1 MANAGING LONG LINKED LISTS USING LOCK FREE TECHNIQUES

Mohammad Farook and Peter Graham

University of Manitoba, Canada

Abstract

When writing parallel programs for shared memory machines, it is common to use shared data structures (linked lists, queues, trees, etc.). Concurrency control for such data structures may be implemented using either blocking or non-blocking/lock-free synchronization. Blocking synchronization is popular because it is familiar and also amenable to worst-case performance analysis (important for certain real time applications). Unfortunately it also suffers from some undesirable characteristics. These include the possibility of deadlock, limited concurrency since processes are blocked, and certain scheduling anomalies. Non-blocking techniques, though less prevalent, do not suffer from these problems and as such may be well suited to many parallel programming problems.

Despite increased interest in non-blocking techniques, even simple data structures such as singly linked lists often have annoying limitations. This paper presents a new implementation of non-blocking linked lists which overcomes some common problems with existing implementations. In addition to offering greater overall concurrency, the presented technique also significantly increases performance for long *linked lists. We provide algorithms for list traversal, insertion and removal, show their correctness, and also analyze the results of a simulation study which compares the performance of our implementation against two existing techniques under varying load conditions and access patterns and for various list lengths.*

Keywords: Concurrency Control, Lock Free Data Structures, Non-Blocking Synchronization, Parallel Programming

1. Introduction

In shared memory parallel programming, concurrently executing processes/threads operate on shared data structures to cooperate to solve a problem. Conventionally, the consistency of such data structures is maintained by ensuring mutually exclusive access to them using locks or other blocking synchronization primitives. Blocking techniques, however, have several drawbacks including decreased concurrency, the potential for deadlock, and certain undesirable scheduling side effects (i.e. priority inversion and convoying). An alternative approach is to use non-blocking synchronization techniques (E.g. (Barnes, 1993), (Greenwald et al, 1996), (Herlihy, 1988), (Herlihy, 1991), (Massalin et al, 1991), (Prakash et al, 1991), (Valois, 1995)). Non-blocking techniques do not suffer from these problems since processes optimistically execute concurrently and therefore never wait on one another. As such, they offer distinct advantages for certain parallel applications. Non-blocking techniques that make hard guarantees concerning completion of processes are often referred to as being "wait free".

Existing work on non-blocking data structures has been fruitful, yielding non-blocking implementations of a wide variety of conventional data structures including linked lists, queues, and trees. Nevertheless, even for such simple structures as singly linked lists, problems persist. Among the limitations of current implementations are high overhead, restrictions on types of concurrency, and high re-execution costs in the face of conflicting concurrent operations. This paper addresses these problems for singly linked lists.

Through simple changes to existing techniques for non-blocking linked lists, we derive a new implementation that significantly improves on existing techniques and that offers increasing performance as the size of the list grows.

The rest of this paper is organized as follows. Section 2 presents related work. Section 3 discusses the limitations of existing non-blocking linked list designs. Section 4 proposes a new design that addresses the problems, describes the algorithms, and then illustrates their correctness. Simulation results comparing the performance of our design to two other designs are presented in Section 5. Finally, Section 6 concludes and discusses possible future research.

2. Related Work

Work related to that presented in this paper falls into three categories; non-blocking synchronization primitives, generic techniques for implementing non-

blocking data structures and specific techniques for implementing non-blocking linked lists. Work already done on other non-blocking data structures is beyond the scope of this paper and is therefore not discussed.

2.1 Non-Blocking Synchronization Primitives

Shared, non-blocking data structures are built using different synchronization primitives than those from which shared, blocking data structures are implemented. The most discussed of these synchronization primitives is the "Compare And Swap" (CAS) instruction[1].

When using the CAS instruction, a programmer records the state of a shared variable (commonly a pointer for shared data structure implementations) at the beginning of a computation. A new value for the shared variable is then computed and the CAS instruction is used to atomically update the shared variable with that value only if it has not changed value from the beginning of the computation. If the shared variable cannot be updated, then an error indication is returned and the computation must be re-executed[2] using the new value for the shared variable. This technique may be used to "swing" a pointer in a linked list to point to a new node. This use of CAS to effect a "conditional store" is the basis for both insertions into and deletions from lists.

