Implementing Lock Free Data Structure for Shared-Memory Systems using Linked List

Nuku Atta Kordzo Abiew
Faculty of Computing and Information Systems, GTUC, Ghana

Dominic Asamoah
Department of Computer Science, KNUST, Ghana

ABSTRACT
It is becoming evident that non-blocking algorithms can deliver significant benefits to parallel system. 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 system which scale to highly-parallel environments and which are resilient to scheduling decisions. Lock-free data structures implement concurrent objects without the use of mutual exclusion. This approach can avoid performance problems due to unpredictable delays while processes are within critical sections. Although universal methods are known that give lock-free data structures for any abstract data type, the overhead of these methods makes them inefficient when compared to conventional techniques using mutual exclusion, such as spin locks. In using lock-free data structures and algorithms for implementing a shared linked list and skip list, it allows concurrent traversal, insertion, and deletion by any number of processes.

General Terms
Data Structure, Memory Management, Algorithms

Keywords
Exclusion, Linked List, Lock-free, Non-blocking, Spin-Lock.

1. INTRODUCTION
Mutual exclusion is one of the best and most standard ways of implementing data structure in distributed systems. With mutual exclusion, locks are used to resolve conflicts between processes attempting to access data structure simultaneously. Although this approach is widely used and it is easy to implement data structures this way, it has a major weakness, that is, when one process is in the critical section (that is, when it is modifying the data structure), all other processes must wait before they are permitted to access it. In contrast, an implementation of a shared-memory object is lock-free (or non-blocking), if for any possible execution, the system as a whole is always making progress, that is, some process is guaranteed to complete its operation.

Insertion and deletion using compare and swap during the design and implementation of lock-free data structure is of a major problem to most researchers and designers. In implementing insertion and deletion with a linked list with nodes A,B and C as shown in the Fig. 1, Node B can easily be deleted by moving Node A to right from Node B to Node D, and at the same time trying to insert Node C by switching B right from D to C. The resultant list will contain only node A and D. This implies that, Node B was deleted or removed successfully however there was unsuccessfully insertion of Node C. A similar problem arises when one try to delete two adjacent nodes concurrently. The root of these problems lies in the fact that both the right pointer of the deleting node and the right pointer of its preceding node cannot be control at the same time if using a simple C&S primitive. It can be done with a stronger double compare-and-swap primitive[1].The study is aimed at presenting a novel implementation of linked-lists which is non-blocking, linearizable and which is based on the compare-and-swap (C & S) operation found on contemporary processors.

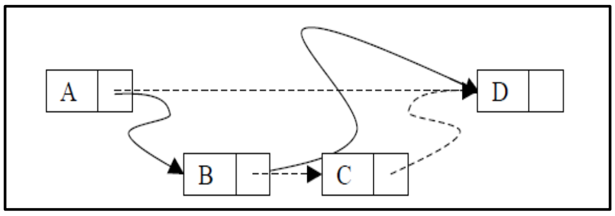

Fig 1: Concurrent deletion of B and insertion of C

2. LITERATURE REVIEW
Researchers have considered the benefits of avoiding mutual exclusion since at least the early 1970’s. Leslie Lamport [2] gave the first lock-free algorithm for the problem of a singe-writer/multi-reader shared variable. Herlihy [3] proved that for non-blocking implementation of most interesting data types (linked lists among them), a synchronization primitive that is universal, in conjunction with READ and WRITE, is both necessary and sufficient. Fisher et al[4] opined that COMPARE and SWAP is a universal primitive, since it can be used to solve consensus problem for any number of processes. Herlihy [5] gave the first algorithm for implementing lock-free using universal methods such as mutual exclusion. This novelty have been improved upon by several researchers. However, it has become increasingly apparent that universal methods suffer from several sources of inefficiency, such as wasted parallelism, excessive copying and generally high overhead. Massalin et al [6] coined the term lock-free and implemented it on a multiprocessor operating system kernel using lock-free data structures. However, their algorithms require a two word version of the COMPARE and SWAP synchronization primitive that is not widely available. Another implementation of lock-free data structure using linked list was also presented by Harris [7]. Michael [8] as part of his lock-free hash table design also presented an algorithm on the implementation using linked list data structure. Treiber [9] also proposed an enhanced algorithm of Michael’s technique which was more memory efficient. Shalev et al[11] contributed to the growth of lock-free data structures by suggesting the implementation of lock free extensible hash table using linked list and skip list dictionary.

3. METHODOLOGY

3.1 The Link List Data Structure

Linked list as shown in Fig 2, is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of a data and a reference (in other words, a link) to the next node in the sequence; more complex variants add additional links. This structure allows for efficient insertion or removal of elements from any position in the sequence.

