Wait-Free Made Fast

(Extended Abstract)

Yehuda Afek Dalia Dauber Dan Touitou
Computer Science Department,
Tel-Aviv University,
Israel 69978.

Abstract

An implementation of an asynchronous shared-data structure is wait-free if no adversarial scheduler can stop an *individual* operation on the data structure from making progress (that is the implementation can tolerate a fail-stop fault of any number of processes). An implementation is non-blocking if an adversarial scheduler cannot stop the *system* from making progress.

Existing wait-free implementations of shared data structures are slow when contention (processor concurrency) is low, namely, they have an $\Omega(n)$ term in their time complexity, where n is the total number of processes, however their time complexity for each operation is bounded. On the other hand, non-blocking implementations are fast when contention is low, but do not provide bounds on the time complexity of *individual* operations when contention is high. Neither seems to be satisfactory in practice. Inspired by Lamport's fast mutual exclusion algorithm [Lam87], we pose in this paper the question of implementing fast and wait-free data-structures. We then devise universal methods that combine the advantages of both non-blocking and wait-free approaches, i.e., fast when contention is low and guaranteed progress of any individual operation. The time complexity of the algorithms presented here is not a function of n, but a function of k the actual number of processes that access the object concurrently.

Two such methods are presented here; The first method, which is called *group update*, replaces the $O(n)$ term in the time complexity of wait-free implementations by an $O(\min(n, k \log k))$ term (or more precisely, $O(k \log k + s + kf)$, where s and f are the data structure size and single update complexity, respectively). The second method, which is called *individual update*, replaces the $O(n)$ term of wait-free implementations with an $O(k)$ term (or $O(kf \log f)$).

1 Introduction

In asynchronous shared memory machines a set of concurrent and asynchronous processors communicate and perform common computations by accessing a shared memory. A shared data structure may be accessed by many processes, concurrently. In particular the number of processes may be huge and much more than the number of processors [TUX90, MT93]. The implementation of the sharing of such objects has to guarantee that failed processors (or processes), even if in the middle of an update, do not block any other process from performing farther operations on the object [Her91, Her90, Her93]. Hence, the use of locks, or mutual exclusion, is prohibited. Herlihy [Her91] showed that in order to implement even simple data structures, such as a queue with a seek operation, without locking, one needs a universal building block such as compare-and-swap, or the load-linked and store-conditional pair of operations. Still the implementation of either non-blocking or wait-free data structures is a complicated and demanding programming task. Hence, in recent years universal methods to transform sequential, or blocking, implementations of data structures into either wait-free or non-blocking equivalent concurrent data structures were developed [Her90, Her93, Bar93, HM92, ST94, IR94, Ber91, TSP92, AF92, LaM94, AM94]. Specific non-blocking implementations of data structures were given in [Lam83, LS88, AW91, And92, IR93, MP91].

The basic principle of non-blocking implementations is that concurrent operations are computed concurrently, however only one is successfully committed, while all the others have to be re-computed and try again to commit. In wait-free implementations concurrent operations are also computed concurrently, but then the successful operations "help" the other processes to complete their operation before they try a new operation. The "helping" mechanism is the main cause for the $\Omega(n)$ time complexity per operation of all wait-free implementations that have been suggested so far. Thus, even if only one process accesses a small object alone, the time complexity of its access is at least $\Omega(n)$. This is unacceptable in practice since the number of processes could be huge compared to either the number of processors, or to the number of processes that usually access the object simultaneously. If wait-free methodologies are to be used in practice, then we claim that the issue of their time complexity must be addressed as we do here.

Here we raise the question of implementing the "helping" mechanism without the inherent $O(n)$ time complexity, and present two fast universal wait-free methods for implementing shared data structures that are based on the standard load-linked/store-conditional pair of operations. The *load-linked/store-conditional*pair of operations is supported on existing machines [Alp, Pow, MIP]. The time complexity of our implementations is $O(k)$ and $O(k \log k)$ instructions per operation per process, where k is the number of processes that access the object concurrently with that operation, which could be a dramatic improvement over the $O(n)$ complexity in many practical settings [TUX90, MT93].

Our first method, which is called *group update*, is based on an elementary building block which maintains an active-list of processes in a fast and wait-free manner. Processes wait-freely join and leave the shared list in time complexity which is a function of the list size rather than the total number of processes. For example, such a list is useful in the implementation of the entrance to the queue of a resource (e.g. printer) (which is usually implemented in the least fault-tolerant way of locking). Following Herlihy [Her90, Her93] once a process has joined the active list it applies the operations of all the processes in the list to the data structure. Although the time complexity of this method is not a function of n, its time complexity is $O(k \log k)$ and not our desired $O(k)$. Therefore we suggest a second method, called *individual update*. In this solution we introduce and implement a different "helping" mechanism which operates like a queue. Processes enter the queue in a fast and wait-free manner and ensure that every process gets to be at the head of the queue. The "helping" is realized here by requiring each process that progresses through the queue to also move forward certain other slower processes in the queue. When a process is at the head of the queue its operation is applied to the data structure. The time complexity of this method is indeed $O(k)$ instructions per operation. Although the complexity of the group update method is worse than that of the individual method ($O(k \log k)$ vs. $O(k)$) it is expected to out perform the individual method on cache based architectures because, it enables one process to apply the operations of a group of processes in a fast way. That is, several updates are applied in one copy of the object. Moreover, its worst case dependency on k, ($O(k \log k)$) is exhibited only under peculiar scenarios; Under most normal conditions its time complexity is $O(k)$.