CAS is a three-operand atomic operation (pseudo-code for the CAS operation is shown in Figure 1). The operands A, B, and C are single word variables. C is a shared variable and A is a private copy of C made prior to a computation involving C. B is the newly computed value for C which the process wishes to write into C. The process is allowed to write B into C only if C is unchanged since it was last read (i.e. only if A==C).

```
CAS(A,B,C)
BEGIN ATOMIC
if (A==C) {A=B;  return TRUE; }
else { return FALSE; }
END ATOMIC
```

Figure 1 – Implementation of CAS

---

[1] Most modern machine architectures provide support for both blocking and non-blocking synchronization in hardware via specific machine instructions.
[2] Such techniques utilizing re-execution are familiar to the database community as *optimistic* concurrency control as used in database management systems.

The use of CAS suffers from what is known as the “A-B-A problem”. This occurs when the value of C is the same as A even though C has been modified at least once since the process read the value of C into A. In this situation, the CAS instruction succeeds even though the value B was computed using stale data. In certain applications this may lead to erroneous results.

A variant of CAS is the “Double Compare And Swap” (DCAS) instruction. This form of CAS exchanges not one, but two values indivisibly. If the second value is used as a “version number” for the first (which is incremented on every update to the first shared variable) then the A-B-A problem may be avoided. The pseudo-code for the DCAS instruction is shown in Figure 2.

```
DCAS(A1,A2,B1,B2,C1,C2)
BEGIN ATOMIC
if ((A1==B1) && (A2==B2)) { A1=C1; A2=C2; return TRUE; }
else { return FALSE; }
END ATOMIC
```

Figure 2 – Implementation of DCAS

Recently a number of processors (e.g. the MIPS family of machines) have implemented a more general non-blocking synchronization primitive in the form of the “Load Linked” (LL) and “Store Conditional” (SC) instruction pair. Our new algorithms for insertion and deletion into a shared linked list which are presented in Section 4 utilize only the CAS and DCAS instructions.

2.2 Generic Non-Blocking Data Structures

A number of so-called “universal methods” (E.g. (Barnes, 1993), (Herlihy, 1993), (Herlihy et al, 1993), (Herlihy et al, 1987), (Prakash et al, 1991)) for constructing non-blocking data structures of any type have been discussed in the literature. (Lamport, 1983) described the first lock-free algorithm for the problem of managing a single-writer, multiple-reader shared variable. Such variables may be used as a basis for developing a wide variety of non-blocking data structures but Lamport’s work is not generally considered to be a universal method.

(Herlihy, 1993) presented the first widely accepted universal method. He maintained a write-ahead log of operations for each shared data structure. The order of entries in the log determines a serialization order. Private copies of structures, built by applying a sequential reconstruction algorithm to the

operations in the log, are updated and then finally added to the log itself. Check-pointing the log is used to decrease the cost of state reconstruction.

(Herlihy et al, 1987) also proposed a method for constructing non-blocking data structures using “copying”. Utilizing the ‘Load Linked’ (LL) instruction, a copy of the data structure (or in certain situations, a subset of it) may be made. Changes are made to the local copy and then the ‘Store Conditional’ (SC) instruction is used to update the shared structure. Herlihy’s method has two drawbacks. First, it can be very expensive in terms of copying overhead if the data structure is large. Second, it permits only a single process to successfully update the structure at a time.

A third universal method has been proposed by (Barnes, 1993). In this technique, private copies of individual cells/nodes are made (i.e. they are “cached”) using LL. Updates are made to the cells and then they are validated and returned to the shared structure using SC. An interesting feature of Barnes’ work is that it supports “cooperative completion” whereby one process may complete the work of another process rather than having that process fail its update. To do this, each process records its operation list in a global work queue. If a process is interrupted before completing its work, the remaining operations will be recorded in the work queue and may be completed by a later executing process when it accesses the shared data structure.

2.3 Non-Blocking Linked Lists

Non-blocking algorithms that are specific to particular data structures are typically more efficient than universal methods. In this paper we focus on algorithms for insertions and removals on non-blocking singly linked lists. We now review existing approaches that will be used for comparison purposes with our new technique described later in Section 4.