Linked lists are among the simplest and most common data structures. They can be used to implement several other common abstract data types, including lists (the abstract data type), stacks, queues, associative arrays, and S-expressions, though it is not uncommon to implement the other data structures directly without using a list as the basis of implementation. The principal benefit of a linked list over a conventional array is that the list elements can easily be inserted or removed without reallocation or reorganization of the entire data structure because the data items need not be stored contiguously in memory, while an array has to be declared in the source code, before compiling and running the program.

A non-blocking algorithm ensures that threads competing for a shared resource do not have their execution indefinitely postponed by mutual exclusion.

A non-blocking algorithm is lock-free if there is guaranteed system-wide progress regardless of scheduling; wait-free if there is also guaranteed per-thread progress. Obstruction-freedom is possibly the weakest natural non-blocking progress guarantee. All lock-free algorithms are obstruction-free.

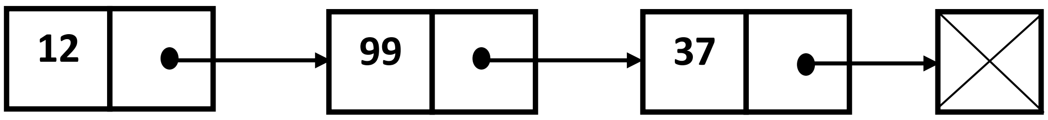

Fig 2: A linked list with two nodes

The definition of 'lock-free' and 'wait-free' only mention the upper bound of an operation. Therefore lock-free data structures are not necessarily the best choice for every use case. In order to maximize the throughput of an application one should consider high-performance concurrent data structures. Lock free data structures will be a better choice in order to optimize the latency of a system or to avoid priority inversion, which may be necessary in real-time applications. In general, it's advisable to consider if lock-free data structures are necessary or if concurrent data structures are sufficient. In any case it is advisable to perform benchmarks with different data structures for a specific workload.

3.2 System Design and Algorithms

One of the major problems one runs into when designing a lock-free linked list as stated earlier is the fact that one cannot easily control both the right pointer of the node that is to be deleted and the right pointer of its preceding node at the same time using a simple C&S primitive. It can be done with a stronger double compare- and-swap primitive [1]. To counter these challenges, one can use the technique of Harris's implementation [6]. Instead of applying Compare and Swap's (C&S) to the right pointer of the node, apply it to the successor field, which consists of a right pointer and a mark bit.

Another solution to the challenges is the introduction of back_link pointers, similar to the ones proposed by Valois [10]. This is then used to replace the mark bits, so that the successor field consists of a right pointer and a back_link. If the node has a null back_link pointer, it is not marked, otherwise it is. In the marking stage of a deletion, a process would set the back_link of the node to be deleted to point to the preceding node, using C & S on the node's successor filed. The simple introduction of back_links does not yield a data structure with good performance. One of the ways to do so is to ensure that whenever a back_link is set, it is pointing to an unmarked node. This is done by introducing one more bit known as the flag bit to reflect the status of the node. The algorithm for the three major routines: Search, Delete and Insert, are shown in Fig. 3, Fig.5 and Fig 6 respectively.

Search (Key k): Node

```
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
```

Fig 3: Search Routine

SearchFrom (Key k, Node *curr_node): (Node, Node)

```
1 next_node = curr_node.right
2 while (next_node.key <= k)
3 while (next_node.mark == 1 && (curr_node.mark == 0 || 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)
```

Fig 4: Search from Routine

Delete(Key k): Node

```
1 (prev_node, del_node) = SearchFrom(k – e, 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
```

Fig 5: Delete Routine

The SearchFrom routine is used to perform the searches in the data structure. This routine takes a key and a node as its arguments. It traverses the list starting from the specified node, looking for the first node with a key strictly greater than the specified key. This routine calls the SearchFrom routine in its first line, then used the first of the two nodes returned to determine if there is a node with key k in the list. The Insert routine (Fig. 6) first calls SearchFrom to find where to insert the new key. SearchFrom returns a pair of node pointers prev_node and next_node such that prev_node.key $\leq$ k $<$ next_node.key. Insert compares the key of prev_node to the new key it is trying to insert, and if they are not equal, it creates a new node since it is requirement that all the keys are to be unique, and enters the loop in lines 5-23, from which it can only exit if it successfully inserts the new node or another process inserts a node with the same key (lines 20-22). The Delete routine in Fig 5, performs a three-step deletion of the node. If the deletion is successful, Delete returns a pointer to the deleted node, otherwise it returns NO_SUCH_KEY. A successful deletion is linearized when the marking is performed. The Linked List Workshop application provides the three main list operations presented in Java. The three main routines are the same operations presented in The Linked List Workshop application. Fig. 7 shows how the Link List Workshop application looks when it’s started. Initially, there are sixty (60) data elements on the list generated automatically.

