A Pragmatic Implementation of Non-Blocking Linked-Lists

Timothy L. Harris

University of Cambridge Computer Laboratory,
Cambridge, UK, tim.harris@cl.cam.ac.uk

Abstract. We present a new non-blocking implementation of concurrent linked-lists supporting linearizable insertion and deletion operations. The new algorithm provides substantial benefits over previous schemes: it is conceptually simpler and our prototype operates substantially faster.

1 Introduction

It is becoming evident that non-blocking algorithms can deliver significant benefits to parallel systems [MP91,LaM94,GC96,ABP98,Gre99]. Such algorithms use low-level atomic primitives such as compare-and-swap – through careful design and by eschewing the use of locks it is possible to build systems which scale to highly-parallel environments and which are resilient to scheduling decisions.

Linked-lists are one of the most basic data structures used in program design, and so a simple and effective non-blocking linked-list implementation could serve as the basis for many data structures. This paper presents a novel implementation of linked-lists which is non-blocking, linearizable and which is based on the the compare-and-swap (CAS) operation found on contemporary processors.

Section 5 sketches a proof of correctness, describes the use of model-checking to perform exhaustive verification within a limited application domain and also describes empirical tests performed on execution traces from an actual implementation.

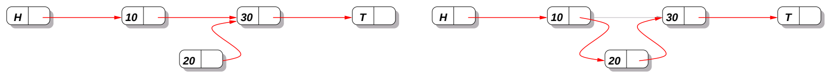
In Sect. 6 we compare the performance of the new algorithm against that of a lock-based implementation and against an existing non-blocking algorithm. Compared with these other thread-safe algorithms, ours provides the best performance on each of three simulated workloads and for every level of concurrency.

2 Overview

In this section we present an overview of our algorithm and the difficulty in implementing non-blocking linked-lists. As a running example consider an ordered list containing the integers 10 and 30 along with sentinel *head* and *tail* nodes:

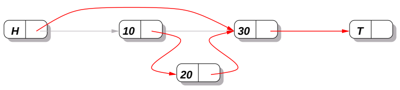
Such a data structure may comprise cells containing two fields: a `key` field used to store the element and a `next` field to contain a reference to the next cell in the list.

Insertion is straightforward: a new list cell is created (*below, left*) and then introduced using single CAS operation on the `next` field of the proposed predecessor (*below, right*).

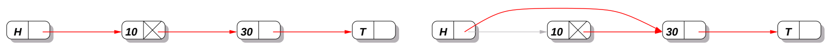


In this case the atomicity of the CAS ensures the the nodes either side of the insertion have remained adjacent. This simple guarantee is insufficient for deletions within the list. Suppose that we wish to remove the value 10. An obvious way of excising this node would be to perform a CAS that swings the reference from the head so that the node containing 30 becomes the first in the list:

Although this CAS ensures that the node 10 was still at the start of the list it cannot ensure that no additional nodes were introduced between the 10 node and the 30 node. If this deletion took place concurrently with the previous insertion then that new node would be lost:



The single CAS could neither detect nor prevent changes between 10 and 30 once the deletion procedure had selected 30. Our proposed solution – and indeed the crux of the algorithms presented here – is to use two separate CAS operations in place of that single one. The first of these is used to *mark* the `next` field of the deleted node in some way (*below, left*), whereas the second is used to excise the node (*below, right*):



We say that a node is *logically deleted* after the first stage and that it is *physically deleted* after the second. A marked field may still be traversed but takes a numerically distinct value from its previous unmarked state; the structure of the list is retained while signalling concurrent insertions to avoid introducing new nodes immediately after those that are logically deleted. In our example the concurrent insertion of 20 would observe 10 to be logically deleted and would attempt to physically delete it before re-trying the insertion.

3 Related Work

Generalized non-blocking implementations based on CAS were presented by Herlihy [Her91,Her93]. However, linked-lists based on this general scheme are highly centralized and suffer poor performance because they essentially use CAS to change a shared global pointer from one version of the structure to the next.

