Lock-Free Linked Lists and Skip Lists

Mikhail Fomitchev
Department of Computer Science,
York University

Eric Ruppert
Department of Computer Science
York University

ABSTRACT

Lock-free shared data structures implement distributed objects without the use of mutual exclusion, thus providing robustness and reliability. We present a new lock-free implementation of singly-linked lists. We prove that the worst-case amortized cost of the operations on our linked lists is linear in the length of the list plus the contention, which is better than in previous lock-free implementations of this data structure. Our implementation uses backlinks that are set when a node is deleted so that concurrent operations visiting the deleted node can recover. To avoid performance problems that would arise from traversing long chains of backlink pointers, we introduce flag bits, which indicate that a deletion of the next node is underway. We then give a lock-free implementation of a skip list dictionary data structure that uses the new linked list algorithms to implement individual levels. Our algorithms use the single-word C&S synchronization primitive.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures—*Distributed Data Structures*; D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*; F.2.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity

General Terms

Algorithms, Performance, Design, Reliability, Theory

Keywords

distributed, fault-tolerant, lock-free, linked list, skip list, efficient, analysis, amortized analysis.

1. INTRODUCTION

A common way to implement shared data structures in distributed systems is to use *mutual exclusion locks*. However, this approach has a major weakness: when one process holds a lock, no other processes can modify the locked part. Thus, a delay of one process can cause performance degradation and priority inversion. When halting failures can occur, this becomes particularly important, because the entire system can stop making progress if one process fails. By contrast, an implementation of a shared-memory object is *lock-free* (or *non-blocking*) if a finite number of steps taken by any process guarantees the completion of some operation. If an implementation is lock-free, delays or failures of individual processes do not block the progress of other processes in the system. Lock-free data structures also have the potential to have better performance, because several processes are allowed to modify a data structure at the same time.

Herlihy [4, 5] introduced the first *universal constructions* for designing lock-free data structures using the *Compare&Swap (C&S) synchronization primitive.* Others followed, but they suffer from several flaws, such as inefficiency, low parallelism, excessive copying, and generally high overhead, which often make them impractical. To achieve adequate performance, original algorithms, specific to a particular data structure, are usually required.

Implementing *linked lists* efficiently is very important, as they act as building blocks for many other data structures. We present a new lock-free implementation of a sorted singly-linked list, which handles all dictionary operations with a better average complexity than any prior implementation. Most recent implementations of lock-free linked lists [3, 8] were evaluated only by doing experimental testing. We believe that there exists a certain lack of theoretical development in this area, and our work addresses this problem. A *skip list* [12] is a dictionary data structure, that provides randomized algorithms for searches, insertions, and deletions that run in $O(\log n)$ expected time, where n is the number of elements in the skip-list. The expectation is taken over random choices made by the algorithms. We also give a lock-free implementation of a skip list that is based on using our linked list algorithms to maintain each level of the skip list. Recently, other lock-free skip list designs have been given independently of this work [2, 14, 15].

Our model is an *asynchronous shared-memory* distributed system of several processes, where an arbitrary number of process *halting failures* are allowed. Our algorithms use atomic single-word C&S synchronization primitives. The implementations that we present are *linearizable* [6].

Lock-free implementations allow individual operations to take arbitrarily many steps, so one generally cannot evaluate their worst-case cost. It is natural to analyze the average cost of operations instead, because this evaluates the performance of the system as a whole. To calculate the average

PODC'04, July 25–28, 2004, St. Johns, Newfoundland, Canada.

cost of operations in our linked list implementation, we use an amortized analysis that relies on a fairly complex technique of billing part of the cost of each operation S to concurrent operations that slow S down by modifying the data structure. The amortized cost of an operation S, denoted $\hat{t}(S)$, is equal to the actual cost of S plus the total cost billed to S from other operations minus the total cost billed from S to other operations. We measure the cost of operations as a function of the size of the list and the contention. The *point contention* at time T is the number of processes running concurrently at T. We define the *contention of operation* S, denoted $c(S)$, to be the maximum point contention during the execution of S. We prove that $\hat{t}(S) \in O(n(S) + c(S))$, where $n(S)$ is the number of elements in the list when S is invoked and $c(S)$ is the contention of S. The $O(n(S))$ term comes from the cost of traversing the list, while the overhead that comes from concurrency is bounded by $O(c(S))$. It then follows that for any execution E, the average cost of an operation in E is

$$\bar{t}_E \in O\left(\frac{\sum_{S\in E}(n(S) + c(S))}{m_E}\right) = O(\bar{n}_E + \bar{c}_E),$$

where the sum is taken over all operations S invoked during E, m_E is the total number of these operations. The values $\bar{n}_E$ and $\bar{c}_E$ are the average number of elements in the list during E and the average operation contention during E, which are defined as follows: $\bar{n}_E = \frac{\sum_{S\in E} n(S)}{m_E}$; $\bar{c}_E = \frac{\sum_{S\in E} c(S)}{m_E}$.

The rest of the paper is organized as follows. In Section 2 we discuss related work. We give our implementation of lock-free linked lists, including a sketch of the proof of correctness and analysis, in Section 3. We briefly present our implementation of lock-free skip lists in Section 4.

2. RELATED WORK

The first implementation designed for lock-free linked lists was presented by Valois [17]. The main idea of his approach was to maintain auxiliary nodes in between normal nodes of the list in order to resolve the problems that arise because of interference between concurrent operations. Also, each node in his list had a *backlink pointer* which was set to point to the predecessor when the node was deleted. These backlinks were then used to backtrack through the list when there was interference from a concurrent deletion. (A similar idea was used in an earlier, lock-based implementation of linked lists by Pugh [11].) Another lock-free implementation of linked lists was given by Harris [3]. His main idea was to *mark* a node before deleting it in order to prevent concurrent operations from changing its right pointer. We look at this implementation in detail in Section 3.1. Harris's algorithms are simpler than Valois's and his experimental results show that generally they also perform better. Yet another implementation of a lock-free linked list was proposed by Michael [8]. He used Harris's design to implement the underlying data structure, but his algorithms, unlike Harris's, were compatible with efficient memory management techniques, such as IBM freelists [7, 16] and the safe memory reclamation method [9].