More precisely, the time complexity of the *group update* is $O(s + kf(s) + \min(n, k \log k))$ instructions per operation, where s is the maximum size of the shared object in memory words during the execution of the operation and $f(s)$ is the sequential complexity of an update operation on the object. The time complexity of the *individual update* is $O(kf(s) \log(f(s)))$ instructions. Hence we have combined the fastness of non-blocking when contention is low, with wait-freedom, i.e., providing an upper bound on the time complexity when contention is high.

While the group update method might be more efficient in practice, its space complexity is worse than that of the individual method. The individual method requires $O(f(s))$ addresses in the memory of each process. The space complexity of the group update method is $O(n \log n + s)$ per process. Note that the knowledge of n is not assumed in the code of any process in our methods, except for the memory allocation part at the group update method (here we assume that for each method a global array of size n was pre-allocated).

The paper is organized as follows: After the model section, the group update method is described in Section 3, and the Individual update method is presented in Section 4. The proofs and most of the Lemmas statements were omitted from this extended abstract due to space limitations.

1.1 Related work

As shown by Herlihy in [Her88, Her91] any general implementation of shared data structures requires a universal synchronization primitive such as *compare-and-swap* or *load-linked/store-conditional*. Following the identification of universal primitives, Herlihy in [Her88] and Plotkin in [Plo89, Plo88] presented universal wait-free constructions of linearizable shared memory objects. These universal constructions were made more efficient by Jayanti and Toueg in [JT92]. All of these constructions were pretty much "theoretical", the first more practical approach was suggested by Herlihy in [Her90, Her93], where he describes a methodology for transforming sequential implementations of data structures into either non-blocking or wait-free implementations, using *load-linked/store-conditional*. Alemany and Felten in [AF92] and LaMarca in [LaM94] present techniques for improving the performance of Herlihy's non-blocking method in practice. In [TSP92], Turek Shasha and Prakash show how to translate locked based data structures into non-blocking ones. Most of the above methods either copy the entire data structure for each update or assume special operating system support. In

[Bar93], Barnes gives a method that avoids copying the whole data structure and allows disjoint accesses to the data structure to update simultaneously. Israeli and Rappoport in [IR94] simplify Barnes method by providing a non-blocking implementation of n-words *compare-and-swap* . In [HM92] Herlihy and Moss introduce the idea of transactional memory (TM), which is an architectural approach to support lock-free data structures. In this system the user writes transactions that operate on the data structure, and when these transactions are run in the TM environment they satisfy a lock-free fairness condition. Shavit and Touitou [ST94] provide software transactional memory system that achieves the same effect without the special hardware support required by Herlihy and Moss.

Specific lock-free implementations of some data structures are known. Lamport [Lam83] and Herlihy and Wing[HW87] give non-blocking algorithm for queues. Lanin and Shasha [LS88] give a non-blocking set manipulation algorithm. Anderson and Woll [AW91] design a non blocking algorithm for Union-Find of sets and Anderson [And92] analyzes non-blocking algorithms for the related problem of list compression. Massalin and Pu [MP91] implement non-blocking stacks, queues and linked lists in his Synthesis operating system kernel. Israeli and Rappoport [IR93] give a non-blocking implementation of a priority queue.

In [AM94] Anderson and Moir have also proposed to reduce the time complexity of operations on wait-free shared data structures. However, they reduce the time complexity by giving up the wait-freeness of the implementation and lowering the level of resiliency, i.e., assuming less powerful adversarial schedulers. Essentially they reduce the dependency of the time complexity on n to a dependency on a smaller number, l, by allowing at most l ($< n$) processes to access the data structure. This is achieved with the help of locking, that is, an l-exclusion and l-renaming algorithms which assume the existence of *compare-and-swap* and *fetch-and-increment*. However, Anderson and Moir's method is only l-resilient, since if all l locks are taken by faulty processes the system is deadlocked.

2 The Model

We assume the standard model of asynchronous shared memory machines as in [Her90, Her93]. Processors on that machine may execute instructions at a varying rate. Programs are executed by *threads* which are time-shared among the processors. There may be considerably more threads than processors and threads may suffer delays of any length at any time due to page faults or expired time quantum. The shared memory in the machine supports atomic read and write instructions and *load-linked/store-conditional*. The *load-linked*operation reads a shared variable and a subsequent *store-conditional*to that variable will succeed in changing its value only if no other process has modified that variable in the interim.

3 The Group Update Algorithm

The basic idea of the algorithm presented here is that processors cooperatively maintain a dynamic list of concurrently active processes. Ideally, this list contains the names of the processes that are currently in the process of updating the data structure. To apply an operation to the data structure a process goes through three major steps. First it adds itself to the list of active processes. Secondly, it applies its operation to the data structure and in order to maintain the wait-freeness property, it helps all the other processes in the active list to apply their operations. Finally, in the third step it removes itself from the active list. In what follows we explain how each step is performed in a fast manner. The key to the construction is the update of the active list.