Valois was the first to present an effective CAS-based non-blocking implementation of linked-lists [Val95]. Although highly distributed, his implementation is very involved. The list is held with *auxilliary cells* between adjacent pairs of ordinary cells. Auxilliary exist to provide an extra level of indirection so that a cell may be removed by joining together the auxilliary cells adjacent to it. Valois' algorithm exposes a more general and lower level interface than we do here; he provides explicit *cursors* to identify cells in the list and operations to insert or delete nodes at those points.

The originally-published algorithm contained a number of errors relating to how reference-counted storage was managed. One has been reported previously and others were identified when implementing Valois' algorithm for comparison in this paper [MS95,Val01].

To overcome the complexity of building linearizable lock-free linked-lists using CAS, Greenwald suggested a stronger double-compare-and-swap (DCAS) primitive that atomically updates two storage locations after confirming that they both contain required values [Gre99]. DCAS is not available on today's multi-processor architectures. However, it does admit a simple linearizable linked-list algorithm: insertions proceed as described in Sect. 2 and deletions by atomic updates to the `next` field of the cell being removed as well as that of its predecessor. Greenwald's work was an extension of earlier non-linearizable DCAS-based linked-list algorithms due to Massalin and Pu [MP91].

4 Algorithms

In this section we present our new algorithm in pseudo-code modeled on C++ and designed for execution on a conventional shared-memory multi-processor system supporting *read*, *write* and atomic *compare-and-swap* operations. We assume that the operations defined here are the only means of accessing linked list objects. Each processor executes a sequence of these operations, defining a *history* of invocations/responses and inducing a *real-time* order between them. We say that an operation A *precedes* B if the response to A occurs before the invocation of B and that operations are *concurrent* if they have no real-time ordering.

A *sequential* history is one in which each invocation is followed immediately by its corresponding response. Our basic correctness requirement is linearizability which requires that (a) the responses received in every concurrent history are equivalent to those of some legal sequential history of the same requests and (b) the ordering of operations within the sequential history is consistent with the real-time order [HW90]. Linearizability means that operations appear

```
class List<KeyType> {
  Node<KeyType> *head;
  Node<KeyType> *tail;

  List() {
    head = new Node<KeyType> ();
    tail = new Node<KeyType> ();
    head.next = tail;
  }
}
```

```
class Node<KeyType> {
  KeyType  key;
  Node     *next;

  Node (KeyType key) {
    this.key = key;
  }
}
```

Fig. 1. An instance of the List class contains two fields which identify the head and the tail. Instances of Node contain two fields identifying the key and successor of the node.

```
public boolean List::insert (KeyType key) {
  Node *new_node = new Node(key);
  Node *right_node, *left_node;

  do {
    right_node = search (key, &left_node);
    if ((right_node != tail) && (right_node.key == key)) /*T1*/
      return false;
    new_node.next = right_node;
    if (CAS (&(left_node.next), right_node, new_node)) /*C2*/
      return true;
  } while (true); /*B3*/
}
```

Fig. 2. The `List::insert` method attempts to insert a new node with the supplied key.

to take effect atomically at some point between their invocation and response. Our implementation is additionally *non-blocking*, meaning that some operation will complete in a finite number of steps, even if other operations halt.

We write `CAS(addr,o,n)` for a CAS operation that atomically compares the contents of `addr` against the old value `o` and – if they match – writes `n` to that location. CAS returns a boolean indicating whether this update took place. Our design was guided by the assumption that a CAS operation is slower to execute than a write which in turn is slower than a read.

4.1 Implementing Sets

Initially we will consider a set object supporting three operations: Insert(k), Delete(k), Find(k). Each parameter k is drawn from a set of totally-ordered keys. The result of an Insert, a Delete or a Find is a boolean indicating success or failure. The set is represented by an instance of `List` which contains a singly-linked list of instances of `Node`. As sketched in Sect. 2 these are held in ascending order with sentinel head and tail nodes.

```
public boolean List::delete (KeyType search_key) {
  Node *right_node, *right_node_next, *left_node;

  do {
    right_node = search (search_key, &left_node);
    if ((right_node == tail) || (right_node.key != search_key)) /*T1*/
      return false;
    right_node_next = right_node.next;
    if (!is_marked_reference(right_node_next))
      if (CAS (&(right_node.next), /*C3*/
          right_node_next, get_marked_reference (right_node_next)))
        break;
  } while (true); /*B4*/
  if (!CAS (&(left_node.next), right_node, right_node_next)) /*C4*/
    right_node = search (right_node.key, &left_node);
  return true;
}
```

