International Journal of Computer Applications Technology and Research
Volume 4– Issue 3, 202 - 208, 2015, ISSN:- 2319–8656

Efficient & Lock-Free Modified Skip List in Concurrent Environment

Ranjeet Kaur
Department of Computer Science and Application
Kurukshetra University, Kurukshetra
Kurukshetra, Haryana

Pushpa Rani Suri
Department of Computer Science and Application
Kurukshetra University, Kurukshetra
Kurukshetra, Haryana

Abstract: In this era the trend of increasing software demands continues consistently, the traditional approach of faster processes comes to an end, forcing major processor manufactures to turn to multi-threading and multi-core architectures, in what is called the concurrency revolution. At the heart of many concurrent applications lie concurrent data structures. Concurrent data structures coordinate access to shared resources; implementing them is hard. The main goal of this paper is to provide an efficient and practical lock-free implementation of modified skip list data structure. That is suitable for both fully concurrent (large multi-processor) systems as well as pre-emptive (multi-process) systems. The algorithms for concurrent MSL based on mutual exclusion, Causes blocking which has several drawbacks and degrades the system's overall performance. Non-blocking algorithms avoid blocking, and are either lock-free or wait-free.

Keywords: skip-list, CAS, Modified Skip List, concurrency, lock-free

1. INTRODUCTION

Modern applications require concurrent data structures for their computations. Concurrent data structures can be accessed simultaneously by multiple threads running on several cores. Designing concurrent data structures and ensuring their correctness is a difficult task, significantly more challenging than doing so for their sequential counterparts. The difficult of concurrency is aggravated by the fact that threads are asynchronous since they are subject to page faults, interrupts, and so on. To manage the difficulty of concurrent programming, multithreaded applications need synchronization to ensure thread-safety by coordinating the concurrent accesses of the threads. At the same time, it is crucial to allow many operations to make progress concurrently and complete without interference in order to utilize the parallel processing capabilities of contemporary architectures. The traditional approach that helps maintaining data integrity among threads is to use lock primitives. Mutexes, semaphores, and critical sections are used to ensure that certain sections of code are executed in exclusion[1]

To address these problems, researchers have proposed non-blocking algorithms for shared data objects. Nonblocking methods do not rely on mutual exclusion, thereby avoiding some of these inherent problems. Most non-blocking implementations guarantee that in any infinite execution, some pending operation completes within a finite number of steps. Nonblocking algorithms have been shown to be of big practical importance in practical applications [2][3]

In the previous work we presented the concurrent access of Modified skip list with locking techniques[12] , as we have discussed the limitation due to locking method ,we present the lock free access of modified skip list data structure. This one is the initial efforts in this direction.

2. SKIP LIST AND MODIFIED SKIP LIST

Skip-lists [4] are an increasingly important data structure for storing and retrieving ordered in-memory data. SkipLists have received little attention in the parallel computing world, in spite of their highly decentralized nature. This structure uses randomization and has a probabilistic time complexity of O(logN) where N is the maximum number of elements in the list.

The data structure is basically an ordered list with randomly distributed short-cuts in order to improve search times, see Figure 1. In this paper, we propose a new lock-free concurrent modified skip-list pseudo code that appears to perform as well as the best existing concurrent skip-list implementation under most common usage conditions. The principal advantage of our implementation is that it is much simpler, and much easier to reason about. The original lock-based concurrent SkipList implementation by [13] is rather complex due to its use of pointer-reversal,

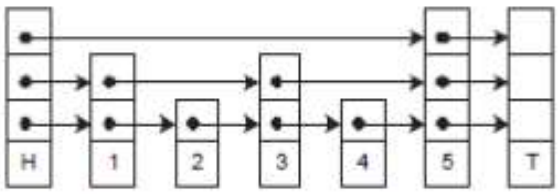

Figure:1Skiplist data structure.

While the search, insert, and delete algorithms for skip lists are simple and have probabilistic complexity of O (log n) when the level 1 chain has n elements. with these observations in mind [5] introduced modified skip list(MSL) structure in which each node has one data field and three pointer fields :left, right, and down. Each level l chain worked separate

doubly linked list. The down field of level l node x points to the leftmost node in the level l-1 chain that has key value larger than the key in x. H and T respectively , point to the head and tail of the level lcurrent chain. Below Figure 2 shows the MSL.