The building block presented here is a fast way to maintain a (sorted) list in shared-memory multiprocessor system.

To enter the list each process performs at most $O(min(n, k \log k))$ instructions, where n is the total number of processes and k is the list size, i.e., the number of processes that are simultaneously active.

To maintain the active list we place it at the root of a complete binary tree of size n. To enter the active list, a process has to climb to the root of the tree, where a pointer to the list of active processes is held. Intuitively, in climbing the tree a process collects all the process-id's it sees on the way, until it reaches the root and updates the active list. However, if each process starts its climbing from a leaf the number of instructions each performs would be at least $O(\log n)$ independently of the number of other processes that were concurrently active, which would not make the implementation "fast".

To remove the logarithmic dependency of the time complexity on n, we devise a special mechanism. We modify the tree by installing a two fields record at each node; an owner field which is a process id, and a pointer field which points to a list of processes. Instead of starting at the leaves, a process starts traversing the tree from the root in a BFS order (level by level) until it captures one node in the tree. To capture a node the process has to successfully load-linked and store-conditional its name into a vacant owner field in that node. A key observation is that although this process takes place concurrently and asynchronously by several processes, if k processes overlap the operation of another process then this process will capture a node at depth at most $\log k$. After capturing a node, a process climbs up the tree by following the parent pointers from the captured node to the root. At each node on the way up, it updates a pointer to a list. Ultimately the list at each node v would contain the ids of all the processes that have captured nodes in the subtree rooted at v. Thus the complexity of an operation at a node is proportional to the number of its captured descendant nodes. In the sequel we prove that due to the asynchrony and concurrency the total cost of entering the list in this mechanism is $O(min(n, k \log k))$.

The three major steps mentioned above are thus refined to the following five steps that each process goes through: (1) announcing its operation, (2) capturing a node in the binary tree, (3) climbing the tree and updating the active-list, (4) performing the operations of the processes in the active list on the data structure, and (5) removing itself from the binary tree and the active-list. An overview of each of the five steps follows. The data structures and the code are given in Figures 1, 2, 3, 4, and 5.

Announcing: Each process has to announce the type of the operation and its parameters before it joins the tree. This information is written in a single-writer multi-reader record that each process keeps.

Capturing a node in the binary tree: Each process scans the tree in a breadth-first-search order. In each node the process reads the owner field using load-linked, and if it has no owner (the owner equals *vacant*) it tries to write its name in it using store-conditional.

Climbing the tree and updating the active-list: After a process succeeds in capturing a node, it climbs from that node up to the root. In each node on the path to the root the process iterates at most twice over a sequence of four operations. The sequence is (1) reading the pointer to the node's list using load-linked, (2) reading the lists in the node's children (ordinary reading), (3) merging the two child lists together with the two child nodes' owners to one sorted list in a new portion of the memory and (4) trying to swing the pointer at the current node, using store-conditional, to point to the updated list. The lists are held sorted, as required by the updating of the data structure step. As in [BND89, AAD$^+$90, Her93], if due to the writings of other processes, P fails to change the pointer twice then, it is guaranteed that its name is already on the updated list [1].

During the algorithm each list is produced by some process which wrote it and then successfully "swinged" the pointer at some node to point at the new list. Since the amount of memory used by our mechanism is bounded, processes will have to recycle the space used to store the different lists. This results in a scenario in which a slow process is reading a list while the list is being changed by a fast process. While a list is being written it is said to be in an inconsistent state. We need to guarantee that if a process reads a list at one of the children of node v (in order to update v's list) and the read list is being changed then the process will fail in swinging the pointer at v. This problem here stems from the fact that the list being read is not the list whose pointer is going to be updated (as in standard non-blocking algorithms that use pointer swinging).

We overcome this problem in the following straightforward way: Each process is given a single-writer-multi-reader memory section in which it writes the lists it produces. Since a process might be the last to update the list at any node of the tree, it maintains one list for each node, that is its space requirements are $n \log n$ (assuming each memory location can hold one id). However to solve the problem mentioned above we double its space so it maintains two lists for each node in the tree (these lists are called p_lists in the code). If a process was successful in updating the pointer to a list at node v, then in the next time it tries to update that pointer at v it will use the other portion of memory it has for v. Now, when a process P reads an inconsistent list that belongs to process Q and, that is pointed by a child of some node u, it is assured that after P read the pointer to the list (at u's child, which was after P load-linked the pointer at u), Q changed the pointer to point to its next copy. This changing was part of some operation of Q that must have included a successful store-conditional of the pointer at u, either by Q or by some other process, but must have been performed after P's load-linked and before Q returned to u's child. Therefore, it is guaranteed that P will fail in trying to update u's list pointer.

Another problem that reading an inconsistent list, may cause, is that the complexity of the read might not be linear in the list size. To avoid this each list terminates with some non-existing name (NULL_NAME in the code) and the writing of a value in the ith place on a list is preceded by the writing of NULL_NAME in the $i+1$st place.