Our linked lists are built combining the techniques of marking nodes [3] and using backlink pointers [11, 17], and also new ideas, such as the flag bits described in Section 3.1, which are introduced to improve the worst-case performance. We show that for any execution E, the average cost of an operation in the execution is $O(\bar{n}_E + \bar{c}_E)$, where $\bar{n}_E$ and $\bar{c}_E$ were defined in the introduction. To compare, the average cost per operation in Valois's implementation can be $\Omega(m_E)$, where m_E is the total number of operations invoked during E. This is possible even when $\bar{n}_E$ and $\bar{c}_E$ are $O(1)$ [17]. It is not hard to see that $\bar{n}_E + \bar{c}_E \leq m_E$ (because m_E includes both completed operations and operations that are currently in progress), and the difference can be quite significant. As we show in Section 3.1, the average cost of operations in Harris's implementation can be $\Omega(\bar{n}_E\bar{c}_E)$, which is also strictly worse than in our implementation.

Pugh's skip list data structure, originally designed for sequential accesses [12], is a natural candidate for concurrent dictionary implementations, since it has good expected performance without requiring any explicit, centralized balancing. Lock-based concurrent implementations have been given by Pugh [11] and by Lotan and Shavit [13]. Valois claimed that his lock-free linked list can easily be used to obtain a lock-free skip lists [17], but it is not clear how: for example, a process traversing his linked list must maintain a collection of pointers called a cursor, and it is difficult to do so when one descends through the levels of a skip list.

Sundell and Tsigas recently gave the first lock-free implementation of a skip list [14]. Their implementation supports the INSERT, UPDATE and DELETEMIN operations. They later extended it to implement the full range of dictionary operations [15]. Another recent implementation of lock-free skip lists using single-word C&S's was presented by Fraser [2]. Although both of these designs were done independently of ours and of each other, there are some similarities between the three resulting skip list algorithms. All use the marking technique [3] to implement deletions on the individual levels of the skip list. Fraser's algorithms use Harris's design style where an operation restarts if it detects interference from a concurrent operation. Sundell and Tsigas's design allows processes to overcome the interference in some cases by using backlink pointers [11, 17]. Our design employs backlink pointers and flag bits in order to ensure that processes can always recover efficiently from such interference. All implementations use helping (in different ways) to complete deletions that could block the progress of other operations. Sundell and Tsigas incorporate a reference counting scheme to handle memory management.

Fraser gives other skip list designs that use more powerful primitives, such as multi-word C&S and software transactional memory [2]. Experimental results on lock-free linked lists [3, 8] and skip lists [2, 14, 15] suggest that they can be a practical alternative to lock-based implementations.

3. LINKED LISTS

We now present our singly-linked list implementation. Our algorithms use the C&S primitive, which atomically executes the following code.

```
C&S (Word* $address$, Word $old\_val$, Word $new\_val$) : Word
$value = *address$
if ($value == old\_val$)
    $*address = new\_val$
return $value$
```

3.1 Linked List Design

The basic problem in designing a lock-free linked list is that when a process is deleting a node X by performing a

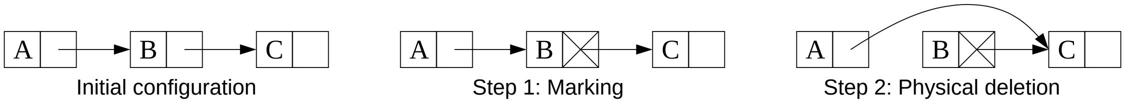

Figure 1: Harris's two-step deletion of a node.

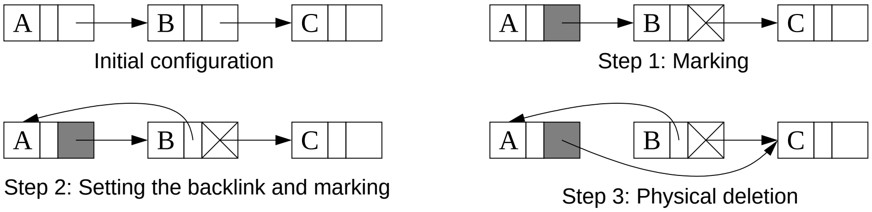
A
B
C
Initial configuration
A
B
C
Step 1: Marking
A
B
C
Step 2: Setting the backlink and marking
A
B
C
Step 3: Physical deletion

Figure 2: Three-step deletion of a node used in our implementation.

C&S on X's predecessor, there must be a guarantee that X's right pointer is not changed by a concurrent operation. Otherwise, incorrect executions can be constructed (see [17] or [3]). One of the ways to deal with this issue was given by Harris [3]. Our linked list implementation uses a similar technique, so we will look at Harris's implementation in more detail.

Harris replaced the right pointer of each node with a composite field, which we will call a *successor field*. The successor field consists of a right pointer and a *mark bit*. [1] When a process needs to change the right pointer of a node, it applies a C&S's to the successor field of that node. A mark bit acts as a toggle that is used to control when the right pointer of the node can be changed. Normally, the mark bit is 0. To delete a node B, a process uses two C&S's: the first *marks* B's successor field by setting its mark bit to 1, and the second removes B from the list, as illustrated in Figure 1, where marked successor fields are crossed. A node is *logically deleted* after the first step, and *physically deleted* after the second step. All of the C&S's performed by the algorithms modify only unmarked successor fields. Therefore, once the successor field of a node is marked, it never changes.

Harris's approach, however, has certain performance-related problems. Consider two processes P_1 and P_2 performing concurrent operations: P_1 attempts to insert a new node after node X, and P_2 attempts to delete node X. Suppose that, just before P_1 is about to execute a C&S, P_2 marks node X, and so P_1's C&S fails. When this happens, Harris's algorithms require P_1 to restart from the beginning of the list, which can lead to poor performance. Consider an execution E in a system of q processes. First insert n keys into the list. Then make one process P_q repeatedly delete the last node of the list, while the rest of the processes $P_1, \ldots, P_{q-1}$ attempt to insert new nodes at the end of the list. In each round of the execution, P_q marks a node right after processes $P_1, \ldots, P_{q-1}$ have located the correct insertion position, but before any of them perform a C&S. Each time $P_1, \ldots, P_{q-1}$ attempt to insert the keys at the end of the list, they have to search through the whole list to locate the appropriate insertion position, and therefore the total work done by the system is $\Omega(q \cdot (n + (n-1) + \ldots + 1)) = \Omega(qn^2)$. If we make $n > q$, then the average cost of an operation in this execution is $\Omega(qn) = \Omega(\bar{n}_E \bar{c}_E)$. (The variables $\bar{n}_E$ and $\bar{c}_E$ were defined in the introduction.)