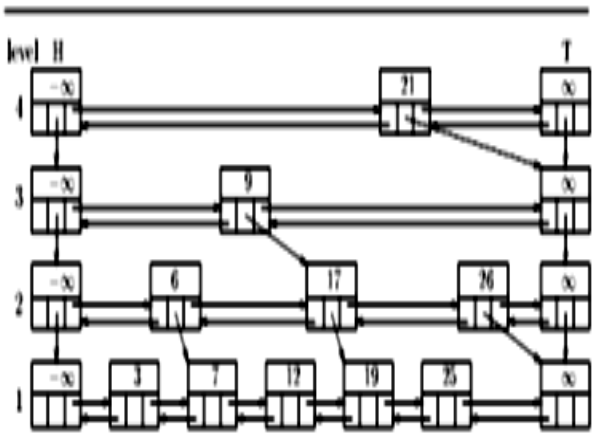

Figure: 2 modified skip list

3. CONCURRENT OPERATIONS ON MSL

This paper describes the simple concurrent algorithms for access and update of MSL. Our algorithm based on [11] . In this paper we present a lock-free algorithm of a concurrent modified skip list that is designed for efficient use in both pre-emptive as well as in fully concurrent environments. The algorithm is implemented using common synchronization primitives that are available in modern systems. Double link list is used as a basic structure of modified skip list.

A shared memory multiprocessor system configuration is given in Figure 3.Each node of the system contains a processor together with its local memory. All nodes are connected to the shared memory via an interconnection network. A set of co- operating tasks is running on the system performing their respective operations. Each task is sequentially executed on one of the processors, while each processor can run many tasks at a time. The co-operating tasks, possibly running on different processors, use shared data objects built in the shared memory to co-ordinate and communicate. Tasks synchronize their operations on the shared data objects through sub-operations on top of a cache-coherent shared memory. The shared memory may not though be uniformly accessible for all nodes in the system; processors can have different access times on different parts of the memory [6].

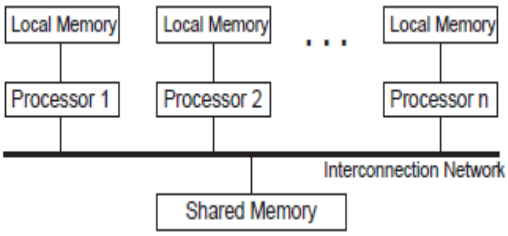

Figure: 3 Shared Memory Access

4. OUR ALGORITHM

We present a modified skip list algorithm in the context of an implementation of n set objects supporting three methods, search_node, insert_node, del_node:search_node (key) search for a node with key k equal to key, and return true if key found otherwise return false. Insert_node (d) inserts adds d to the set and returns true iff d was not already in the set; del_node (v) removes v from the set and returns true iff v was in the set, the below Figure 4 & 5 shows the field of a node. Using the strategy of [11]. To insert or delete a node from the modified skip list we have to change the respective set of prev and next pointers. These pointers have to be changed consistently, but it is not necessary to change them at once. According to Sundell & Tsigas [11] we can consider the doubly linked list as being a singly linked list with auxiliary information in the left pointers, with the right pointers being updated before the left pointers. Thus, the right pointers always form a consistent singly linked list and thus define the nodes positional relations in the logical abstraction of the doubly linked list, but the left pointers only give hints as to where to find the previous node. The down pointer of modified skip list is according its criteria.

```
Union link::word
<p, d> :< pointer to  node , Boolean>
Structure Node
Value: pointer to word
left: union link
right: union  link
down :union link
//local variables
Node, prev,prev2,next,next2:pointer to node
Last,link1:union link
```

Figure :4 local and global variables

The concurrent traversal of nodes makes a continuously allocation and reclamation of nodes, in such kind of scenario several aspects of memory management need to be considered, like No node should be reclaimed and then later re-allocated while some other process is traversing this node. This can be done with the help of reference counting. We have selected to use the lock-free memory management scheme invented by Valois [7] and corrected by Michael and Scott [8], which makes use of the FAA,TAS and CAS atomic synchronization primitives. The operation done by these primitives given below figure 5 & 6.

```
procedure FAA (address: pointer to word, number: integer)
atomic do
*address := *address + number;
```

Figure:5 FAA Atomic primitive

```
function CAS (address: pointer to word, oldvalue: word,
new value: word):boolean
atomic do
if *address = old value then
*address: = new value;
return true;
else
return  false;
```

Figure:6 CAS Atomic primitive

One problem, that arises with non-blocking implementations of MSL that are based on the linked-list structure, is when inserting a new node into the list. Because of the linked-list structure one has to make sure that the previous node is not about to be deleted. If we are changing the next pointer of this previous node atomically with CAS, to point to the new node, and then immediately afterwards the previous node is deleted then the new node will be deleted as well, as illustrated in Figure 7. This problem can be resolved with the latest method introduced by Harris [4] is to use one bit of the pointer values as a deletion mark. On most modern 32-bit systems, 32-bit values can only be located at addresses that are evenly dividable by 4, thereof ore bits 0 and 1 of the address are always set to zero. Any concurrent Insert operation will then be notified about the deletion, when its CAS operation will fail.