Updating the data structure: This is done as in Herlihy's algorithm [Her93], except that a process helps only the processes that appear on the root's list and the root's owner. The data structure is held in a portion of the shared memory which is pointed by a shared pointer. Each process reads this pointer using load-linked, copies the data structure to a new portion of the memory, applies all the operations of the processes that appear on the list and have not been performed yet to

[1] Sometimes, in practice, the *store-conditional* operation may fail spuriously (due to operating system intervention). If such are assumed, a process must explicitly verify that its name appears in the list. This verification is omitted from this extended abstract.

the data structure in that portion and, finally, tries to update the pointer, using store-conditional, to point to the new structure. The responses of the operations are kept alongside the data structure and are copied with it. The responses are held as a list of processes' names and their responses. Only processes that have not completed their operations appear on the list, so there is no need of copying an array of size n. When applying operations to the data structure, every process that appears on the root's list is searched in the responses' list. If it is on the responses' list, its response is taken as it is to the updated list. If it is not on this list, its operation (as appears in the announcing array) is applied to the data structure, and the response for that operation is added to the responses' list. All the members of the responses' list that do not appear on the root's list are removed from it. The responses' list, as well as the root's list, is held sorted, so that the search does not change the time complexity. Again, when a process fails to change the data structure's pointer twice, its operation must have been performed. Note: In order to enable a process to read its response, we have to place it in a fixed place in the list so it will not be changed during copying. This is done by using pointers.

Getting out of the active-list: At the end of its operation, a process has to remove its name from the tree, first in order to prevent unnecessary operations by other processes and second, to make sure the response it reads in the next operation is a new one. To do so, the process starts with the node it previously captured and clears the owner field, and then climbs up the tree as before while updating the lists of processes in each node it encounters. When the process gets to the root its name is no longer on the tree's lists. Finally, it performs once again the updating of the data structure, applying all the waiting operations to the data structure, so that at the end of this stage, there is no occurrence of the process name neither on the tree's lists nor on the responses' list held alongside the data structure.

3.1 Correctness and complexity

In order to prove the correctness of the implementation we must show that the implementation satisfies the *linearizability* property [HW87], i.e that every run of the algorithm has an equivalent sequential legal history of the implemented object that is an extension the natural partial order of the run. The Lemmas and proofs were omitted due to space limitations.

Theorem 1 *The time complexity of an update operation on the data structure which is concurrent with the operations of k other processes is $O(\min(n, k \log k) + kf(s) + s)$ where s is the size of the data structure and $f(s)$ is the sequential complexity of an operation on it.*

```
NODE record of
   ptr_to_list: pointer to LIST
   owner: integer
   index: integer                         {used as index in p_lists}

{ shared memory variables }
p_lists[1..n][1..2]: array of LISTs               { for each process }
          { each process has 2 lists for every node in the tree }
nodes[1..n]: array of NODEs
      { nodes[i] is the ith node in the tree, in the BFS order. }

{ local memory }
curr_list[1..n]: array of integers     {for each node, holds 1 or 2}
my_node: integer
```

Figure 1: Group Update - Shared and local variables

```
procedure update_data_structure (operation)              { for Pi }

   announce (operation)
   update_tree (TRUE)
   apply_operations_to_the_ds (TRUE)
   get_response_from_list ()
   update_tree (FALSE)
   apply_operations_to_the_ds (FALSE)
         { in order to remove Pi's response from the d.s. list }
```

Figure 2: Group Update - Main procedure

```
procedure update_tree (insert: boolean)                  { for Pi }
{insert is TRUE if Pi wants to add its name to the list,          }
{and it is FALSE if Pi wants to remove its name from the list     }

  node: NODE
  new_list: pointer to LIST
  ind: integer

  if insert then
     my_node := get_captured_node ()
  else                          {Pi is removing itself from the tree.}
     nodes[my_node].owner := vacant

  node := nodes[my_node]
  while node ≠ root do
     node := parent (node)
     ind := node.index  { index in the p_lists, curr_list arrays }
     new_list := address of p_lists[ind][curr_list[ind]]
     for i=1 to 2      {Instead of iterating twice, a process could
                        check explicitly its appearance in the list.}
        LL (node.ptr_to_list)
        merge (left_child(node), right_child(node), new_list)
        if SC (node.ptr_to_list, new_list) then
              curr_list[ind] := toggle curr_list[ind]
     end for
  end while
```

Figure 3: Group Update - Updating the tree

```
function get_captured_node () returns integer
{BFS the tree to find a node to insert P_i's name in its owner field.}
   ind, process_name: integer

   ind:=1
   forever do
      process_name := LL (nodes[ind].owner)
      if process_name = vacant then
         if SC (nodes[ind].owner, i) then return ind
      ind := ind +1
   end forever

procedure merge (left: NODE, right: NODE, dest: pointer to LIST)
{Merges the two sorted lists pointed by left.list and right.list    }
{ and the two processes' names in left.owner and right.owner        }
{into one sorted list, pointed by 'dest'.                           }
   ind, name: integer

   ind := 1
   dest[ind] := NULL_NAME
   while one of the children's lists or owner fields contains values do
      name := min name of the four sources which is neither a
               NULL_NAME nor vacant
      dest[ind+1] := NULL_NAME
      dest[ind] := name
      ind := ind + 1
   end while
```