Our implementation achieves better worst-case performance by making processes recover from failures instead of restarting. We augment each node of our data structure with an additional pointer field called *backlink*. When a node X gets deleted, its backlink is set to X's predecessor. If some process P then fails a C&S because X is marked, P follows X's backlink to X's predecessor. If the predecessor is also marked, P follows the predecessor's backlink, and so on, until it reaches an unmarked node U. Then P resumes its operation from U rather than from the beginning of the list. The sequence of backlinks that P traverses before reaching U is called a *chain of backlinks*. The introduction of backlinks alone, however, does not guarantee the desired operation complexity. The problem is that long chains of backlinks can be traversed by the same process many times. This happens when these chains *grow towards the right*, i.e. when backlink pointers are set to marked nodes, and thus nodes are linked to the right end of the chains. We eliminate this possibility by introducing *flag bits*.

The flag bit can be thought of as a warning that a deletion of the next node is in progress. Like the mark bit, the flag bit is part of the successor field, and is initially set to 0. When a node is *flagged* (i.e. when its flag bit is set to 1), its successor field is fixed and cannot be marked or otherwise changed until the flag is removed. Also, a marked node can never get flagged, and therefore no node can be both flagged and marked. Before marking a node B, a process flags the predecessor node A, thus ensuring that when B's backlink is set to point to A, it will not be pointing to a marked node. Figure 2 illustrates how deletions are performed in our data structure. Shaded boxes denote flagged successor fields, and crossed boxes denote marked successor fields. The deletion of node B consists of three steps. (1) Flagging the predecessor node A by applying C&S to its successor field (Figure 2, Step 1). (2) Setting B's backlink to point to its predecessor A and then marking B by applying C&S to its successor field (Figure 2, Step 2). (3) Performing a physical deletion of node B and removing A's flag by applying C&S to A's successor field (Figure 2, Step 3).

To preserve the lock-freedom property, we allow processes to help one another with deletions. For example, if a process

[1] In many modern architectures, a 32-bit word that stores a pointer has two unused bits. One of those can be used to store the mark bit and the other can be used to store the flag bit that we introduce later.

```
Search (Key k) : Node
// Searches for a node with the supplied key.
1  (curr_node, next_node) = SearchFrom(k, head)
2  if (curr_node.key == k)
3      return curr_node
4  else
5      return NO_SUCH_KEY

SearchFrom (Key k, Node *curr_node) : (Node, Node)
// Finds two consecutive nodes n1 and n2
// such that $n1.key \le k < n2.key$.
1  next_node = curr_node.right
2  while (next_node.key ≤ k)
       // Ensure that either next_node is unmarked,
       // or both curr_node and next_node are
       // marked and curr_node was marked earlier.
3      while (next_node.mark == 1 and
               (curr_node.mark == 0 or
               curr_node.right ≠ next_node))
4          if (curr_node.right == next_node)
5              HelpMarked(curr_node, next_node)
6          next_node = curr_node.right
7      if (next_node.key ≤ k)
8          curr_node = next_node
9          next_node = curr_node.right
10 return (curr_node, next_node)

HelpMarked (Node *prev_node, Node *del_node)
// Attempts to physically delete the marked
// node del_node and unflag prev_node.
1  next_node = del_node.right
2  c&s(prev_node.succ, (del_node, 0, 1), (next_node, 0, 0))
```

Figure 3: Search, SearchFrom, and HelpMarked.

cannot complete its operation because of a flagged node, it will try to complete the corresponding deletion, thus removing the flag, and then continue with its own operation.

3.2 Algorithms

The nodes in our linked list are ordered by their keys, and for simplicity our data structure does not allow users to insert duplicate keys. Each node has the following fields: key, element, backlink, and successor. The successor field is denoted *succ* in our pseudocode, and it is composed of three parts: a *right* pointer, a *mark* bit, and a *flag* bit. So, for each node n, $n.succ = (n.right, n.mark, n.flag)$. The head node and the tail node of the list contain dummy keys $-\infty$ and $+\infty$, and are referenced by the shared variables *head* and *tail* respectively. The pseudocode for our algorithms is shown in Figures 3 to 5. The routines Search, Insert, and Delete implement the corresponding dictionary operations.

The SearchFrom routine is used to perform searches in our data structure. It traverses the list starting from the specified node, and returns pointers to two nodes $n1$ and $n2$, that satisfy the following condition at some time during the execution of SearchFrom: $n1.right = n2$ and $n1.key \le k < n2.key$. SearchFrom also deletes any marked nodes that it sees by calling the HelpMarked routine (line 5). We could also write a SearchFrom2 routine, identical to the SearchFrom, except that "≤" in lines 2 and 7 would be replaced with "<". In our pseudocode, we use SearchFrom($k - \epsilon$, n) to denote SearchFrom2(k, n). The two nodes that SearchFrom($k - \epsilon$, *head*) returns satisfy $n1.key < k \le n2.key$ (and $n1.right = n2$).

```
Delete (Key k) : Node
// Attempts to delete a node with the supplied key.
1  (prev_node, del_node) = SearchFrom(k − ε, head)
2  if (del_node.key ≠ k)        // k is not found in the list.
3      return NO_SUCH_KEY
4  (prev_node, result) = TryFlag(prev_node, del_node)
5  if (prev_node ≠ null)
6      HelpFlagged(prev_node, del_node)
7  if (result == false)
8      return NO_SUCH_KEY
9  return del_node

HelpFlagged (Node *prev_node, Node *del_node)
// Attempts to mark and physically delete node del_node,
// which is the successor of the flagged node prev_node.
1  del_node.backlink = prev_node
2  if (del_node.mark == 0)
3      TryMark(del_node)
4  HelpMarked(prev_node, del_node)

TryMark (Node del_node)
// Attempts to mark the node del_node.
1  repeat
2      next_node = del_node.right
3      result = c&s(del_node.succ, (next_node, 0, 0),
                                   (next_node, 1, 0))
4      if (result == (∗, 0, 1))     // failure due to flagging
5          HelpFlagged(del_node, result.right)
6  until (del_node.mark == 1)
```

Figure 4: Delete, HelpFlagged, and TryMark.