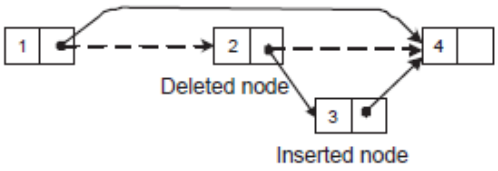

Figure: 7 Concurrent insert and delete operation can delete both nodes

One memory management issue is how to de-reference pointers safely. If we simply de-reference the pointer, it might be that the corresponding node has been reclaimed before we could access it. It can also be that bit 0 of the pointer was set, thus marking that the node is deleted, and therefore the pointer is not valid. The following functions are defined for safe handling of the memory management: shown in figure 8, 9 & 10.

```
function READ_NODE (node  **address):

/* De-reference the pointer and increase the reference
counter for the corresponding node. In case the pointer is
marked, NULL is returned */
```

Figure: 8 Memory management function

```
procedure RELEASE_NODE(node: pointer to Node)

/* Decrement the reference counter on the corresponding
given node. If the reference count reaches zero, then call
RELEASE_NODE on the nodes that this node has owned
pointers to, then reclaim the node */
```

Figure: 9 Memory management function

```
function COPY_NODE(node: pointer to Node):pointer

to Node /* Increase the reference counter for the
corresponding given node */
```

Figure: 10 Memory management function

4.1 search_node

Searching in MSL is accomplished by taking a value v and search exactly like a searching in sequential linked list, starting at the highest level and proceeding to the next down level, each time it encounters a node whose key is greater than or equal to v. the search process also save the predecessor and successor of a searched node v for further reference. In concurrent environment, while searching for a node in MSL a processes will eventually reach nodes that are marked to be deleted. It might be due to forcefully preemption of deletion process that invoked the corresponding operation. The searching operation helps the delete process to finish the pending work before continuing the search operation. However, it is only necessary to help the part of the delete operation on the current level in order to be able to traverse to the next node. The search operation traverses in several steps through the next pointers (starting from node1) at the current level until it finds a node that has the same or higher key value than the given key. See Figure 11.

```
pointer to node function search _node ( int   v)
{
node *t,*t_right, *save [maxlevel], *found_node
int i
t=COPY_NODE (head)
t_right=head→right
while (t! =NULL)
{
while( t→value<v)
{
If ( IS_MARKED (t→right))
t=help_del(t)
t_right=READ_NODE(t→right)
save[i]=t
}
if (t→value==v)
break;
else
{
t=t→left→down
i=i-1
}
}
found_node=t
return found_node
}
```