Insert (Key k, Elem e): Node

```
1 (prev_node, next_node) = SearchFrom(k, head)
2 if (prev_node.key == k)
3 return DUPLICATE_KEY
4 newNode = new node(key = k, elem = e)
5 loop
6 prev_succ = prev_node.succ
7 if (prev_succ.flag == 1)
8 HelpFlagged(prev_node, prev_succ.right)
9 else
10 newNode.succ = (next_node, 0, 0)
11 result = c&s(prev_node.succ, (next_node, 0, 0), (newNode, 0, 0))
12 if (result == (newNode, 0, 0)) // SUCCESS
13 return newNode
14 else // FAILURE
15 if (result == (*, 0, 1)) // failure due to flagging
16 HelpFlagged(prev_node, result.right)
17 while (prev_node.mark == 1) // possibly a failure due to marking
18 prev_node = prev_node.back_link
19 (prev_node, next_node) = SearchFrom(k, prev_node)
20 if (prev_node.key == k)
21 free newNode
22 return DUPLICATE_KEY
23 end loop
```

Fig 6: Insert Routine

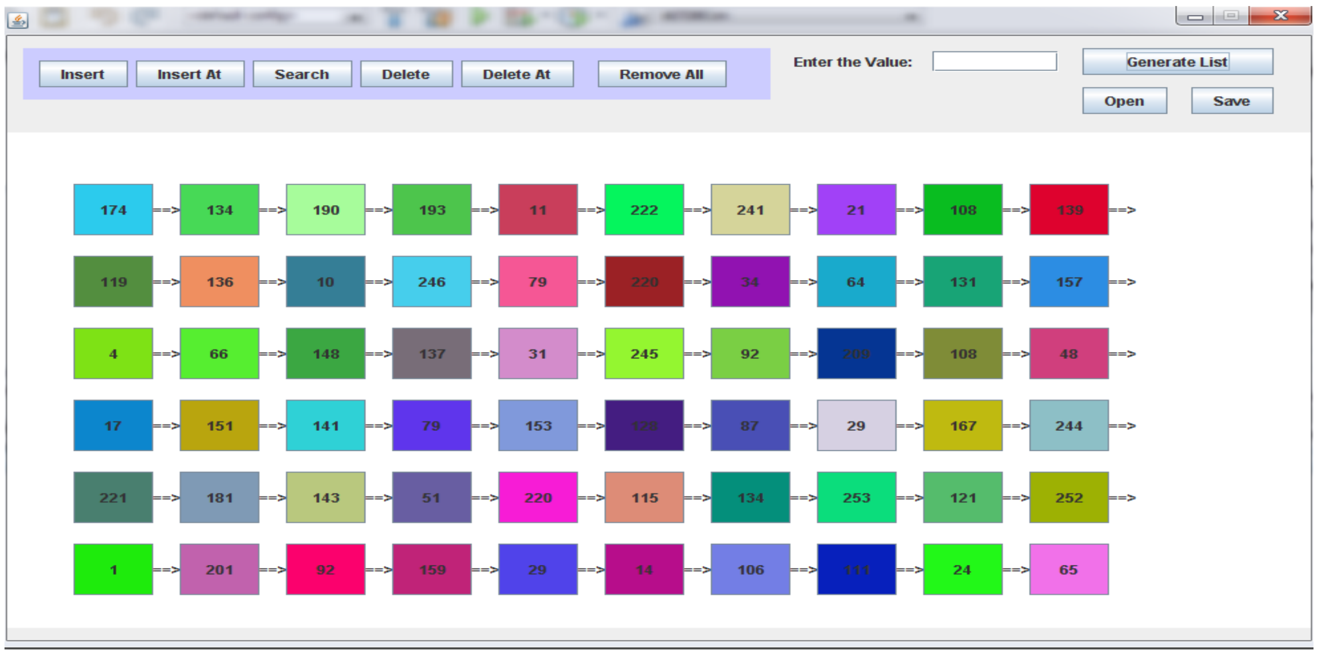

Fig 7: Linked List Workshop Application

4. ANALYSIS

This section is the description of the implementation, empirical evidence and the experimental proof of the correctness of the proposed algorithms. There are several invariants and definitions that hold throughout the execution and experimentation. The definitions of these terms are given below.

- **Def. 1**: A preinserted node is a node that was created, but has not yet been inserted into the list.
- **Def. 2**: A node is regular if it is unmarked, and it is not a preinserted node.
- **Def. 3**: A node is logically deleted if it is marked and has a regular node linked to it.
- **Def. 4**: A node is physically deleted if it is marked and there is no regular node linked to it.
- **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 with head being the first node and tail being the last node.
- **Inv 3**: For any logically deleted node, its predecessor is flagged (and unmarked), and its successor is not marked.
- **Inv 4**: For any logically deleted node, its back_link 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.back_link = m.
- **Inv 5**: No node can be both marked and flagged at the same time, i.e. there is no node n such that n.succ = (*, 1, 1).