The Search(k) routine simply uses SearchFrom to find the node with key k in the list, if it exists. The Insert routine starts by calling SearchFrom to find where to insert the new key. Then it verifies that the new key is not a duplicate, creates a new node, and enters the loop in lines 5–22, from which it can exit only if it successfully inserts the new node or another process inserts a node with the same key (lines 20–22). In each iteration of the loop, it attempts to insert the new node between *prev_node* and *next_node* by performing a C&S in line 11. If the C&S fails, Insert detects the reason, recovers from the failure, and enters the next iteration. The reason for the failure can only be the change of *prev_node*'s successor field. There are several possible ways in which this successor field can change: it can get redirected to another node, flagged, marked, or any two of the above, except that it cannot be both marked and flagged. If *prev_node* got flagged, it means that another process was performing a deletion of the successor node. In this case Insert calls the HelpFlagged routine (lines 15-16), which helps to complete that deletion and remove the flag from *prev_node*. If *prev_node* got marked, Insert traverses the backlinks until it finds an unmarked node and then sets *prev_node* to point to it (lines 17-18). In any case, in line 19 Insert invokes SearchFrom starting from *prev_node* to find the correct location for the insertion in the updated list, and updates its *prev_node* and *next_node* pointers. Then Insert enters the next iteration of the loop.

```
TryFlag (Node *prev_node, Node *target_node) : (Node, Boolean)
// Attempts to flag the predecessor of target_node. Prev_node is the last node known to be the predecessor.
while (true)
    if (prev_node.succ == (target_node, 0, 1))          // Predecessor is already flagged. Report
        return (prev_node, false)                        // the failure, return a pointer to prev_node.
    result = c&s(prev_node.succ, (target_node, 0, 0), (target_node, 0, 1))        // Flagging attempt
    if (result == (target_node, 0, 0))                   // Successful flagging. Report the success,
        return (prev_node, true)                         // return a pointer to prev_node.
    if (result == (target_node, 0, 1))                          // Failure due to flagging by a concurrent operation.
        return (prev_node, false)                               // Report the failure, return a pointer to prev_node.
    while (prev_node.mark == 1)                                 // Possibly a failure due to marking. Traverse
        prev_node = prev_node.backlink                          // a chain of backlinks to reach an unmarked node.
    (prev_node, del_node) = SearchFrom(target_node.key − $\epsilon$, prev_node)
    if (del_node $\neq$ target_node)                            // target_node got deleted.
        return (null, false)                                    // Report the failure, return no pointer.
```

```
Insert (Key k, Element e) : Node
// Attempts to insert a new node with the supplied key.
(prev_node, next_node) = SearchFrom(k, head)            // prev_node.key $\leq$ k $<$ next_node.key
if (prev_node.key == k)
    return DUPLICATE_KEY
newNode = new Node(key = k, element = e)
while (true)
    prev_succ = prev_node.succ
    if (prev_succ.flag == 1)                             // If the predecessor is flagged, help
        HelpFlagged(prev_node, prev_succ.right)          // the corresponding deletion to complete.
    else
        newNode.succ = (next_node, 0, 0)
        result = c&s(prev_node.succ, (next_node, 0, 0), (newNode, 0, 0))       // Insertion attempt.
        if (result == (next_node, 0, 0))                                        // Successful insertion.
            return newNode
        else                                                                    // Failure.
            if (result == (*, 0, 1))                     // Failure due to flagging.
                HelpFlagged(prev_node, result.right)     // Help complete the corresponding deletion.
            while (prev_node.mark == 1)                  // Possibly a failure due to marking. Traverse a
                prev_node = prev_node.back_link          // chain of backlinks to reach an unmarked node.
    (prev_node, next_node) = SearchFrom(k, prev_node)    // prev_node.key $\leq$ k $<$ next_node.key
    if (prev_node.key == k)
        free newNode
        return DUPLICATE_KEY
```

Figure 5: TryFlagand Insert.

The Delete routine performs a three-step deletion of the node, as discussed in Section 3.1. Delete starts by calling SearchFrom, and then calls TryFlag to perform the first deletion step (flagging the predecessor). TryFlag repeatedly attempts to flag *del_node*'s predecessor, until the flag is placed or *del_node* gets deleted. TryFlag returns two values: a node pointer *prev_node* and a boolean *result* value. There can be three ways the TryFlag routine can return. If TryFlag itself flags *del_node*'s predecessor, it returns a pointer to the predecessor and *result* = **true**. If TryFlag detects that another process flagged *del_node*'s predecessor (which means that another process is performing a deletion of *del_node*), it returns a pointer to the predecessor and *result* = **false**. If TryFlag detects that *del_node* got deleted from the list, it returns **null** and *result* = **false**. If *prev_node* returned by TryFlag is not **null**, Delete proceeds by calling the HelpFlagged routine, which performs the second and the third deletion steps by calling TryMark and HelpMarked. If TryFlag also returned *result* = **true**, Delete returns a pointer to the deleted node in line 9 (i.e. reports success). If *result* = **false**, it means that either *del_node* got deleted, or another process flagged *del_node*'s predecessor (and is going to report success). In this case Delete returns NO_SUCH_KEY.

3.3 Correctness

We will now present a sketch of the proof of correctness. The complete proof is available in [1]. We first prove several invariants. To state these invariants we classify the nodes into three categories as follows.

Def 1. *A node is* regular *if it is was inserted into the list, and it is unmarked.*

Def 2. *A node is* logically deleted *if it is marked and has a regular node linked to it, i.e. n is logically deleted if $n.mark = 1$ and there exists a regular node m such that $m.right = n$.*

DEF 3. *A node is* physically deleted *if it is marked and there is no regular node linked to it.*

At any time, each node that was ever inserted into the list fits into exactly one of these three categories. We prove that the following invariants apply to all regular, logically deleted, and physically deleted nodes of the list.

INV 1. *Keys are strictly sorted: for any two nodes $n1$, $n2$, if $n1.right = n2$, then $n1.key < n2.key$.*

INV 2. *The union of regular and logically deleted nodes forms a linked list structure, i.e. if n is a regular or a logically deleted node and $n \neq head$, then there is exactly one regular or logically deleted node m such that $m.right = n$. Node m is called n's* predecessor. *If $n \neq tail$, then node $n.right$ is regular or logically deleted, and it is called n's* successor. *The head node has no predecessor, and the tail node has no successor.*

INV 3. *For any logically deleted node, its predecessor is flagged (and unmarked), and its successor is not marked, i.e. if n is logically deleted, and m is a node of the list such that m is not physically deleted and $m.right = n$, then $m.succ = (n, 0, 1)$ and $(n.right).mark = 0$.*

INV 4. *For any logically deleted node, its backlink is pointing to its predecessor, i.e if n is logically deleted, and m is a node of the list such that m is not physically deleted and $m.right = n$, then $n.backlink = m$.*

INV 5. *No node can be both marked and flagged at the same time.*

It follows from Inv 3, that if two marked nodes are adjacent, then at least one of them is physically deleted.