Figure:11

4.2 insert_node operation

The implementation of insert operation is shown in figure:12 With a search_node operation to find the node after which the new node should be inserted. This search phase starts from the head node at the highest level and traverses down to the lowest level until the correct node is found .if there exist already a node with key same as new node with value v, then insertion algorithm exit otherwise it searched for node with key value more than the new node value v. after inserting a new node, it is possible that it is deleted by a concurrent delete_node operation before it has been inserted completely at the particular level. The main steps for insertion algorithm are (I) after setting the new node's left and right pointers, atomically update the right pointer of the to-be-previous node, (II) atomically update the left pointer of the to-be-right node. Iii) atomically update the down pointer of nodes according to the value generated by random no. generator function. if it is more than current level, a new level is created for new node ,if it is less than current level new node is to be inserted in between of existing levels ,say it inserted at level l<current level. the down pointer of node at (l+1) level is to be changed accordingly ,as well as the down pointer of new node is to be updated with address of a node at level l-1 with value greater than the value at new node respectively.

```
function insert_node(key  int , value: pointer to word)
node *p,*t,*save[max],*t_right,*up,*found_node
k=randomlevel ()
temp,i =current_level
t=COPY_NODE (head)
while (t! =NULL)
{
while( t→key<key)
{
If ( IS_MARKED (t→right→value))
t=Help_Del(t)
save[i]=t
t_right=READ_NODE(t→right)
}
if (t→key==key)
break;
else
{
t=t→left→down
i=i-1
save[i]=t
}
}
found_node=t
new_node=create_node ( value)
node1=COPY_NODE(head)
If(k>temp)
{
//create new head and tail
h1=createnode(∞)
COPY_NODE(h1)
h1→left=null
h1→right=x
h1→down=h
RELEASE_NODE(H1)
t1=CreateNode(∞)
COPY_NODE (t1)
t1→left=x
t1→right=NULL
t1→down=t
RELEASE_NODE (t1)
new_node→left=h1
new_node→right=t1
if((save[k-1]→right→down)==NULL OR(save[k-1]→right→value>new_node→value))
new_node→down=save[k-1]→right
RELEA SE_NODE(new_node)
RELEASE_NODE(t1)
RELEASE_NODE(h1)
}
If(k<temp)//insert the new node after save[k]
{
prev=COPY_NODE(save[k])
next=READ_NODE(& prev→right)
While T do
{
If ( prev→right != next)
RELEASE_NODE(next)
next=READ_NODE(&prev→right)
continue
new_node→left=prev
new_node→tight=next
If CAS(&prev→right,next,new_node)//update the next pointer of Prev of to be inserted node
COPY_NODE(new_node)
break
back-off
}
While  T do //update the left pointer of next of to be inserted node
{
Link1= next→left
If (IS_MARKED(link1) || new_node→right!=next)
Break
```

```
if CAS(&next→left,link1,<new_node,F>)
COPY_NODE(new_node)
RELEASE_NODE(link1)
if(IS_MARKED(new_node)
prev2=COPY_NODE(new_node)
prev2=update_prev(prev2,next)
RELEASE_NODE(prev2)
break
back-off
}

RELEASE_NODE(next)
RELEASE_NODE(new_node)
//update the down pointer
if (k>1)
{
if((save[k-1]→right→down)==NULL OR(save[k-
1]→right→ value >new_node→ value)) then
new_node→down= save [k-1]→right

if ((save[k+1]→right→down)==NULL OR
(save[k+1]→right→ value <new_node→ value )) then
save [k+1]→right→down =new_node

RELEASE_NODE(new_node)
return true
}
 if (k==1)
{
if(save[k+1]→right→down)==NULL &&
(save[k+1]→right→value<x→value)) then

save[k+1]→right→down= new_node

new_node→down=NULL

}
RELEA SE_NODE(new_node)
return true
}  }
```

figure:12

4.3 Delete_node operation

The delete operation uses search operation to locate the node with key k, and then uses two stage process to perform the deletion. Firstly the node is logically deleted by marking the reference contained in it (delete_node→right→value). , secondly the node is physically deleted. The main steps of the algorithm for deleting a node at an arbitrary position are the following: (I) Set the deletion mark on the right pointer of the to-be-deleted node, (II) Set the deletion mark on the left pointer of the to-be-deleted node, (III) Set the deletion mark on the down pointer of the to-be-deleted node, IV) Atomically update the right pointer of the previous node of the to-be-deleted node,(V) Atomically update the left pointer of the right node of the to-be-deleted node. (VI) atomically update the down pointer for a node which was pointed by down pointer of to be deleted node.

```
Function delete_node (int key):boolean
{
node *delete_node,*prev,*succ,*up,*s[max]
temp, I =current_level
t=COPY_NODE (head)
while (t! =NULL)
{
while( t→key<key)
{
If ( IS_MARKED (t→right→value))
t=help_del(t)
save[i]=t
t_right=READ_NODE(t→right)
}
if (t→key==key)
break
else
{
t=t→left→down
i=i-1
save[i]=t
}
}
del_node=t
While T do
if  (del_node==NULL) then
RELEASE_NODE(del_node)
return NULL
linkk1=del_node→right
IF  IS_MARKED(link1) then
help_del(del_node)
RELEASE_NODE(node)
continue
if CAS( &del_node→right,link1<link1.p,T>) then
help_del(del_node)
next=READ_NODE(&del_node→right)
prev=update_prev(del_node,next)
RELEASE_NODE(prev)
release_node(next)
break
RELEASE_NODE(del_node)
```

```
Procedure mark_prev(pointer to node node)
while T do
 link1=node→left
 if IS_MARKED(link1) or
CAS(&node→left,link1,<link1.p,T>)then break
```