Theorem 1: Invariant 5 always holds.
The invariant obviously holds for an empty list. When a new node is created, its successor pointer is set to be unflagged and unmarked (line 10 in Insert), and it does not change until the node is inserted into the list.

Theorem 2: Invariants 1-3 always hold.
Initially the list contains no keys and all the invariants obviously hold. The algorithms modify the data structure only by performing C&S operations or by setting back_links in line 1 of HelpFlagged routine. By other auxiliary Propositions 4, 7, 8, and 11 none of these four C&S operations can violate invariants 1-3, so they always hold.

Theorem 3: Inv 4 always holds.
This invariant obviously holds in the beginning when the list is empty. It can also be prove that if it holds before a modification, it will hold afterwards as well. It has been prove that the modification below cannot violate Inv 4.

The C&S in the Insert routine is result = c&s(prev_node.succ, (next_node, 0, 0), (newNode, 0, 0)). Immediately before the C&S prev_node is not flagged, and thus by Inv 3, next_node is not marked. Also, when the C&S is performed, newNode is not marked. Thus this C&S preserves Inv 4.

Theorem 4 (Delete correctness)
If an execution of the Delete (k) routine returns NO_SUCH_KEY (indicating an unsuccessful deletion), then for this execution we can choose a linearization point, at which there was no regular node with key k in the data structure. If an execution of the Delete (k) routine returns a pointer to a node (indicating a successful deletion), then for this execution we can choose a linearization point, at which this node became marked.

Theorem 5 (Insert correctness):
If an execution of the Insert(k, e) routine returns DUPLICATE_KEY (indicating an unsuccessful insertion), then for this execution we can choose a linearization point, at which there was a regular node with key k in the data structure. If an execution of the Insert(k, e) routine returns a pointer to a node (indicating a successful insertion), then the node's key is equal to k, and for this execution we can choose a linearization point, at which this node gets inserted, becoming a regular node.

Theorem 6 (Search correctness)
If an execution of the Search(k) routine returns NO_SUCH_KEY, then we can choose a linearization point, at which there was no regular node with key k in the data structure. If an execution of the Search(k) routine returns a pointer to a node, then the key of this node is k, and for this execution we can choose a linearization point, at which this node was a regular node.

5. CONCLUSION

This research is aimed at presenting a novel implementation of linked-lists which is non-blocking, linearizable and which is based on the compare-and-swap (CAS) operation found on contemporary processors. The presented algorithms (searching, insertion, and deletion) exploit a fine granularity synchronization strategy to significantly outperform existing algorithms, particularly for long linked lists and skip list data structures.

The use of COMPARE & SWAP to swing pointers is susceptible to the ABA problem. The solution relies on the careful memory management and in particular the use of two operations-SAFEREAD and RELEASE. Also the implementing the algorithm in a desired programming language such as an Assembly language required a high level of expertise.

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, heaps 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 the algorithm to decrease the number of conflicts.

6. REFERENCES

[1] Greenwald, M. 1999. Non-Blocking synchronization and system design. PhD Thesis. Stanford University, Technical Report STAN-CS-TR-99

[2] Lamport L. 1974 A new solution of Djikstra concurrent programming problem. Communication of the ACM,

[3] Herlihy, M.1991, Wait-free synchronization. ACM Transactions on Programming Languages and Systems, 124-149.

[4] Fischer, M.J, Lynch, N.A and Paterson M.S. 1985, Impossibility of distributed consensus with one faulty process. ACM Journal 32, 2 , 374 – 382.

[5] Herlihy, M.1993, A methodology for implementing highly concurrent data object. ACM Transactions on Programming Language and Systems Vol 15, 745-770.

[6] Massalin H and Pu, C. 1991, A lock-free multiprocessor OS Kernel. Technical Report CUCS-005-91, Columbia University.

[7] Harris, T.L. 2001, A pragmatic implementation of non-blocking list. Proceedings of the 15th International Symposium on Distributed Computing, 300 -314.

[8] Michael M. 2002, Safe Memory Reclamation for Dynamic lock-free objects using atomic reads and writes. Proceedings of the 21st Annual Symposium on Principles of Distributed Computing, 21 – 30.

[9] Treiber R.K.1986, Systems programming: Coping with Parallelism, Research report RJ 5118, IBM Almaden Research Center, 123.

[10] Valois John D.1995, Lock-free linked lists using compare-and-swap. Proceedings of the 14th ACM Symposium on Principles of Distributed Computing, 214-222.

[11] Shalev O and Shavit N, 2003, Split-Ordered Lists: Lock-Free Extensible Hash Tables, ACM Press.