The proof of the invariants goes as follows. Inv 5 is trivial. Inv 1–3 are proved by induction on the number of successful C&S's. This proof is lengthy, but fairly straightforward. After this we use the proved invariants to show that once a node's backlink is set, it never changes. This fact is used to prove Inv 4 by induction on the number of successful C&S's. We then prove two important properties of our algorithms. First, we show that deletions in our data structure work as intended, i.e. they are performed in three steps: first flagging the predecessor, then marking the node, and finally physically deleting the node. The second proposition states SEARCHFROM postconditions: if SEARCHFROM(k, n) returns $(n1, n2)$ and if $n.key \leq k$, then (1) $n1.key \leq k < n2.key$, (2) there exists a time during the execution of SEARCHFROM when $n1.right = n2$, and (3) if n is unmarked at some time T' before SEARCHFROM is invoked, then there exists time T between T' and the moment SEARCHFROM returns, when $n1$ is unmarked and $n1.right = n2$.

Finally, we use all these facts to prove the correctness of our implementation. At any time, we say that the set of elements currently stored in the dictionary is the set of the elements contained in the regular nodes, and we show that all operations can be linearized so that their return values are consistent with this definition. Specifically,

- The searches are linearized at time T specified by postcondition (3) of the SEARCHFROM routine they invoke. If the search is successful, the node it returns is a regular node at time T; if the search is unsuccessful there are no regular nodes with key k in the list at T.
- Each successful insertion is linearized when it successfully performs a C&S (line 11 in the INSERT routine) that inserts the node created in line 4. Each unsuccessful insertion is linearized at time T when the third postcondition holds for the last SEARCHFROM routine it invokes (line 1 or 19 in INSERT routine). At that time there is a regular node with the same key in the list.
- We linearize a successful deletion when the node it returns becomes marked (and therefore logically deleted). Unsuccessful deletions are linearized as follows. If the SEARCHFROM called by DELETE in line 2 found no node with key k, linearize the deletion at the time T specified by postcondition (3) for that SEARCHFROM. If the TRYFLAG called by DELETE returned in line 3, 8, or 13 (which means that another process was executing a concurrent deletion of the same node, and performed at least the first step of the deletion — flagging the predecessor), then we linearize the deletion immediately after *del_node* gets marked. Note that lines 5–6 of DELETE ensure that *del_node* gets marked (and then physically deleted) before DELETE returns in line 8, so this linearization is valid. Also note that the concurrent deletion that flagged *del_node*'s predecessor reports success when it returns.

3.4 Performance Analysis

Here we present a sketch of the amortized analysis of our linked list data structure. We start by explaining our billing scheme, first giving a general intuition behind it, and then defining it formally using the mapping β in Def 4. We then explain how we use this billing scheme to prove the bound on the amortized cost of operations. The full version of our amortized analysis is available in [1].

It is not hard to show that in order to calculate the cost of our algorithms, it is only essential to calculate the number of C&S attempts, the number of backlink pointer traversals (line 10 in TRYFLAG and line 18 in INSERT), and the number of *next_node* and *curr_node* pointer updates by searches (lines 6 and 8 in SEARCHFROM respectively). Counting these steps gives an accurate picture of the required time (up to a constant factor), and therefore we ignore other steps in our amortized analysis. When, later on, we talk about steps taken by the processes, we mean one of these *essential steps*.

We classify the (essential) steps of each operation S into three categories: successful C&S's, *necessary steps*, and *extra steps*. The necessary steps are the (non-C&S) steps that S normally has to perform in order to complete (e.g. in order to complete a search for key k, S has to traverse all nodes with keys smaller than k). Intuitively, the necessary steps are the steps that an operation needs to perform even if it is executing on a sequential linked list. By contrast, the extra steps are the steps that S has to take because of interference from other operations (e.g. when S fails a C&S because of a change performed by a concurrent operation). The cost of the necessary steps of S is called the *necessary cost of S*, and the cost of the extra steps of S is called the *extra cost of S*. In our analysis, we show that the necessary cost of S is always $O(n(S))$ ($n(S)$ and $c(S)$ were defined in the introduction), and we use a mapping to bill all of the extra cost of S to successful C&S's that are *part of* operations concurrent with S. We say that a C&S is *part of operation S* if it is successful, and it logically belongs to that operation.

Specifically, each successful C&S that inserts a new node is part of the corresponding successful insertion, and successful C&S's that flag, mark, and physically delete nodes are part of the corresponding successful deletions. A (successful) C&S that is part of a given operation is not necessarily performed by the process that is executing this operation, because processes help one another with deletions.

We define the *amortized cost* of a successful C&S C, denoted $\hat{t}(C)$, to be $(\text{actual cost of } C) + (\text{total cost billed to } C)$. Note that the first term is 1. We define the *amortized cost* of S, denoted $\hat{t}(S)$, to be $(\text{actual cost of } S) - (\text{total cost billed from } S \text{ to successful C\&S's}) + (\text{total cost billed to successful C\&S's that are part of } S)$. The second term is the extra cost of S, so

$$\begin{aligned}\hat{t}(S) &= ((\text{necessary cost of } S) + (\text{extra cost of } S) + \\ &\quad (\text{cost of successful C\&S's performed by } S)) - \\ &\quad (\text{extra cost of } S) + (\text{total cost billed to} \\ &\quad \text{successful C\&S's that are part of } S) \\ &= (\text{necessary cost of } S) + (\text{cost of successful} \\ &\quad \text{C\&S's performed by } S) + (\text{total cost billed to} \\ &\quad \text{successful C\&S's that are part of } S) \\ &= (\text{necessary cost of } S) + (\text{amortized cost of} \\ &\quad \text{successful C\&S's that are part of } S).\end{aligned}$$

We prove that the first term is $O(n(S))$ and that, for any C&S C that is part of operation S, the total cost billed to C is $O(c(S))$. Since at most three C&S's can be part of any given operation, we conclude that the second term is $O(c(S))$. Therefore, $\hat{t}(S) = O(n(S) + c(S))$. Note that here the $O(n(S))$ term comes purely from the cost of the steps that even a sequential algorithm needs to perform, while the overhead that comes from concurrency is limited by an additive term of $O(c(S))$. We now describe all of the steps outlined above in more detail.