Fig. 3. The `List::delete` method attempts to remove a node containing the supplied key.

```
public boolean List::find (KeyType search_key) {
  Node *right_node, *left_node;

  right_node = search (search_key, &left_node);
  if ((right_node == tail) ||
      (right_node.key != search_key))
    return false;
  else
    return true;
}
```

Fig. 4. The `List::find method tests whether the list contains a node with the supplied key.`

The reference contained in the next field of a node may be in one of two states: marked or unmarked. A node is marked if and only if its next field is marked. Marked references are distinct from normal references but still allow the referred-to node to be determined – for example they may be indicated by an otherwise-unused low-order bit in each reference. Intuitively a marked node is one which should be ignored because some process is deleting it. The function `is_marked_reference(r)` returns `true` if and only if `r` is a marked reference. Similarly `get_marked_reference(r)` and `get_unmarked_reference(r)` convert between marked and unmarked references.

The concurrent implementation comprises four methods (Fig. 2-5). The first three, `List::insert`, `List::delete` and `List::find` implement the Insert, Delete and Find operations respectively. The fourth, `List::search`, is used during each of these operations. It takes a search key and returns references to two nodes called the *left node* and *right node* for that key. The method ensures that these nodes satisfy a number of conditions. Firstly, the key of the left node must be less than the search key and the key of the right node must be greater than

```
private Node *List::search (KeyType search_key, Node **left_node) {
  Node *left_node_next, *right_node;

search_again:
  do {
    Node *t = head;
    Node *t_next = head.next;

    /* 1: Find left_node and right_node */
    do {
      if (!is_marked_reference(t_next)) {
        (*left_node) = t;
        left_node_next = t_next;
      }
      t = get_unmarked_reference(t_next);
      if (t == tail) break;
      t_next = t.next;
    } while (is_marked_reference(t_next) || (t.key<search_key)); /*B1*/
    right_node = t;

    /* 2: Check nodes are adjacent */
    if (left_node_next == right_node)
      if ((right_node != tail) && is_marked_reference(right_node.next))
        goto search_again; /*G1*/
      else
        return right_node; /*R1*/

    /* 3: Remove one or more marked nodes */
    if (CAS (&(left_node.next), left_node_next, right_node)) /*C1*/
      if ((right_node != tail) && is_marked_reference(right_node.next))
        goto search_again; /*G2*/
      else
        return right_node; /*R2*/
  } while (true); /*B2*/
}
```

Fig. 5. The `List::search` operation finds the left and right nodes for a particular search key.

or equal to the search key. Secondly, both nodes must be unmarked. Finally, the right node must be the immediate successor of the left node. This last condition requires the search operation to remove marked nodes from the list so that the left and right nodes are adjacent. As we will show the `List::search` method is implemented so that these conditions are satisfied concurrently at some point between the method's invocation and its completion.

`List::search` is divided into three sections. The first section iterates along the list to find the first unmarked node with a key greater than or equal to the search key. This is the right node. The left node preliminarily refers to the previous unmarked node that was found. The second stage examines these nodes. If `left_node` is the immediate predecessor of `right_node` then `List::search` returns. Otherwise, the third stage uses a CAS operation to remove marked nodes between `left_node` and `right_node`.

`List::insert` uses `List::search` to locate the pair of nodes between which the new node is to be inserted. The update itself takes place with a single CAS operation (`C2`) which swings the reference in `left_node.next` from `right_node` to the new node.

`List:delete` uses `List::search` to locate the node to delete and then uses a two-stage process to perform the deletion. Firstly, the node is logically deleted by marking the reference contained in `right_node.next` (`C3`). Secondly, the node is physically deleted. This may be performed directly (`C4`) or within a separate invocation of search.

The `List::find` method is shown in Fig. 4. It invokes `List::search` and examines the resulting right node.