Figure:14

```
Pointer to node  function Help_Del(node: pointer to
Node)
 Mark_Prev(node);
last=NULL;
prev= READ_NODE(&node→left)
next= READ_NODE (&node→right)
 while T do
 if prev == next  then
break
 if IS_MARKED(next→right) then
mark_prev(next)
 next2:= READ_NODE (&next→right)
 RELEASE_NODE(next)
next:=next2
 continue
 prev2= READ_NODE (&prev→right)
 if prev2 = NULL then
 if last != NULL then
mark_prev(prev)
next2= READ_NODE (&prev→right)
if CAS(&last→right,<prev,F>),<next2,F>) then
 RELEASE_NODE(prev)
 else
 RELEASE_NODE(next2)
 RELEASE_NODE(prev)
prev=last
 last=NULL
 else
prev2=READ_NODE(&prev→left)
 RELEASE_NODE(prev)
prev=prev2
 continue
 if prev2 != node then
 if last !=NULL   then
RELEASE_NODE(last)
 last:=prev
 prev=prev2
 continue
 RELEASE_NODE(prev2)
 if CAS(&lprev→right, <node,F>,<next,F>) then
 COPY_NODE(next)
 RELEASE_NODE(node)
 break
 Back-Off
 if last != NULL then RELease_node(last)
 RELEASE_NODE(left)
RELEASE_NODE (next)
```

figure: 15

```
function update_prev(prev, node: pointer to
Node): pointer to Node
 last=null
 while T do
prev2:=READ_NODE(&prev→right)
 if prev2 = null then
 if last != null then
mark_prev(prev)
 next2:=READ_NODE(&prev→right)
 if CAS(&last→right,<prev,F>,<next2,F>) then
RELEASE_NODE (prev)
 Else
 RELEASE_NODE (next2)
 RELEASE_NODE (prev)
```

```
prev=last
 last=null
 else
 prev2=READ_NODE(&prev→left)
 RELEASE_NODE (prev)
prev=prev2
 continue
 link1=node→left
 if IS_MARKED(link1) then
RELEASE_NODE (prev2)
 break;
 if prev2!= node then
 if last!= null then
 RELEASE_NODE (last)
 last=prev
 prev:=prev2
 continue
 RELEASE_NODE (prev2)
 if link1→p = prev then
break
 if prev→right = node and CAS(
&node→left,link1,<prev,F>) then
COPY(prev)
 RELEASE_NODE (link1→p)
 if IS_MARKED(prev→left) then break
back-off
if last != NULL then
RELEASE_NODE (last)
 return prev
```

Figure:16

5. Correctness

In this section we describe the correctness of presented algorithm .here we outline a proof of linearizability [10] and then we prove that algorithm is lock-free. Few definitions are required before giving proof of correctness.

Definition 1 We denote with M_t the abstract internal state of a modified skip list at the time t. Mt is viewed as a list of values (v_1 ,----, v_n) .The operations that can be performed on the modified skip list are Insert (I) and Delete(D). The time t1 is defined as the time just before the atomic execution of the operation that we are looking at, and the time t2 is defined as the time just after the atomic execution of the same operation. The return value of $true_2$ is returned by an Insert operation that has succeeded to update an existing node, the return value of true is returned by an Insert operation that succeeds to insert a new node. In the following expressions that defines the sequential semantics of our operations, the syntax is S1 : O1; S2, where S1 is the conditional state before the operation O1, and S2 is the resulting state after performing the corresponding operation:

$$M_{t1}: I(v_1), \quad M_{t2} = M_{t1} + [v_1] \qquad (1)$$

$$M_{t1} = \theta : D() = NULL \ , M_{t2} = \quad \theta \qquad (2)$$

$$M_{t1}[\ v_1\ ] + M_1 : D() = v_1 , \quad M_{t2} = M_1 \qquad (3)$$

Definition 2 In order for an implementation of a shared concurrent data object to be linearizable [10], for every concurrent execution there should exist an equal (in the sense of the effect) and valid (i.e. it should respect the semantics of the shared data object) sequential execution that respects the partial order of the operations in the concurrent execution.

Definition 3 The value v is present (∃ i.M[i]=v) in the abstract internal state M of implementation , when there is a connected chain of next pointers (i.e. prev→link→right) from a present node in the doubly linked list that connects to a node that contains the value v, and this node is not marked as deleted (i.e. is_marked(node)=false)).

Definition 4 The decision point of an operation is defined as the atomic statement where the result of the operation is finitely decided, i.e. independent of the result of any suboperations after the decision point, the operation will have the same result. We also define the state-change point as the atomic statement where the operation changes the abstract internal state of the priority queue after it has passed the corresponding decision point.
We will now use these definitions to show the execution history of point where the concurrent operation occurred atomically.