To define our billing scheme formally, we introduce a *mapping function* β, given below. This mapping also formally defines the set of the extra steps and the set of the necessary steps for every operation. Function β will map successful C&S's to themselves. All other steps mapped to themselves are necessary steps. The remaining steps are extra steps. The logic behind the design of this mapping function is that each extra step is mapped to the successful C&S that performed the change that causes this extra step to be taken. For example, the step of traversing node n that was inserted after S was invoked is mapped to the C&S that inserted n. To make it easier to define β, we categorize C&S's performed by our algorithms into four types: (1) insertion C&S (line 11 in INSERT), (2) flagging C&S (line 4 in TRYFLAG), (3) marking C&S (line 3 in TRYMARK), and (4) physical deletion C&S (line 2 in HELPMARKED).

DEF 4. *Let Q be the set of essential steps in the entire execution E. Function β maps Q to itself. If some operation S performs step $s \in Q$, β maps this step either to itself, or to a successful C&S that is part of another operation as described below.*

- ***C&S's:*** *Suppose a C&S C on the successor field of node n was executed. If C is successful, then we map it to itself. If C fails, and it is not of the fourth type, we map it to the C&S that last modified $n.succ$. If C is of the fourth type and it fails, we map it to the C&S that physically deleted the node that C was trying to delete. (We show that such a C&S had to be performed.)*

- ***Backlink traversals:*** *A backlink pointer traversal from node n to node m is mapped to the C&S that marked node n.*

- ***Next_node pointer updates:*** *Suppose the update changes next_node from m to m'. If m is physically deleted before the update, we map the update to the C&S that physically deleted m. (Note that even though this C&S could be performed by* HELPMARKED *called from this* SEARCHFROM *routine in line 5, it is part of another operation.) Otherwise we map the update to the C&S that inserted m'.*

- ***Curr_node pointer updates:*** *Suppose the update sets curr_node pointer to node n. If n was inserted into the list after operation S was invoked, then the update is mapped to the C&S that inserted n. Otherwise, the update is mapped to itself.*

To prove our bound on the amortized cost of operations, we need to show that the amortized cost of each C&S that is part of an operation S, is $O(c(S))$. This is the most important and the most technical part of our amortized analysis. Below we briefly describe this proof.

There are four types of steps that Def 4 bills to successful C&S's. For each of them we prove that if a step of that type performed by operation S' is mapped by β to a (successful) C&S C, then (1) no other steps of the same type performed by S' are mapped to C, and (2) C was performed during the execution of S'. It then follows that no more than $c(S)$ steps of each type can be mapped to C, where S is the operation C is part of. Proving (1) and (2) for *next_node* updates is fairly straightforward. For *curr_node* pointer updates, we first show that no operation can set *curr_node* pointer (in line 8 of a SEARCHFROM) to a given node more than once, and then (1) and (2) follow. For backlink traversals, we show that if operation S traverses a backlink from node n, then n got marked during S, and S never traversed a backlink from n before, which leads to (1) and (2). In this part of the proof we rely on the fact that chains of backlinks never grow towards the right (see Section 3.1). For unsuccessful C&S's, we prove two lemmas. The first one states that if C' is an unsuccessful C&S of type four on the successor field of node n performed by operation S', then there exists a time T during S' when $n.succ$ was such that C' would have succeeded, and S' performed no C&S's on $n.succ$ between T and C'. The second lemma states a similar, but slightly weaker claim for the C&S's of the first three types. Using these two lemmas, we show that (1) and (2) hold for unsuccessful C&S's as well.

Since no more than $c(S)$ steps of each type can be mapped by β to a successful C&S that is part of S, it follows that the amortized cost of a successful C&S is $O(c(S))$. Since at most three successful C&S's can be part of S, it follows that the amortized cost of successful C&S's that are part of S is $O(c(S))$. To prove that the amortized cost of S is $O(n(S)+c(S))$ we now only need to show that the total cost of the steps of S that are not mapped by β to the successful C&S's (i.e. the necessary cost of S) is $O(n(S))$.

First, note that the only steps of S that are not mapped to the successful C&S's are the *curr_node* pointer updates in line 8 of SearchFrom routines called by S. Furthermore, by the definition of β such an update is mapped to itself (and not to a successful C&S) only if the node n to which the *curr_node* pointer is set to by this update is inserted before the invocation of S. It is also not hard to show that n must be unmarked at some moment during the execution of S, which means that n is a regular node when S is invoked (since nodes never get unmarked). Also, as mentioned above, no operation can set the *curr_node* pointer (in line 8 of a SearchFrom routine) to a given node more than once. Consequently, the total number of steps of S that are not mapped to the successful C&S's cannot be greater than the number of regular nodes when S is invoked, i.e. $n(S)$. This concludes our amortized analysis, yielding $\hat{t}(S) = O(n(S)) + O(c(S))$ (where the $O(n(S))$ term comes from the necessary cost of S, and the $O(c(S))$ term comes from the concurrency overhead).

4. SKIP LISTS

In this section we briefly discuss our lock-free implementation of a skip list data structure and give a sketch of the proof of correctness. The algorithms and the complete proof of correctness are available in [1].

A skip list [12] is a sequential dictionary data structure, in which searches, insertions, and deletions have an expected cost of $O(\log(n))$ (and worst-case cost of $O(n)$), where n is the number of elements in the dictionary. The expectation is taken over the random numbers generated inside the algorithms. Our lock-free skip list architecture has some differences from Pugh's original design to make it easier to reuse our linked list algorithms. As shown in Figure 6, we represent each key by a *tower* of nodes. A tower that has H nodes in it is said to have *height* H. The height of each tower is chosen randomly by coin flips. The bottom node of a tower is called the *root node*, and it acts as a representative of the whole tower. The *head tower* and the *tail tower* store dummy keys $-\infty$ and $+\infty$ respectively. Horizontally, the nodes of the skip list are arranged in *levels*: the root nodes are on level one, the nodes immediately above them are on level two, and so on. Nodes of the same level form a singly-linked list, sorted according to their keys.