5 Correctness

In this section we describe three approaches taken to checking the correctness of the algorithms presented here. Section 5.1 outlines a proof of linearizability and progress, Sect. 5.2 describes the exhaustive testing of some cases through model checking and Sect. 5.3 describes a method we used for examining traces from particular program runs.

5.1 Proof Sketch

We will take a fairly direct approach to outlining the linearizability of the operations by identifying particular instants during their execution at which the complete operation appears to occur atomically.

Conditions Maintained by Search. Our argument relies on the conditions identified in Sect. 4.1 which the implementation of `List::search` guarantees hold at some point during its invocation. For the ordering constraints, note that when right node is initialized the preceding loop ensured that `search_key` $\leq$ `right_node.key`. Similarly `left_node.key` $<$ `search_key` because otherwise the loop would have terminated earlier.

For the adjacency condition and the mark state of the left node we must separately consider each return path. If `List::search` returns at `R1` then the test guarding the return statement ensures that right node was the immediate successor of the left node when the `next` field of that node was read into the local variable `t_next`. The same value of `t_next` is found to be unmarked before initializing `left_node`. If `List::search` returns at `R2` then `C1` establishes the required conditions.

For the mark state of the right node, observe that both return paths confirm that the right node is unmarked after the point at which the first three conditions must be true. Nodes never become unmarked and so we may deduce that the right node was unmarked at that earlier point.

Linearization points. Let $\mathrm{op}_{i,m}$ be the m^{th} operation performed by processor i and let $d_{i,m}$ be the final real-time at which the `List::search` post-conditions are satisfied during its execution. These $d_{i,m}$ identify the times at which the outcome of the operations become inevitable and we shall take the ordering between them to define the linearized order of Find(k) operations or unsuccessful updates. For a successful find at $d_{i,m}$ the right node was unmarked and contained the search key. For an unsuccessful insertion it exhibits a node with a matching key. For an unsuccessful deletion or find it exhibits the left and right nodes which, respectively, have keys strictly less-than and strictly greater-than the search key.

Furthermore let $u_{i,m}$ be the real-time at which the update `C2` inserts a node or `C3` logically deletes a node. We shall take $u_{i,m}$ as the linearization points for such successful updates. In the case of a successful insertion the CAS at $u_{i,m}$ ensures that the left node is still unmarked and that the right node is still its successor. For a successful deletion the CAS at $u_{i,m}$ serves two purposes. Firstly, it ensures that the right node is still unmarked immediately before the update (that is, it has not been logically deleted by a preceding successful deletion). Secondly, the update itself marks the right node and therefore makes the deletion visible to other processors.

Progress. We will show that the concurrent implementation is non-blocking. We will show that each successful insertion causes exactly one update, that each successful deletion causes at most two updates and that unsuccessful operations do not cause any updates.

The CAS instructions `C1` and `C4` each succeed only by unlinking marked nodes from the list. Therefore the number of times that these CAS instructions succeed is bounded above by the number of nodes that have been marked. Exactly one node is marked during each successful deletion (`C3`) and therefore at most one update may be performed by `C1` or `C4` for each successful deletion. The remaining CAS instructions (`C2` and `C3`) occur respectively exactly once on the return paths from successful insertions and deletions.

Since there are no recursive or mutually-recursive method definitions consider each backward branch in turn:

- Each time `B1` is taken the local variable `t` is advanced once node down the list. The list is always contains the unmarked tail node and the nodes visited have successively strictly larger keys.
- Each time `B2` is taken the CAS at `C1` has failed and therefore the value of `left_node.next` $\neq$ `left_node_next`. The value of the field must have been modified since it was read during the loop ending at `B1`. Modifications are only made by successful CAS instructions and each operation causes at most two successful CAS instructions.
- Each time `B3` or `B4` is taken the CAS at `C2` or `C4` has failed. As before, the value held in that location must have been modified since it was read in `List::search` and at most two such updates may occur for each operation.

– Each time `G1` or `G2` is taken then a node which was previously unmarked has been marked by another processor. As before, at most two updates may occur for each operation.