(Valois, 1995) describes a non-blocking linked list where every list node has an “auxiliary” node that is used during concurrent insertions and deletions to help maintain the consistency of the list. Each auxiliary node consists of a single pointer to the next regular node in the list. An empty linked list consists of two dummy nodes ‘First’ and ‘Last’ with an auxiliary node between them.

A structure known as a cursor is defined which contains three pointers to nodes (dummy or regular) which surround the point of insertion/deletion. The first pointer, ‘target’, points to the node being visited. The second pointer, ‘pre-aux’, points to the auxiliary node immediately preceding ‘target’. The final pointer, ‘pre-cell’, points to the last *normal* node preceding ‘target’ in the list.

Referring to Figure 3, suppose that a process wants to insert a node ‘new’ (together with the required auxiliary node ‘new.aux’) between nodes ‘B’ and ‘C’ in the shared linked list. The cursor ‘c’ maintains pointers, ‘c.pre-aux’ pointing to the auxiliary node before node ‘C’ and ‘c.target’ pointing to node ‘C’. The new node and following auxiliary node are created and initialized so that node ‘new’ points to its auxiliary and its auxiliary points to node ‘C’. CAS is then used to atomically check that ‘c.pre-aux’ still points to ‘c.target’. If the pointers have not changed, then ‘c.pre-aux’ is set to point to ‘new’ and ‘TRUE’ is returned. If they have changed, then ‘FALSE’ is returned and the process must attempt the insertion again.

When a process wants to do a deletion, the node to be deleted is pointed to by the ‘target’ node in the cursor ‘c’. CAS is used to atomically set ‘c.pre-aux’ to c.target^.next. The result of a deletion is a pair of adjacent auxiliary nodes. As deletions continue, long chains of contiguous auxiliary nodes may develop. As described, Valois’ algorithm also suffers the A-B-A problem since it uses CAS.

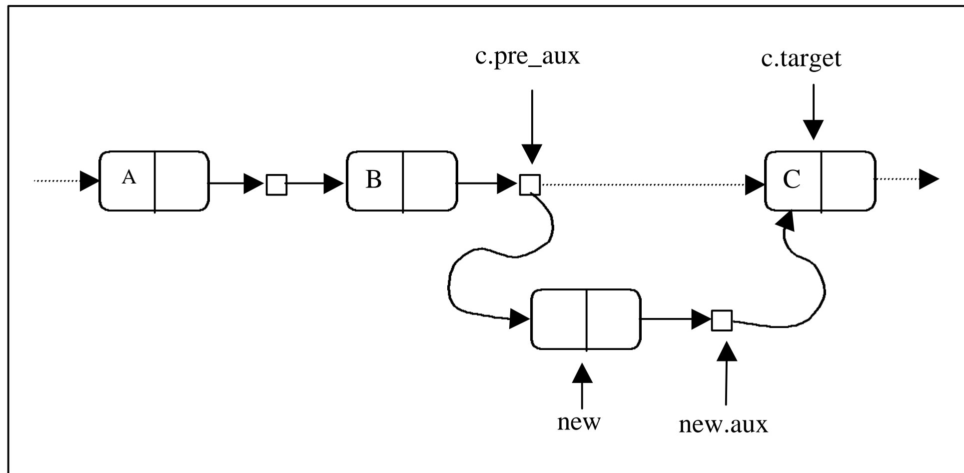

Figure 3 – Insertion under Valois’ Implementation

In a recent paper, (Greenwald et al, 1996) propose the use of non-blocking data structures in the design of a multiprocessor operating system. They describe a non-blocking linked list approach that uses the DCAS instruction and a version number for each list to avoid the A-B-A problem.

During deletion, a process traverses the linked list until it finds the node to be deleted. If the node is not found, the version number is checked. If the

version number has changed, then the desired node may have just been inserted and the list must be scanned again. Assuming the node has been found, DCAS is used to atomically swing the next pointer of the node preceding the target to point to the node following it and to increment the version number. This may fail or succeed depending on whether the list has been changed or not.

Inserting an element is done by traversing the linked list to locate the node after which the new node should be inserted. DCAS is then used to increment the version number and swing the next pointer of the target node to point to the new node (in much the same way as in Valois' algorithm). Again the DCAS may succeed or fail depending on the state of the list.

While Greenwald and Cheriton's algorithm is simple and intuitive, the use of a single version number is too coarse grained to be effective. This is particularly true as the size of the linked list grows.