In the original skip list design [12], Pugh uses a single node with an array of H forward pointers to represent a tower of height H. The difference between our architecture and Pugh's architecture is not very significant, but it makes it easier to explain our algorithms in terms of the linked list algorithms already described. For convenience, we use the same terminology when we compare our skip list implementation with others [2, 15], even though they use Pugh's architecture.

Other recent lock-free skip list designs [2, 15] implement individual levels using linked list algorithms that can exhibit bad worst-case behaviour, as described in Section 3.1. (Furthermore, although Sundell and Tsigas incorporate backlinks in their implementation, a backlink is not guaranteed to be set when it is needed, and their backlink is useful on a given level only if the tower it is pointing to is sufficiently high.) Because of the randomization used by the algorithm, it is unclear whether an adversary could exploit the worst-case behaviour on individual levels to force the skip list as a whole to experience bad worst-case behaviour. Our design was driven by an effort to ensure that individual levels of the skip list have good worst-case complexity by using our new linked list algorithms, so that a tight analysis of the average expected complexity of the skip list operations would be feasible. However, new difficulties arise when attempting to do this, as explained in more detail below. Thus, the problem of proving a good upper bound on the complexity of a lock-free skip list implementation remains open.

In our data structure, a node Q that is not a node of the head or tail tower has the following fields: *key*, *backlink*, *succ*, *down*, and *tower_root*. The first three fields are the same as in our lock-free linked lists, *down* is a pointer to the node one level lower than Q (or **null** if Q is a root node), and *tower_root* is a pointer to the root node of Q's tower. If Q is a root node, it also has an *element* field. Nodes of the head tower do not have elements, backlinks or tower_root pointers, but each of them has an *up* pointer, pointing to the node above. The top node of the head tower has its up pointer set to itself. Nodes of the tail tower contain only the key $+\infty$. A pointer to the bottom node of the head tower is referred to by a shared variable *head*.

We now give a high-level overview of our algorithms. An insertion builds the tower from bottom to top, i.e. first it inserts the root node, then, if necessary, the node at level two, and so on. An insertion is linearized when the root node is inserted, since after that moment, all the searches are able to find the key. A deletion first deletes the root node of the tower, and then deletes the rest of the nodes of the tower from top to bottom. A deletion is linearized when the root node gets marked. A tower whose root node is marked is called *superfluous*; all the nodes of such a tower are called *superfluous* as well.

Regardless of whether deletions delete the towers from top to bottom, or from bottom to top, superfluous nodes can still exist, because while a process P is constructing a tower Q, Q's root node can get marked by another process, and P can add a new node to Q before it notices the marking. It is possible to solve this problem by marking uninserted nodes if Harris's design is used to implement individual levels of the skip list [2], but with our design this is not feasible because of flags.

The searches in our skip list help deletions by physically deleting superfluous nodes they encounter in order to avoid traversing superfluous towers. Our decision to implement searches this way was motivated by the observation that if searches traverse superfluous towers without physically deleting or marking their nodes, it is possible to construct an execution E where the average cost of operations would be $\Omega(m_E)$ by forcing operations to repeatedly traverse a chain of backlinks of length $\Omega(m_E)$ on the lowest level of the skip list (m_E was defined in the Introduction). Sundell and Tsigas [15] use a different method to deal with this problem: their searches can enter superfluous towers via unmarked nodes, but if a search detects a marked node in a tower it is traversing, it marks all the nodes of this tower. Subsequent searches physically delete these marked nodes if they encounter them (assuming the main Delete operation has not already done so), thus making numerous traversals of the same chain of backlinks impossible.

Even though searches in our implementation delete superfluous nodes whenever they encounter them, and therefore they cannot be forced so to traverse the same chain of backlinks repeatedly, there exist scenarios when an operation can be forced to traverse backlinks of the nodes that were deleted before the operation started (something that never happens

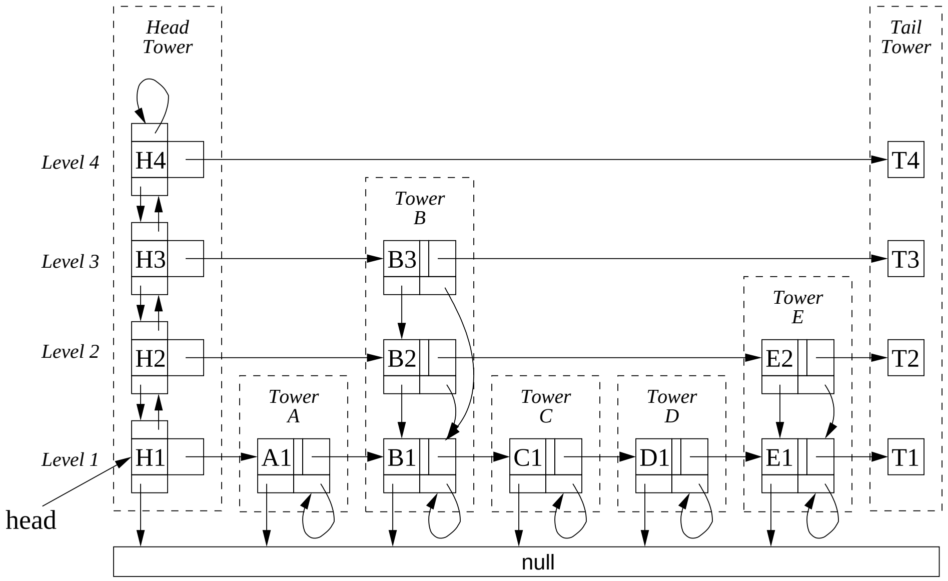

Figure 6: Lock-free skip list design.

in our linked list implementation). These scenarios can only be constructed by a very careful scheduling of processes tailored for a given distribution of the heights of the towers. Their existence, however, makes our correctness proofs quite complicated, but more importantly, it is not clear what effect they may have on the worst-case performance of our implementation.