5.2 Model Checking

The dSPIN model checker was used to exhaustively verify the operations for certain problem domains. dSPIN is an extension of the SPIN model checker with adds support for pointers, storage management, function calls and local scopes [Hol97,IS99]. This made it more suitable than SPIN for a natural representation of these algorithms.

The modeled state contains two representations of the set: one comprises a linked list of cells whereas the other is summarized as a bit vector. The linked list is updated as proposed here using atomic `d_step` instructions to implement CAS. The bit vector is checked or updated using further `d_step`s at the proposed linearization points.

The model was parameterized according to the number of concurrent threads, the number of operations that each would attempt and the range of key values that could be used. The two largest configurations we could practicably test were with four threads, each performing one operation with three potential keys and with two threads each performing two operations with four potential keys.

5.3 Practical Testing

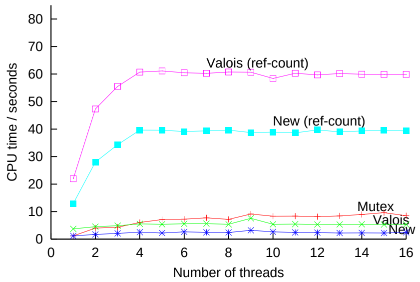
The linearizability of the operations has also been tested pragmatically. Although such tests cannot provide the assurances of formal methods they are nonetheless important because they avoid the need to make simplifying assumptions for tractability. In particular, the use of relaxed memory models means that the operations supported by a conventional shared memory machine are not linearizable; a direct implementation is likely to fail without further memory barrier instructions.

It is not generally possible to record actual timestamp values for arbitrary operations within an running process. Instead, we surrounded the code executed at each linearization point with further instructions to record coherent per-processor cycle counts.

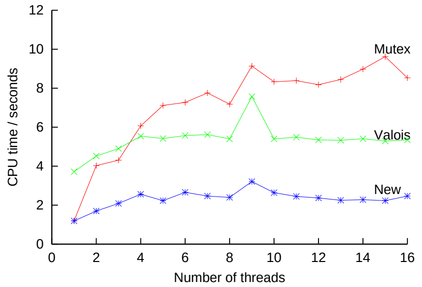
The resulting intervals were recorded to an in-memory log which was then replayed sequentially in timestamp order. The results thus obtained were compared with those from the concurrent execution. The replay program contains simple heuristics to deal with overlapping intervals. If these cannot determine a consistent linearized order then the replay program reports unresolved inconsistencies for manual inspection and re-ordering.

6 Results

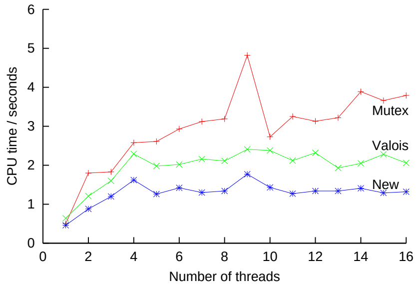
The algorithm described in Sect. 4 has been implemented in a combination of C and SPARC V9 assembly language. We evaluated its performance on an E450



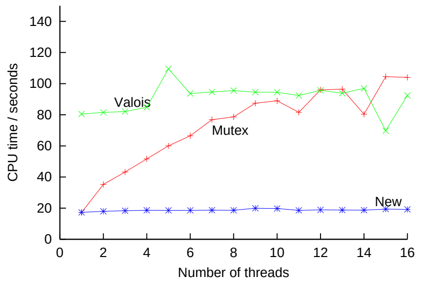
(a) All five algorithms operating with keys in the range $0 \ldots 255$.



(b) Non-reference-counted algorithms with keys $0 \ldots 255$.



(c) Non-reference-counted algorithms all operating on key 0.



(d) Non-reference-counted algorithms on keys $0 \ldots 8191$.

Fig. 6. CPU time (user+system) accounted to the benchmark appliction for each algorithm on a variety of workloads. In each case the x-axis shows the number of concurrent threads.

server running Solaris 8 and fitted with four 400MHz SPARC V9 processors and 4GB physical memory. It is worth emphasising that the code in Fig. 1-4 is intended merely as pseudo-code and does not reflect an optimised (or even necessarily correct) implementation. Processors may require additional memory barriers – for example between initializing the fields of a new node and introducing it into the list, or between the CAS that logically deletes a node and the CAS that physically deletes it.