Finally, (Massalin et al, 1991) have also described a non-blocking implementation of a shared linked list using the DCAS instruction for deletion. Deletion of nodes from within the linked list is done in two steps. First, the nodes that are to be deleted are marked but left in the list. Then, in the second step, if the node preceding the one marked for deletion is not itself marked for deletion the marked node may be safely removed. This implementation, however, limits concurrency to a single process.

3. Problems with Existing Non-Blocking Linked Lists

There are a number of problems with existing non-blocking implementations of linked lists. Many of these problems have already been illustrated in the implementations discussed in the preceding section.

First, successful concurrency is seldom attained. This is true of the universal methods as well as many of the linked list specific approaches. By successful concurrency we mean concurrent operations on the list, which do not cause re-execution of one another. For example, insertions into distinct parts of a linked list should not affect one another yet, for example, in Greenwald and Cheriton's algorithm and in Massalin and Pu's algorithm they do. In the former, this is due to the coarse granularity at which the version number is applied. As there is a single version number for the entire linked list, even a change to an unrelated part of the list results in the re-execution of other concurrent updates.

The A-B-A problem is also a concern for those techniques that do not use an explicit version number (e.g. Valois' algorithm). For some applications of

linked lists, the A-B-A problem may not be tolerated and thus it must be addressed.

While Valois' algorithm does permit concurrent updates and may be modified to manage the A-B-A problem, it introduces execution overhead in two ways. First, long chains of auxiliary nodes may develop after many deletions have taken place. This introduces overhead either in traversing them (Valois' algorithm suffers 100% overhead in list traversal even when there are no adjacent auxiliary nodes) or to delete the excess nodes. Second, when re-execution is necessary due to two adjacent update operations in the list, the list must always be re-scanned from the start of the list. Particularly for long linked lists, this represents high overhead in an environment where such conflicts are relatively common. Massalin and Pu's algorithm also suffers from this latter form of overhead.

4. An Improved Algorithm for Non-Blocking Linked Lists

The desirable characteristics of an effective non-blocking implementation of linked lists are:

- As much successful concurrency as possible,
- Effective handling of the A-B-A problem,
- Minimal re-execution costs after conflicting operations, and
- Minimal overhead relative to a conventional linked list implementation (time overhead primarily but also space overhead as a secondary concern).

We now present an implementation that achieves most of these goals. It does provide a high degree of successful concurrency, handles the A-B-A problem, has minimal re-execution costs in all but one case, and has minimal time overhead compared to a conventional linked list implementation. It does achieve these goals at the expense of increased space requirements but we consider the additional space overhead to be negligible for all linked lists except those storing the simplest (and hence, smallest) data items.

4.1 Structure of the Linked List

The structure of the linked list in our design includes two dummy nodes 'head' (pointing to the first node in the linked list) and 'tail' (pointing to the last). An empty list consists of these two nodes pointing to one another.

Each process manipulating a particular node, 'target', maintains three pointers; 'prev', 'target', and 'next'. The 'prev' pointer points to the node preceding 'target' and the 'next' pointer points to the node following 'target'. These three pointers are obtained by traversing the linked list.

Each node in the linked list consists of four fields; a data field, two pointer fields, and a count field. One of the pointers (PTR) is used to maintain an up-to-date linked list. These pointers never point to a node that has been deleted or one that has been marked for deletion. The other pointer (TRAVERSE) is used to facilitate concurrent processes traversing the list despite the fact that some nodes may have been concurrently marked for deletion. This situation occurs when a process is at a particular node when another process marks that node for deletion (by making its pointer field null). The first process may still traverse the list using the spare pointer thereby increasing overall concurrency. Actual deletions within the list are more difficult because a node may be deleted and de-allocated while another process is traversing the list. If a deleted node is then re-used for some other purpose, its new pointer values may lead to invalid memory references. To avoid this problem, the counter field in each list node is used. Whenever a process is visiting the node, it increments the counter field and decrements it only when it is leaving. A process attempting a node deletion must first verify that its counter field is zero before proceeding.

An example of the proposed linked list is shown in Figure 4.