Figure 4: Group Update - Procedures for updating the tree

```
procedure apply_operations_to_the_ds (insert:Boolean)

   while TRUE do
      if insert
         then if P_i is on the responses' list pointed
                                         by data_structure_ptr
            then return
         else                              { insert is FALSE }
            if P_i is not on the responses' list pointed
                                         by data_structure_ptr
               then return
      old_ptr := LL (data_structure_ptr)
      copy (old_ptr, new_ptr)
      for each process that appears in the tree's root's list or owner
         if its response appears in the responses' list
            then take the response as it is
            else apply its operation to the copy in new_ptr
      { all other processes in the responses' list are removed }
      SC (data_structure_ptr, new_ptr)
   end while
```

Figure 5: Group Update - Updating the data structure

The space complexity of the active list maintenance is $O(n \log n)$ per process. Together with the updating of the data structure the space complexity is $O(n \log n + s)$ for each process.

4 Individual Update

In this section we present a universal implementation of wait-free data structures in which the time complexity of each operation is $O(k)$. As in the group update method we introduce here a new building block that serializes the accesses to the data structure. To apply an operation to the data structure a process would first enter that building block, which can be viewed as a queue. The process that gets to the head of the queue applies its operation and finally in a third step removes itself from the queue building block.

The queue building block is a particular implementation of the "helping" mechanism. Each process that enters the queue building block moves thru it towards the head of the queue. While doing so it helps processes that stand in its way to complete their operation. The implementation of this building block ensures that the overhead incurred by each process while going through the queue building block and out of it is at most $O(k)$ machine instructions where k is the number of active processes in the building block during its operation. In addition, while going through the queue a process might help at most $O(k)$ other processes to apply their operations to the data structure.

The code of the Individual update method is given in Figures 6, 7, 8, and 9. The queue building block is based on a full binary tree, *ClimbTree*, implemented in an array of size n. Each node in the tree is a pair; the owner of the node (as in the group update method) and a direction boolean value which may be either $left$ or $right$. All nodes are initialized to *vacant*. Each process P_i, keeps an operation record Rec_i composed of (1) a *Desc* field which contains the description of the operation to be applied on the object, (2) a *Done* field which determines whether the update has been completed, (3) a *Version* field incremented by P_i at the end of every update [2] and other synchronization fields described later.

A process initiates an update on the object (procedure *UpdateDataStructure* in Figure 6) by writing into its operation record the action to be executed on the data structure and its arguments and then getting into the queue. To enter the queue processes first capture a node in the tree exactly as in the group update method. Following that, processes "climb" (procedure *Climb* in Figure 6) from the captured node to the root. Climbing

[2] If a *Validate* operation as in [IR94] is available on the architecture (which determines if some location has been updated since the last loadlinked operation) then it is possible to eliminate the Version field.

in the tree is achieved by capturing all the nodes along the path from the captured node to the root. The process that owns the root of the tree is considered to be at the head of the queue, and its operation is applied to the data structure. Capturing a node along the climbing path differs from capturing the starting node: the process first reads the contents of the node. If the node already contains the address of its record, the process continues recursively to the parent of the node. If the node is empty the process tries to write its own record in the node and right or left depending on whether it came from the right or left child. If the node contains the address of an update already done, the process overwrites the node with either its record or its sibling's record. Otherwise, the process helps the owner of the node. In helping another process, a process recursively performs all the work necessary to complete the operation of the other process, including climbing, helping other processes and accessing the data structure.

After accessing the data structure the process that owns the root node releases all the ownerships it has in the tree. To ensure fairness and the efficiency of the queue, when a process releases the ownership of node v (procedure *Release*- Figure 7) it makes the owner of one of v's child nodes the new owner of v. The decision of which of the child owners will become the new owner is a critical point in the algorithm. If the releasing owner also owns the left (right) child node then it gives the ownership to the right (left) child and set the direction boolean variable to right (left). Otherwise, if the releasing process does not own either of the child nodes, it toggles the direction variable and gives the ownership to the owner of the child node that is pointed by the new direction value. Thus, processes that climb through a node during the algorithm, come one from its left child, one from its right child in alternation.

The above mechanism ensures that in capturing node v at depth d, a process helps at most 2 other processes: a process that *starts* its climbing at v, and a process that climbs from the other child of v. The naive recursive equation that captures the cost of climbing from a node at depth d, is thus $T(d) = 3T(d-1)+C$ which would result in an $O(n^{\log_2 3})$. However a careful counting shows that the recursive function is $T(d) = 2T(d-1) + C$.