The test application compared our implementation against Valois' lock-free algorithm and against a straightforward one in which the list is protected by a mutual exclusion lock[1]. Both lock-free algorithms were evaluated with and without reference-counting. All list cells were allocated ahead of time so that the performance of particular memory allocation functions was not included in the

[1] This comparison against a lock-based algorithm is somewhat unfair: the simplified programming model there makes it straightforward to implement a more efficient data structure such as a tree or skiplist.

results. The code to manipulate reference-counts is based on Valois' as modified by Michael and Scott with the exception that reference counts are recursively decremented when a cell is freed.

We generated a workload of insertion and deletion operations by randomly choosing keys uniformly distributed within a particular range, selecting equiprobably between insertions and deletions. We used per-thread linear congruential random number generators with the same parameters as the `lrand48` function from the Solaris 8 `libc` library. Seeds were chosen to give non-overlapping series.

The test harness was parameterized on the algorithm to use, the number of concurrent threads to operate and the range of keys that might be inserted or deleted. In each case every thread performed 1 000 000 operations. Figure 6 shows the CPU accounted to the process as a whole for each of the algorithms tested on a variety of workloads.

It is immediately apparent that our algorithm performs notably better for every experiment using more than one thread. In the case of single-threaded execution it outperforms Valois' algorithm in these tests and its performance equals that the lock-based implementation. The relative performance compared with Valois' algorithm is not surprising: we avoid the need to create, traverse and excise auxilliary nodes.

In addition to the workloads presented in those graphs we also tested configurations with larger ranges of keys, or where the list was initially 'primed' with a long sequence of nodes that would never be deleted. In each case this increased the total number of nodes in the list and thereby added to the cost of retrying operations when CAS instructions fail. One fear was that the lock-free algorithms would start to perform poorly because of the potential for multiple retries. We studied workloads up to lists of 65 536 elements and were unable to find any configuration for which the algorithms based on mutual-exclusion give the best performance. We suspect that although each retry becomes more costly, the likelihood of retries decreases as the rate of conflicting updates falls.

Figure 6a shows the performance of reference-counted implementations. The CPU requirements of Valois' algorithm are degraded by a factor of 5 in the single-threaded case, rising to over 11 for sixteen threads. Similarly, the CPU time required by our algorithms is degraded by a factor of 10 rising to over 15. In each case this is a consequence of need to manipulate reference-counts (using CAS operations) at each stage during a list's traversal. Valois reports that he had originally intended to assume the use of a tracing garbage collector [Val01].

The performance of reference-count manipulation is hampered because the SPARC processor does not provide atomic fetch-and-add. However, measurements taken on a dual-processor Intel x86 machine (with that facility [PPr96]) suggest that the degradation is low when compared with the overall costs seen here. When the reference counts lie on separate lines in the L1 data cache then updates implemented through CAS are 10% slower than those using fetch-and-add. This rises to a factor of 2 degredation when the two processors attempt to update the same address.

Of course, our results are optimistic in that they do not consider the cost of performing GC. However, as Jones and Lins write, if the size of the active data structure is fixed then the cost of copying collectors may be reduced arbitrarily at the expense of the total heap size [JL96]. More practically, they report that overall costs of around 10-20% are typical in modern well-implemented systems.

We examined a further approach to storage reclamation based on the deferred freeing of nodes. In this scheme each node contains an additional field through which it can be linked onto a to-be-freed list when it is excised from the main list. Each thread takes a snapshot of a global timer as its *current time* before starting each operation. Entries are removed from a to-be-freed list when the time of their excision precedes the minimum current time of any thread: at that point no thread can still have a reference to the node held in any of its local variables.

Our implementation allocates a pair of to-be-freed lists for each thread. These are termed the *old list* and the *new list* and are held along with a separate per-thread timer snapshot that is more recent than the excision time of any element of the old list. When the minimum current time exceeds the snapshot then the entire contents of the old list are freed and the elements of the new list are moved to the old list.