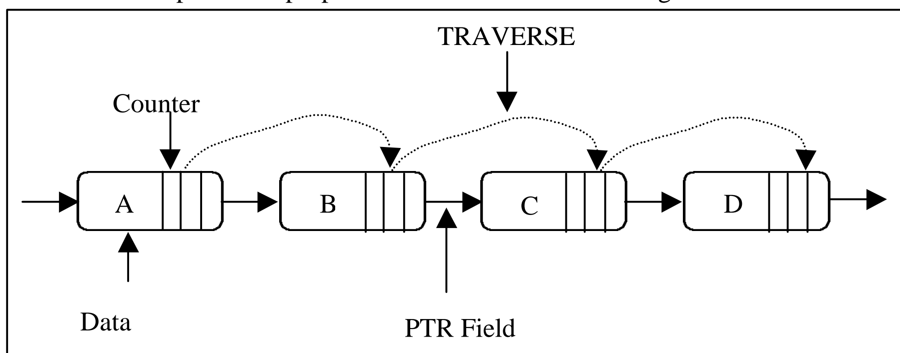

Figure 4 – Structure of the Linked List

4.2 Linked List Traversal

Algorithm 'Cursor' (see Figure 5) returns the FIRST and LAST nodes as 'prev' and 'target' if the list is empty (lines 2-5). When a process finds the key by traversing the list using the 'ptr' pointer, it returns pointers to the 'prev',

‘target’ (matching the key), and ‘next’ nodes (lines 7-11). If the key is not found, the algorithm returns ‘NOTFOUND’. If the ‘ptr’ field of any node is NULL and the node is currently being visited by another process, then the ‘traverseptr’ is used to traverse the list. While traversing the list, each process decrements the counter of a node it is leaving and increments the counter of a node it is about to traverse (lines 14 and 16).

If a process visiting a node which has a null pointer field finds the following node contains the key it returns ‘TRY_AGAIN’ (lines 21-25) which will force the caller to re-invoke algorithm Traverse. This is necessary because the process must re-read its pointers to get a non-stale ‘prev’ pointer. Finally, if the ‘ptr’ field is not null and the following node does not contain the key then the search continues (lines 26-30).

```
Algorithm Cursor(key) // prev, target, & next are globally visible
1.prev=FIRST;
2. if (prev->ptr==LAST) { // empty list
3.     target=LAST;
4.     return NOTFOUND;
5. }
6. while (prev->ptr!=LAST) { // traverse the list
7.     if ((prev->ptr!=NULL) && (prev->ptr->data==key)) {
8.             target=prev->ptr;
9.             next=target->ptr;
10.            return FOUND;
11.    }
12.    if ((prev->ptr==NULL)&&(prev->traverseptr->data!=key)) {
13.            // marked for deletion so use traverseptr
14.            prev->counter--;
15.            temp=prev;
16.            prev->traverseptr->counter++;
17.            prev=prev->traverseptr;
18.            if (temp->counter==0) release(temp);
19.            continue;
20.    }
21.    if ((prev->ptr==NULL)&&(prev->traverseptr->data==key)) {
22.            // check if prev points to key via traverseptr
23.            prev->counter--;
24.            if (prev->counter==0) release(prev);
25.            return TRY_AGAIN;
```

```
26.    } else { /* follow normal (PTR) field to next node */
27.               if (prev!=FIRST) prev->counter--;
28.               prev->ptr->counter++;
29.               prev=prev->ptr;
30.    }
31. }
32. return NOTFOUND;
End Cursor
```

Figure 5 – Linked List Traversal

4.3 Deletion

The algorithm for deleting a node from the linked list is shown in Figure 6. It depends on another algorithm called 'TryDelete' which is shown in Figure 7.

```
Algorithm Delete(key)
1.  flag=TRUE;
2.  while (flag==TRUE) {
3.      repeat {
4.               r=Cursor(key);      // - delete position returned in 'prev',
5.      } until (r!=TRY_AGAIN);              //    'target', and 'next'
6.      if (r==FALSE) return NOTFOUND;
7.      repeat {
8.               t=TryDelete(prev,target,next); // attempt deletion
9.      } until (t!=DELETE_AGAIN);
10.    if (t!=TRY_AGAIN) flag==FALSE; // deletion successful
11. }
12. return t;
End Delete
```

