, with a slight advantage to Lea in a multiprogrammed environment when the range is smaller (higher contention). The situation is reversed with the larger range. This phenomenon is more noticeable on the older system: there we see a 13% advantage to Lea's implementation on the smaller range with 64 threads, and 20% advantage to our algorithm with the same number of threads when the range is larger.

To explore this phenomenon, we conducted an experiment with a significantly higher level of contention: half `add` operations and half `remove` operations with a range of 200,000. The results are presented in Figure 10. As can be clearly seen, under this level of contention, our implementation's throughput degrades rapidly when approaching the multiprogramming zone, especially on the T2000 system. This degradation is not surprising: In our current implementation, when an `add` or `remove` operation fails validation, or fails to acquire a lock immediately, it simply calls `yield`; there is no proper mechanism for managing contention.

Since the `add` and `remove` operations require that the predecessors seen during the search phase be unchanged until they are locked, we expect that under high contention, they will repeatedly fail. Thus, we expect that a back-off mechanism, or some other means of contention control, would greatly improve performance in this case. To verify that a high level of conflict is indeed the problem, we added counters to count the number of retries executed by each thread during the experiment. The counters indeed show that many retries are executed on a 64 threads run, especially on the T2000. Most of the retries are executed by the `add` method, which makes sense because the `remove` method marks the node to be removed before searching its predecessors in lower layers, which prevents change of these predecessor's `next` pointers by a concurrent `add` operation.

6 Conclusions

We have shown how to construct a scalable, highly concurrent skip list using a remarkably simple algorithm. The principal open question is whether we can improve the algorithm's performance at high levels of contention. One simple approach is to use a more sophisticated back-off scheme when synchronization conflicts are detected. Another intriguing approach is to allow the randomness of the skip-list's height to be compromised under high contention. In the algorithm presented here, when a thread adds a new item, it gives up and retries if it encounters a synchronization conflict when linking the item at any level. In principle, a thread that encounters a conflict when linking the item at layer $\ell > 0$ could simply stop there, leaving the item linked at levels zero to $\ell - 1$, acting as if it had randomly chosen $\ell - 1$. The resulting data structure, while structurally still a skip list, would be a little flatter than it should be. Whether this approach is effective is the subject of future work.

References

1. Colvin, R., Groves, L., Luchangco, V., and Moir, M. Formal verification of a lazy concurrent list-based set. In *Proceedings of Computer-Aided Verification* (Aug. 2006).
2. Fraser, K. *Practical Lock-Freedom.* PhD thesis, University of Cambridge, 2004.
3. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Shavit, N., and Scherer III, W. N. A lazy concurrent list-based set algorithm. In *Proceedings of 9th International Conference on Principles of Distributed Systems* (Dec. 2005).
4. Herlihy, M., Luchangco, V., and Moir, M. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of Distributed Computing: 16th International Conference* (2002).
5. Herlihy, M., and Wing, J. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12*, 3 (July 1990), 463–492.
6. Michael, M. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems 15*, 6 (June 2004), 491–504.
7. Pugh, W. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM 33*, 6 (June 1990), 668–676.