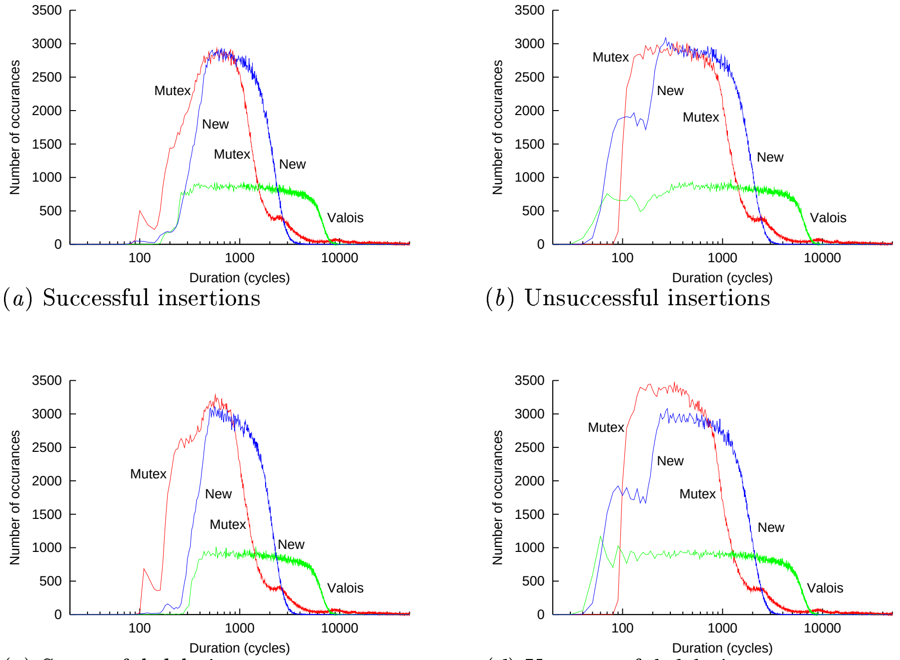
This deferred freeing scheme introduces two principal overheads when compared with the used of garbage collection. Firstly, a CAS operation is needed to place nodes on a to-be-freed list – in our implementation this increased the CPU requirements by 15% compared with the results from Fig. 6a operating with 16 threads. The second overhead is the cost of removing elements from the to-be-freed lists and establishing when it is safe to do so. This was a further 1% when performed every 1000 operations and 5% every 100, rising to 52% if performed after every operation.

Figure 7 presents a further analysis of the run-time performance of the three non-reference-counted algorithms, showing the distribution of execution-times for four different kinds of operation. These results were gathered when 8 concurrent threads performing insertions and deletions of keys in the range $0 \dots 255$. In the case of successful operations the lock-based implementation is able to achieve lower execution times than either lock-free scheme. However, it is also occasionally prone to much longer execution times which explain the higher mean execution time suggested by Fig. 6.

The situation is somewhat different for unsuccessful operations in that both lock-free algorithms obtain some execution times which are lower than those of the lock-based implementation – recall that unsuccessful operations may occur without requiring any CAS operations or other updates to the data structure.

7 Delete Greater-than-or-equal

Now consider the problem of implementing a further operation of the form DeleteGE(k) which returns and removes the smallest item that is greater than or equal to k. It is tempting to implement this by modifying `List::delete` so



(*a*) Successful insertions (*b*) Unsuccessful insertions

(*c*) Successful deletions (*d*) Unsuccessful deletions

Fig. 7. Operation-time distributions. Each graph shows execution times (in processor cycles) on the x-axis and numbers of occurances on the y-axis.

that the test `T1` does not fail if the key of the right node is greater than the search key.