Figure 6 – Deletion from the Linked List

Algorithm 'Delete' repeatedly tries to delete a node with the given 'key'. Repetition is necessary because of potentially concurrent insertions/deletions of the target node. The algorithm first uses algorithm 'Cursor' to determine the location of the target node (lines 3-5). If a node with the required key is not found, then NOTFOUND is returned (line 6). If the target node exists, algorithm 'TryDelete' is then called to do the deletion (lines 7-9). TryDelete may return 'TRY_AGAIN' causing the repeat loop to re-execute.

```
Algorithm TryDelete(prev,target,next)
```

```
1.  r=DCAS(prev->ptr,target->ptr,target,next,next,NULL);
2.  if (r==TRUE) {
3.      prev->counter--; if (target->counter==0) release(target);
4.  } else {
5.      if ((target->ptr!=next)&&(target->ptr!=NULL)) {
6.                  next=target->ptr; return DELETE_AGAIN;  }
7.      prev->counter--;
8.      if ((prev->counter==0)&&(prev->ptr==NULL)) release(prev);
9.      return TRY_AGAIN;
10. }
11. return (r?FOUND:NOTFOUND); /* C conditional execution */
End TryDelete
```

Figure 7 – Delete an existing node from the Linked List

Algorithm ‘TryDelete’ (Figure 7) uses DCAS to attempt the deletion of the target node. If DCAS is successful, it sets the next field of node ‘prev’ to point to node ‘next’. The code then decrements the counter of node ‘prev’ (as it is no longer being used) and, if possible, releases the deleted node (line 3) to reclaim space. If the DCAS instruction fails it is because one or more of the pointers have changed. If the next pointer has changed, the ‘next’ pointer is updated and ‘DELETE_AGAIN’ is returned (line 6). In this case, algorithm ‘Delete’ can call algorithm ‘TryDelete’ again without re-traversing the list (a significant saving over Valois). If the ‘prev’ pointer has changed, however, the algorithm doesn’t know what the new ‘prev’ pointer should be so it returns ‘TRY_AGAIN’.

4.4 Insertion

The algorithm for inserting an element into the linked list is shown in Figure 8.

```
Algorithm Insert(key,value)
1. repeat { R=Cursor(key);} until (r!=TRY_AGAIN);
2. if (r==FOUND) {
3.     repeat {t=TryInsert(prev,target,value); }until (t!=RETRY); }
4. return t;
End Insert
```

Figure 8 – Insertion into the Linked List

Algorithm 'Insert' first uses algorithm 'Cursor' to locate the correct insertion point in the list (line 1). As with the deletion case, there is a second, companion algorithm 'TryInsert' shown in Figure 9 which is used to perform the actual insertion once the insertion point is known.

The 'TryInsert' algorithm starts by allocating a new node to store the new value. It initializes the node to point to the following node in the list ('target') and then uses CAS (line 6) to attempt to insert the newly created node. If CAS fails, the new node is released and a check is made to see if the target node has been marked for deletion (line 9). If so, the 'target' pointer is updated and 'RETRY' is returned (lines 10-11) so that the 'Insert' algorithm will call 'TryInsert' again. Otherwise, the 'prev' node's counter field is decremented (line 14) and the appropriate result status is returned (line 15).

Note that due to space limitations, the insertion and deletion algorithms presented do not attempt to handle duplicate key values in the list.

```
Algorithm TryInsert(prev,target,value)
1. node=new node();
2. node->data=value;
3. node->counter=0;
4. node->ptr=target;
5. node->traverse_ptr=target;
6. r=CAS(prev->ptr,target,node);          // try to insert the node
7. if (r==FALSE) {
8.     release(node);
9.     if (prev->ptr!=NULL) {    // check for retry
10.            target=prev->ptr;
11.            return RETRY;
12.    }
13. }
14. prev->counter--;
15. return FOUND;
End TryInsert
```

Figure 9 – Insertion into the Linked List at a known Position

4.5 Correctness

Space does not permit a formal proof of correctness of the algorithms presented. We sketch such a proof and refer the reader to (Farook, 1998) for details.

We begin by noting that non-adjacent updates do not affect one another since synchronization is localized (i.e. fine granularity concurrency control). The correctness of such inserts and deletes are guaranteed simply through the specified use of CAS and DCAS (respectively). There are four cases to consider:

1. adjacent inserts,
2. adjacent deletes,
3. insert followed by an adjacent delete, and
4. a delete followed by an adjacent insert.