The pseudocode for the skip list algorithms is available in [1], and here we describe them only briefly. Each level of the skip list can be viewed as a linked list. Therefore, the routines that we use to operate on the individual levels are similar to our linked list routines. The three major routines that implement the dictionary operations are Search_SL, Insert_SL, and Delete_SL. The Search_SL routine calls SearchToLevel_SL to determine if there is a root node (and hence, a tower) with key k in the list. SearchToLevel_SL(k, v) is used to locate the nodes on level v with keys closest to k. It traverses levels starting from the top one, and each time it reaches a key larger than k, it goes down one level. To traverse individual levels, it uses the SearchRight routine, which is similar to the SearchFrom in our linked list algorithms. The only difference is that SearchRight deletes the superfluous nodes along its way, performing all three deletion steps if necessary, whereas SearchFrom physically deletes only those nodes that are already logically deleted.

The Insert_SL routine determines the height of the tower it needs to insert by flipping a coin, and enters a loop where it inserts the nodes of the tower one by one from bottom to top. If a concurrent process inserts a root node with the same key, Insert_SL reports failure and returns. Each complete iteration of the loop increases the height of the new tower by one. Insert_SL exits from that loop if it finishes the construction of the new tower, or if the construction of a new tower gets *interrupted* by a deletion: if Insert_SL notices that the root node got marked, it exits reporting success. The Delete_SL routine first deletes the root node of the tower with the supplied key k, making the rest of the nodes in the tower superfluous. It then calls SearchToLevel_SL for k, which deletes these superfluous nodes (from top to bottom).

We now briefly sketch the proof of correctness. The first part of the proof is similar to the correctness proof for the linked lists. We show that Inv 1–5 (see Section 3.3) hold for each level of our skip list, and that nodes never change levels. We also show that deletions of individual nodes are performed in three steps (flag, mark, physical deletion), and that the same postconditions that hold for SearchFrom hold for SearchRight as well. These postconditions guarantee that if SearchRight starts from a node n that is not superfluous at time T', then the node m it ends in is not marked at T'. However, m may be superfluous at T', but we show that this can happen only if SearchRight enters m by traversing backlinks, and in this case $m.key < n.key$. The fact that SearchRight may traverse superfluous nodes leads to the fact that SearchToLevel_SL may enter marked nodes when it descends from one level to the next (although scenarios where this happens are fairly contrived). This is why an operation can traverse backlinks of the nodes that were deleted before the operation started. As mentioned earlier, this is an obstacle to applying the same kind of performance analysis to skip lists, as we used for linked lists. After proving some weaker postconditions for SearchToLevel_SL and Insert_SL, we then show that our skip list has the correct vertical structure within each tower, i.e. the nodes on different levels that contain the same key form a linked list. Then we prove the stronger SearchToLevel_SL(k, v) postconditions: we show that the node n it ends in is unmarked, and, if $n.key = k$, n is also not superfluous (at some time during the search).

Finally, we say that the set of elements currently stored in the dictionary is the set of the elements of the regular root nodes, and we show that all operations can be linearized consistently with this definition. We prove that our implementation is lock-free by showing that the only way a process's operation can be delayed indefinitely is if other processes continually perform successful C&S's.

We also investigate the distribution of the heights of the towers in our skip list. We call a tower *full* if its insertion has finished without an interruption; otherwise we say that a tower is *incomplete*. A non-deleted tower can be incomplete only if its insertion or its deletion is in progress, so the number of incomplete towers at any time is bounded by the point contention. The distribution of the heights of the full towers may be a little different from the heights distribution in a sequential skip list, because higher towers are more likely to be incomplete. However, we believe this would not affect the expected running time significantly.

5. CONCLUSION

We have presented new algorithms implementing lock-free linked lists. We proved that the average cost of operations on our linked lists is linear in the length of the list plus the contention, for any possible sequence of operations and any possible scheduling. To perform our analysis we used a billing technique that might be applicable to other distributed data structures. We showed that our linked list algorithms can be used in a fairly modular way as the basis for a lock-free implementation of skip lists.

We have not explicitly incorporated a memory management technique, but a possible approach is to use Valois's reference counting method [10, 17], which is applicable to both our linked lists and our skip lists, because there are no cycles among the physically deleted nodes.

There are a number of directions for future work in this area. It remains an open problem to get a good bound on the average expected complexity of lock-free implementations of a skip list (or, more generally, a dictionary data structure). We think the implementation given here and the amortized analysis technique may be useful in doing this. However some difficulties remain. For example, an adversary might choose to delete all of the tall towers that are used to traverse the skip list quickly. Although an oblivious adversary (who cannot see the outcomes of coin flips) cannot directly know the heights of the towers, in a distributed application it might indirectly get some information about them by seeing how many steps are required to do searches. It might be more realistic to separate the two roles of the adversary: choosing the operations and choosing the schedule.

On a more general note, it would be interesting to develop a usable and practical alternative to the worst-case amortized analysis, which can be overly pessimistic, in the context of lock-free data structures. A feasible way of doing an amortized analysis that bounds the average complexity over possible schedules would be of great interest.

Acknowledgements

This research was funded by the Natural Sciences and Engineering Research Council of Canada and by an Ontario Graduate Scholarship. We thank Håkan Sundell, Philippas Tsigas and the anonymous referees for pointing out related work and providing helpful comments.

6. REFERENCES

[1] M. Fomitchev. Lock-free linked lists and skip lists. Master's thesis, York University, October 2003. http://www.cs.yorku.ca/∼mikhail.

[2] K. A. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, December 2003. Technical Report UCAM-CL-TR-579.

[3] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Symposium on Distributed Computing*, pages 300–314, 2001.

[4] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[5] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

[6] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[7] IBM System/370 extended architecture, principles of operation., 1983. IBM Publication No. SA22-7085.

[8] 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*, pages 73–82, 2002.

[9] M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, 2002.

[10] M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, Computer Science Department, University of Rochester, 1995.

[11] W. Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222, Computer Science Department, University of Maryland, 1990.

[12] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of ACM*, 33(6):668–676, 1990.

[13] N. Shavit and I. Lotan. Skiplist-based concurrent priority queues. In *Proc. 14th IEEE/ACM International Parallel and Distributed Processing Symposium*, pages 263–268, 2000.

[14] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Proceedings of the 17th IEEE/ACM International Parallel and Distributed Processing Symposium*, pages 84–94, April 2003.

[15] H. Sundell and P. Tsigas. Scalable and lock-free concurrent dictionaries. In *Proceedings of the 19th ACM Symposium on Applied Computing*, pages 1438–1445, March 2004.

[16] R. K. Treiber. Systems programming: Coping with parallelism. Research report RJ 5118, IBM Almaden Research Center, 1986.

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