4.1 Applying the operation to the data structure

Assume that the implemented object has size s and the sequential complexity of updating the object is $f(s)$. We propose two different methods to execute the update. The first is suitable for small objects like *Fetch&Add* in which $f(s) \geq s$. This method is a variation of the copying method of Herlihy [Her90, Her93] which performs well for small size objects in network based architec-

```
UpdateDataStructure(operation)
 Initialize(Rec_i,operation)
                      { capture a node to start the climbing from }
 start_location := 1                        {1 is the root's location.}
 forever do
   if LL(ClimbTree[start_location]) = vacant then
     if SC(ClimbTree[start_location],(Rec_I,FROM_LEFT)) then
       exit forever
   start_location := start_location+1
 end forever
                                  { climb and execute the update }
 Climb(Rec_i,Parent(start_location),Rec_i →Version,start_location)
                                       { Release the climbing path }
 x := start_location
 repeat
    { Release procedure is called twice to avoid race condition }
                               { with some other climbing process. }
   Release(x)
   Release(x)
   x := Parent(x)                          { Parent(x) ≜ ⌊x/2⌋}
 until x=0                   { if x=o we have traversed the root }
                                          { Increment Rec_i version}
 Rec_i →Version := Rec_i →Version+1
 return Rec_i →ReturnedValue
end UpdateDataStructure
```

```
Climb(rec,loc,ver,from_where)
  {Procedure Climb ensures that the operation recorded in rec,}
    {with version # ver, captures a path from vertex loc to the}
                                           {root and is executed.}
                          { If the root of the tree (loc=1) has }
                          { been captured, execute the update}
 if loc= 0 then Execute(rec,ver)
 else                      { Determine the climbing direction }
   if from_where = Left(loc) then                  {Left(x)≜ 2x}
     direction := FROM_LEFT
   else
     direction := FROM_RIGHT
   forever do
     (owner,owner_direction):= LL(ClimbTree[loc])
                      { if the update has been executed leave }
     if rec→Done or (rec→Version≠ ver) then return
     case owner of
       vacant:                          { Try to capture the node }
         SC(ClimbTree[loc],(rec,direction))
       rec: { If the node is already captured continue climbing }
         Climb(rec,Parent(loc),ver,loc)
         return
       otherwise:
           { The node is already captured by another update}
         if owner→Done then
                   {If the owner has been executed, release it }
           Release(loc)
         else
     {The owner has not yet been executed, help it recursively}
           owner_version := owner→Version
           if ClimbTree[loc] = (owner,owner_direction)
             Climb(owner,Parent(loc),owner_version,loc)
     end case
   end forever
end Climb
```

Figure 6: Individual Update- Climbing

```
Release(loc)
  owner,owner_direction:= LL(ClimbTree[loc])
  if owner= vacant or (not owner→Done) then return
  l := ClimbTree[Left(loc)]
  r := ClimbTree[Right(loc)]                      {Right(x) ≜ 2x + 1}
                        { If the content of the node is taken from the}
                 { left then give priority to the right and vice versa}
  if owner_direction=FROM_LEFT then
    if r≠ vacant then
      SC(ClimbTree[loc],(r,FROM_RIGHT)
    else
      SC(ClimbTree[loc],(l,FROM_LEFT)
  else
    if l ≠ vacant then
      SC(ClimbTree[loc],(l,FROM_LEFT))
    else
      SC(ClimbTree[loc],(r,FROM_RIGHT))
end Release
```

Figure 7: Individual Update- Releasing

tures [ST94]. The second method is intended for large data structures as priority queues in which $f(s) \leq s$

Copying Method. In the copying method (procedure *Execute* in Figure 8), the object is addressed through a pointer *OP*. Since all the helping processes try to apply the same update to the data structure, we must ensure that this update is applied only once. Therefore, every helping process first agree on the content of OP at the beginning of the execution. After the process has agreed, it copies the object to a private copy and sequentially updates it. Then, the process agrees on the returned value of the update. At this time, the process tries to update the pointer to the object only if the update has not yet been completed and the contents of OP equals to the address the processes have agreed on. Finally, the process marks the update as executed by setting Done field to True.

Agree on Changes method. The idea of this method (procedure *Execute* in Figure 9), is that instead of copying the whole object, the helping processes rather agree on the changes that should be done on the object in order to achieve the update. For this purpose, every process's record has the following additional fields: *SizeOfChangeNeeded,* an integer on which processes agree on the size of the update. *ChangeNeeded,* a vector on which the processes agree on the changes needed in the object in order to complete the update. Every element in ChangeNeeded is a triplet which contains an address, an old value read from this address at the beginning of the update and a new value which should be written in this address. In order to execute an update a process first executes the update in a caching mode [Bar93], in which every time the process reads a location for the first time it inserts it to a balanced tree it maintains in its private memory. Any subsequent access to this location, (reading or writing) is executed through the tree. After it completes the cache update, the process agrees on (1) the value that should be returned, (2) the number of changes, and (3) the changes that the structure should undergo. Finally it updates the data structure according to the changes agreed upon and marks the update as done.

```
Execute(rec,ver)
  add:= LL(OP)
    { Agree on the address of the current version of the object }
  if LL(rec→OldPointer)= Null and rec→version = ver then
    SC(rec→OldPointer,add)
  old := rec→OldPointer
  Copy(old,new)
          { Execute the changes on a private copy of the object}
  returned_value:= ExecuteInPrivate(rec→Desc,new)
                                       { Agree on the returned value }
  if LL(rec→ReturnedValue)≠ Null and rec→Version= ver then
    SC(rec→ReturnedValue,returned_value)
         { Try to replace the object with the private version }
  if add = old and (not rec→Done) and rec→Version= ver then
    SC(OP,new)
  done := LL(rec→Done)
                                    { Declare the update as executed}
  if (NOT done ) and rec→Version = ver then
    SC(rec→Done,TRUE)
end Execute
```