Theorem 1: *When processes try to insert nodes between the same set of adjacent nodes 'prev' and 'target', only one of the processes succeeds while the others fail (and re-execute).*

In this case, two or more processes have read the same set of nodes 'prev' and 'target' to insert a new node between them. Only that process, from among the processes competing to insert the nodes, which finds the pointer of the node 'prev.ptr' and the nodes 'prev' and 'target' unchanged will succeed (see algorithm 'Insert'). All other processes will fail and later retry their insertions.

Theorem 2: *When processes try to delete adjacent nodes, some of the processes succeed while the others fail.*

In this case, two or more processes are trying to delete adjacent nodes. Consider two such processes. One process will succeed while the other will fail (because the first process will have updated one of the pointers checked by DCAS for the second). This can happen due to a change either to the node pointing to the deleted node or due to a change of the node the deleted node is pointing at. Similarly, when more than two processes are attempting adjacent deletions some of the processes will succeed (those that find their pointers unchanged) while others will fail (those that find modified pointers).

Theorem 3: *When processes try to insert and/or delete adjacent nodes then some of the processes succeed while others fail.*

As with the case for concurrent deletes of adjacent nodes, some processes will execute their CAS or DCAS operations successfully because no other process has yet updated any relevant pointers. These processes will update pointers that some other processes require to remain unchanged in order to succeed. Those processes will then fail and re-execute. Which processes

succeed and which fail cannot be determined a-priori as this is dependent on run-time scheduling order. It should, however, be noted that in a parallel programming environment, the order in which unrelated but potentially concurrent events on shared data structures takes place is insignificant.

We omit discussion of the pathological cases dealing with empty lists.

5. Performance of the New Algorithm

To assess the performance of our new implementation of non-blocking linked lists, we conducted a simulation study. We compared the behaviour of our algorithm with the algorithms of (Valois, 1995) and (Greenwald, et al 1996). We varied both the length of the linked list and the relative frequencies of inserts and deletes performed on the linked list. The number of concurrent processes was held constant at 20. For each experiment, a linked list having a fixed number of nodes was constructed. Each process was given a random insert or delete to perform and was then allowed to traverse the list for its time slice. The time slice was generated randomly and determines the number of nodes a process may visit. Once a process' time slice expires, other processes are allowed to operate on the linked list. Scheduling was performed in a round robin manner. This approach models a parallel environment in which each process executes alone on a processor (or where each process on a machine shares the processor equally).

Figure 10 shows the performance of the three algorithms with 100% deletes.

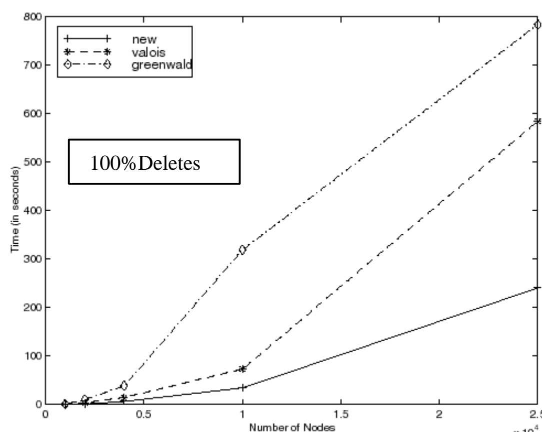

Figure 10 – 100% deletions

Figure 11 shows the performance of the algorithms with 100% inserts.