Unfortunately this implementation is not linearizable. Suppose that three insertion operations are executed in sequence: Insert(20), Insert(15), Insert(20). The first two succeed and the third must fail because 20 is already in the set. However, consider a concurrent DeleteGE(10) operation, attempting to delete any node with a key greater than or equal to 10. Concurrent execution may proceed as follows:

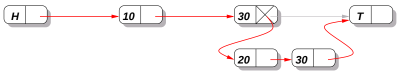
- `List::deleteGE` invokes `List::search` immediately after the first insertion of 20. It takes the head of the list as the left node and the node containing 20 as the right.
- The successful insertion of 15 occurs.
- The unsuccessful insertion of 20 occurs, observing 15 as the key of its left node and 20 as the key of its right node.
- `List::deleteGE(10)` completes after logically deleting the node containing 20.

We must order this DeleteGE(10) operation such that its result of 20 would be obtained by a sequential execution. This requires it to be placed before the insertion of 15 because otherwise the key 15 should have been returned in preference to 20. However, we must also linearize the deletion after the failed insertion of 20 because otherwise that insertion would have succeeded. These constraints are irreconcilable.

Intuitively the problem is that at the execution of `C3` the right node need not be the immediate successor of the left node. This was acceptable when considering the basic Delete(k) operation because we were only concerned with concurrent updates affecting nodes with the same key. Such an update must have marked the right node and so `C3` would have failed. In contrast, during the execution of `List::deleteGE`, we must be concerned with updates to any nodes whose keys are greater than or equal to the search key.

We can address this by retaining the implementation of `List::deleteGE` but changing `List::insert` in such a way that `C3` must fail whenever a new node may have been inserted between the left and right nodes. This would mean that, whenever `C3` succeeds, the key of the right node must still be the smallest key that is greater than or equal to the search key.

This is achieved by using a single `CAS` operation to (a) introduce a pair of new nodes, one that contains the value being inserted and another that duplicates the right node and (b) mark the original right node:



Such a CAS conceptually has two effects. Firstly, it introduces the new node into the list: beforehand the `next` field of the successor is unmarked and therefore the right node must still be the successor of the left node. Secondly, by marking the contents of that `next` field, the CAS will cause any concurrent `List::deleteGE` with the same right node to fail. Note that the key of the now-marked right node is not in the correct order. However, the existing implementations of `List::search`, `List::delete`, `List::find` and `List::deleteGE` are written so that they do not rely on the correct ordering of marked nodes.

8 Conclusion

This paper has presented a new non-blocking implementation of linked lists. We believe that the algorithms presented here are linearizable. They have also been implemented and we have shown that their measured performance improves both on previously published non-blocking data structures and also on a lock-based implementation.

8.1 Acknowledgments

The work described in this paper was carried out during an internship with the Java Technology Research Group at Sun Labs. The design presented here is the result of much fruitful discussion with Dave Detlefs, Christine Flood, Alex Garthwaite, Steve Heller, Nir Shavit and Guy Steele. The implementation and evaluation have similarly benefitted from feedback from Mike Burrows, Keir Fraser, Steven Hand, Mark Moir and John Valois.

References

[ABP98] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, Puerto Vallarta, Mexico, June 28–July 2, 1998. SIGACT/SIGARCH.

[GC96] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 123–136, Berkeley, CA, USA, October 1996. USENIX.

[Gre99] M Greenwald. *Non-blocking synchronization and system design*. PhD thesis, Stanford University, August 1999. Technical report STAN-CS-TR-99-1624.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[Her93] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[Hol97] Gerard J Holzmann. The moel checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[IS99] Radu Iosif and Riccardo Sisto. dSPIN: A dynamic extension of SPIN. In *Proc. of the 6th International SPIN Workshop*, volume 1680 of *LNCS*, pages 261–276. Springer-Verlag, September 1999.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons, July 1996.

[LaM94] Anthony LaMarca. A performance evaluation of lock-free synchronization protocols. In *Proceedings of the Thirteenth Symposium on Principles of Distributed Computing*, pages 130–140, 1994.

[MP91] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, 1991.

[MS95] Maged M. Michael and Michael L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, Computer Science Department, December 1995.

[PPr96] *Pentium Pro Family Developer's Manual, volume 2, programmer's reference manual.* Intel Corporation, 1996. Reference number 242691-001.

[Val95] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 214–222, Ottawa, Ontario, Canada, 2–23 August 1995.

[Val01] John D Valois. Personal communication. March 2001.