Figure 8: Individual Update- Copying Method

4.2 Correctness and Analysis of the Individual Update

In this algorithm any update U of the object is related to some record Rec_i and an instance number (the content of the Version field). Therefore, we define the process that has started the execution of U (which owns Rec_i) the *initiator* of U. We say that U has been *completed* when the Done field of its record has been set to TRUE. All the processes executing the Climb procedure with U as parameter (i.e with $rec = \text{Rec}_i$ and ver equivalent to the instance number of U) are defined to be the *helping* processes of U. The initiator and the helping processes are the *participating* processes of U. The subset of the participating processes of U which have also executed Execute, are the *executing* processes of U. The node that the initiating process has caught before climbing in the tree is the *starting* node of U denoted S_u . The path between S_u and the root of the tree is defined as the *climbing path* of U. Given an update U we say that U has been *erased* when the Version field has been incremented by the initiator of U.

```
Execute(rec,ver)
  desc= rec→Desc
                      { Execute the update on a cached mode. }
        { and determine the changes needed and their number.}
  (size,return_value,change_needed):= ExecuteInCacheMode(desc);
  x := LL(rec→ReturnedValue);
                                    { Agree on the returned value }
  if x= Null and rec→Version = ver then
    SC(rec→ReturnedValue,return_value)
  x := LL(rec→SizeOfChangeNeeded)
  if x= Null and rec→Version = ver then
                      { Agree on the number of changes needed }
    SC(rec→SizeOfChangeNeeded,size)
  x := rec→SizeOfChangeNeeded
                                    { Agree on the changes needed }
  for i:= 1 to x do
    y := LL(rec→ChangeNeeded[i])
    if y ≠Null and rec→Version = ver then
      SC(rec→ChangeNeeded[i],change_needed[i])
  end for
        { Execute the changes according to the values agreed }
  for i:=1 to x do
    (add,old,new) = rec→ChangeNeeded[i]
    y:= LL(add)
    if y=old and (not rec→Done) and rec→Version = ver
    then SC(add,new)
  end for
                                  { Declare the update as executed }
  done := LL(rec→Done)
  if (NOT done) and rec→Version = ver then
    SC(rec→Done,TRUE)
end Execute
```

Figure 9: Individual Update - Agree on Changes Method

4.3 Correctness

Lemma 1.1 *The implementation is linearizable.*

Sketch of proof: The proof of this lemma is based on the following invariants:

1. Given an update U, from the moment that the first executing process of U starts Execute, until U is completed, all the nodes along the climbing path of U contain U's record address.

2. Given an update U, after U has been erased and before the initiator of U starts another operation, all the nodes in the climbing tree contain values which differ from U's record address, and no participating process of U will update the shared data structures on the behalf of U anymore.

3. When an update is completed, the object has been updated, and no executing process of U will make anymore changes to the object on U's behalf.

∎

4.4 Complexity

Given an update U, we define as *invaders* of U all those updates that have captured their starting nodes on the path from S_u to the root after U has captured S_u.

Lemma 1.2 *For every update U, the number of steps executed by the initiator until U is completed is $O(2^d P_E)$, where d is the depth of the starting node of U and P_E is the complexity of the Execute procedure.*

Sketch of proof: Due to the releasing policy, a process which executes an update U on a node v with depth d, will have to execute Climb at most three times on the parent of v in order to have its update completed: once for an eventual invader, once for its sibling node and once for itself. However, after it returns from the first Climb a process will never have to help an invader anymore until its Climb is completed. The reason is that in between the time the process has started the first Climb and its return from it, there was a time in which the climbing path of the helped update has contained only the address of the helped update's record (by Invariant 1 of Lemma 1.1). From this moment, again due to the releasing policy, there will never be any vacant node on U's climbing path and no invader will ever be able to capture a starting node on this path. We denote by $T(d)$ the number of steps necessary to complete a Climb operation from a node with depth d when invaders may appear on the climbing path, and by $T'(d)$ the number of steps necessary to complete a Climb operation from depth d when invaders may not appear on the climbing path. Then,

$$T(0) = T'(0) = P_E \tag{1}$$

$$T'(d) = 2T'(d-1) + C \tag{2}$$

$$T(d) = T(d-1) + 2T'(d-1) + C \tag{3}$$

From equations (1,2) it follows that $T'(d) = O(2^d P_E)$ and therefore from equation (3) it follows that:

$$T(d) = \sum_{i=1}^{d} O(2^i P_E) = O(2^d P_E)$$

∎

Theorem 2 *Any process in the copying method updates an object of size s and sequential update complexity $f(s)$ in $O(k(s+f(s)))$ primitive steps, where k is the number of processes that concurrently update the object.*

The space requirements of this method are: $O(s)$ per process, and $O(n)$ for the tree.