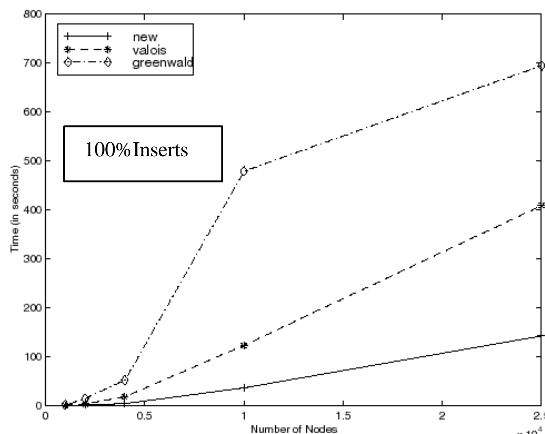

Figure 11 – 100% insertions

Figure 12 shows the performance when there are 50% inserts and 50% deletes

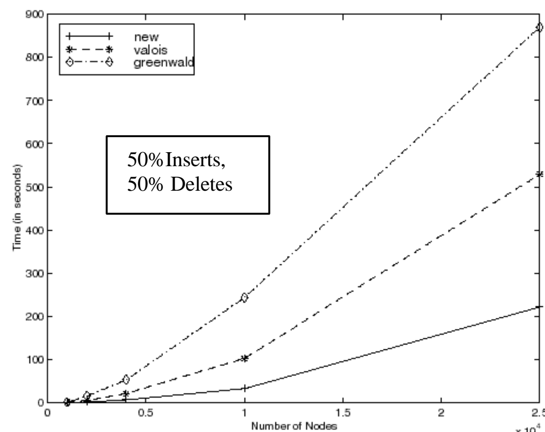

Figure 12 – 50% inserts, 50% deletions

In considering these graphs, it is clear that our algorithm is relatively insensitive to operation mix. From the data we also see that the performance of

our new algorithm is consistently better than both Valois' and Greenwald & Cheriton's algorithms and the performance differential improves with the size of the list. Further, although data is not provided in this paper, our algorithm also outperforms their algorithms for smaller link lists albeit less significantly.

There are three reasons why our algorithm performs better than the others. First, and foremost, unlike Valois' algorithm, we require only 'n' nodes to represent a list containing 'n' distinct key values. This offers significant savings during list traversal. The algorithm also provides significantly enhanced success in concurrent reading and writing. Concurrency control is fine-grained (at the level of node triples rather than the entire list as with Greenwald & Cheriton's approach). Finally, if, during a deletion, a process fails to delete a node because its 'next' pointer has changed, the updated next pointer value is available without re-reading the entire list up to the deletion point.

6. Conclusions & Future Work

In this paper we have described a new implementation for non-blocking linked lists. The presented algorithms (list traversal, insertion, and deletion) exploit a fine granularity synchronization strategy to significantly outperform existing algorithms, particularly for long linked lists.

Possible areas of future research include extensions to other linked data structures. The idea of localizing concurrency control can also be applied to structures such as trees and graphs. It remains to carefully develop the necessary algorithms. Application of non-blocking structures in new areas (e.g. parallel databases) is also of interest as is the idea of applying Barnes' "operation completion" concept to our algorithm to decrease the number of conflicts.

References

G. Barnes (1993). A Method for Implementing Lock-Free Shared Data Structures. Proceedings of the 5th International Symposium on Parallel Algorithms and Architectures, pp. 261-270.

M. Farook (1998). Fast Lock Free Link Lists in Distributed Shared Memory. MSc Thesis, Dept. of Computer Science, Univ. Manitoba,

M. Greenwald and D. Cheriton (1996). The Synergy Between Non-Blocking Synchronization and Operating System Structure. Proc. 2nd Symposium on Operating System Design and Implementation, pp. 123-13.

M. Herlihy (1998). Impossibility and Universality Results for Wait-Free Synchronization. Proceedings of the 7th International Symposium on Principles of Distributed Computing, pp. 276-290.

M. Herlihy (1991). Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123-149.

M. Herlihy (1993). A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745-770.

M. Herlihy and J.E.B. Moss (1993). Transactional Memory: Architectural Support for Lock-Free Data Structures. Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 289-301.

M. Herlihy and J. Wing (1987). Axioms for Concurrent Objects. Proceedings of the 14th International Symposium On Principles of Programming Languages, pp. 13-26.

L. Lamport (1983). Specifying Concurrent Program Modules. *ACM Transactions On Programming Languages And Systems*, 5(2):190-222.

H. Massalin and C. Pu (1991). A Lock-Free Multiprocessor OS Kernel, Columbia University Technical Report CUCS-005-91.

S. Prakash, Y.H. Lee, and T. Johnson (1991). Non-Blocking Algorithms for Concurrent Data Structures, University of Florida Report TR91-002.

J.D. Valois (1995). Lock-Free Linked Lists Using Compare-And-Swap, Proceedings of the 14th International Symposium on Principles of Distributed Computing, pp. 214-222.