Lemma 1 : *A insert_node operation (I(v)) , takes effect atomically at one statement.*
Proof: The decision, state-read and state-change point for an insert operation which succeeds (I(v)), is when the CAS sub-operation **CAS(&prev→right,next,new_node)** of insert operation succeeds. The state of the modified skip list was (Mt1 = M1) directly before the passing of the decision point. The state of the modified skip list after passing the decision point will be MT2 = [v] + M1 as the next pointer of the save[k] node was changed to point to the new node which contains the value v. Consequently, the linearizability point will be the CAS sub-operation in that line.

Lemma 2 : *A delete_node operation which fails (D*() =NULL*),takes effect atomically at one statement*
Proof: The decision point for a delete operation which tails (D() =NULL) is the check in line **if (del_node==NULL) then** . Passing of the decision point gives that the value v we are searching for deletion is not exist in modified skip list i.e (M_{t1} = NULL) .

Lemma 3 : *A delete_node operation which succeeds (D*() *=v), takes effect atomically at one statement.*
Proof: The decision point for a delete operation which succeeds (D() = v) is when the CAS sub-operation inline [**next=read_node(&del_node→right)]** succeeds. Passing of the decision point together with the verification in line [**if is_marked(link1) then].** Directly after passing the CAS sub-operation **in [if CAS(&del_node→right,link1<link1.p,T>) then]** (i.e. the state-change point) the to-be-deleted node will be marked as deleted and therefore not present in the Modified skip list ($\neg\exists i.M_{t2}[i] = v$). Unfortunately this does not match the semantic definition of the operation.

6. Conclusion

We introduced a concurrent modified Skiplist using a remarkably simple algorithm in a lock free environment. Our implementation is raw, various optimization to our algorithm are possible like we can extend the correctness proof. Empirical study of our new algorithm on two different multiprocessor platforms is a pending work. The presented algorithm is first step to lock free algorithmic implementation of modified skip list; it uses a fully described lock free memory management scheme. The atomic primitives used in our algorithm are available in modern computer system.

7. REFERENCES

[1] Eshcar Hillel. Concurrent Data Structures: Methodologies and Inherent, Limitations, PhD thesis, Israel Institute of Technology, 2011]

[2] P. TSIGAS, Y. ZHANG. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. Proceedings of the international conference on Measurement and modeling of computer systems (SIGMETRICS 2001), pp. 320-321, ACM Press, 2001.

[3] P. TSIGAS, Y. ZHANG. Integrating Non-blocking Synchronisation in Parallel Applications: Performance Advantages and Methodologies. Proceedings of the 3rd ACM Workshop on Software and Performance (WOSP '02), ACM Press, 2002.

[4] Pugh, W. Skip lists: A probabilistic alternative to balanced trees. Communications of the ACM 33, 6 (June 1990),

[5] S. Cho and S. Sahni. Weight-biased leftist trees and modified skip lists. ACM J. Exp. Algorithmics, 1998.

[6] H. Sundell and P. Tsigas. Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems. In Proceedings of the 17th International Parallel and Distributed Processing Symposium, page 11. IEEE press, 2003.

[7] J. D. VALOIS. Lock-Free Data Structures. PhD. Thesis, Rensselaer Polytechnic Institute, Troy, New York, 1995.

[8] M. MICHAEL, M. SCOTT. Correction of a Memory Management Method for Lock-Free Data Structures. Computer Science Dept., University of Rochester, 1995.

[9] T. L. HARRIS. A Pragmatic Implementation of Non-Blocking Linked Lists. Proceedings of the 15th International Symposium of Distributed Computing, Oct. 2001.

[10] M. Herlihy and J. Wing, "Linearizability: a correctness condition for concurrent objects," ACM Transactions on Programming Languages and Systems,vol. 12, no. 3, pp. 463–492, 1990.

[11] H. Sundell, P. Tsigas, Lock-free and practical doubly linked list-based deques using single-word compare-and-swap, in: Proceedings of the 8th International Conference on Principles of Distributed Systems, in: LNCS, vol. 3544, Springer Verlag, 2004, pp. 240–255.

[12] Ranjeet Kaur, Dr. Pushpa Rani Suri, Modified Skip List in Concurrent Environment, in : Proceedings of the IJSER, Aug 2014

[13] I. LOTAN, N. SHAVIT. Skiplist-Based Concurrent Priority Queues. International Parallel and Distributed Processing Symposium, 2000.