Theorem 3 *Given an object with size s and sequential update complexity $f(s)$, then with the agree on changes method, any process updates the object in $O(kf(s)\log(f(s)))$, where k is the number of processes that update the object concurrently.*

The $\log(f(s))$ term follows from the cache execution: every process which executes the update in the cache mode keeps the locations in a balanced tree. This tree contains at most $f(s)$ values. The space requirements of this method are: $O(f(s))$ per process, $O(n)$ for the climbing tree, and $O(s)$ for the data structure.

5 Conclusions

Our methodology suggests a practical and universal approach for the construction of highly resilient and highly concurrent shared data structures. Using previous methodologies to adapt various algorithms that use different synchronization primitives, such as, swap, or fetch-and-add to run on machines that support only one primitive such as *load-linked/store-conditional* may increase these algorithms' complexity by a factor of $O(n)$. However, using our method, the increase in the time complexity due to the adaptation is minimized. For example, if the fetch-and-inc in the snapshot algorithm of Riany, Shavit and Touitou [RST95] is adapted using previous methods, the cost of the snapshot is $O(n^2)$, while, if it is adapted using our methods, its cost remains $O(n)$.

References

[AAD+90] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. In *Proc. of the Ninth Ann. ACM Symp. on Principles of Distributed Computing*, pages 1–13, August 1990. Also *Journal of the ACM*, to appear.

[AF92] J. Alemany and E. W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proc. of the 11th ACM Symposium on Principles of Distributed Computing*, pages 124–134, August 1992.

[Alp] Alpha. *Alpha system reference manual.* DEC.

[AM94] J. H. Anderson and M. Moir. Using k-exclusion to implement resilient, scalable shared objects. In *Proc. of the 13th ACM Symposium on Principles of Distributed Computing*, pages 141–150, August 1994.

[And92] R. J. Anderson. Primitives for asynchronous list compression. In *Proceeding of the 4th ACM Symposium on Parallel Algorithms and Architectures*, pages 199–208, 1992.

[AW91] R. J. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In *Proc. of the 23rd ACM Symp. on Theory of Computation*, pages 370–380, May 1991.

[Bar93] G. Barnes. A method for implementing lock-free shared data structures. In *Proc. of the 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.

[Ber91] B. N. Bershad. Practical considerations for lock-free concurrent objects. Technical Report CMU-CS-91-183, Carnegie Mellon University, September 1991.

[BND89] A. Bar-Noy and D. Dolev. Shared memory vs. message-passing in an asynchronous distributed environment. In *Proc. of the 8th Ann. ACM Symp. on Principles of Distributed Computing (PODC)*, pages 307–318, August 1989.

[Her88] M. P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proc. of the 7th ACM Symp. on Principles of Distributed Computing*, pages 291–302, 1988.

[Her90] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proc. second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 197–206, March 1990.

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

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

[HM92] M. Herlihy and E. J. Moss. Transactional memory: Architectural support for lock-free data structures. Technical Report CRL 92/07, Digital Equipment Corporation, December 1992.

[HW87] M. Herlihy and J. M. Wing. Axioms for concurrent objects. In *Proc. 14th ACM Symp. on Principles of Programming Languages*, pages 13–26, January 1987.

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

[IR93] A. Israeli and L. Rappoport. Efficient wait-free implementation of a concurrent priority queue. In *Workshop of Distributed algorithms*, pages 1–16, September 1993.

[IR94] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proc. of the 13th ACM Symposium on Principles of Distributed Computing*, pages 151–160, August 1994.

[JT92] P. Jayanti and S. Toueg. Some results on the impossibility, universality, and decidability of consensus. In *Proc. of the 6th Int. Workshop on Distributed Algorithms: Lecture Notes in Computer Science, 647*, pages 69–84. Springer Verlag, November 1992.

[Lam83] L. Lamport. Specifying concurrent program modules. *TOPLAS*, 5:190–222, 1983.

[Lam87] L. Lamport. A fast mutual exclusion algorithms. *ACM Trans. on Computer Systems*, 5(1):1–11, February 1987.

[LaM94] A. LaMarca. A performance evaluation of lock-free synchronization protocols. In *Proc. 13th ACM Symp. on Principles of Distributed Computing*, pages 130–140, August 1994.

[LS88] V. Lanin and D. Shasha. Concurrent set manipulation without locking. In *Proceedings of the 7th ACM Symposium on Principles of Database Systems*, pages 211–220, March 1988.

[MIP] MIPS. *The MIPS RISC Architecture.* MIPS.

[MP91] H. Massalin and C. Pu. A lock-free multiprocessor os kernel. Technical Report CUCS-005-91, Columbia University, 1991.

[MT93] M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45:137–142, 1993.

[Plo88] S. A. Plotkin. *Chapter 4: Sticky Bits and Universality of Consensus.* PhD thesis, M.I.T., August 1988.

[Plo89] S. A. Plotkin. Sticky bits and universality of consensus. In *Proc. of the 8th ACM Symp. on Principles of Distributed Computing*, pages 159–175, Edmonton, Alberta, Canada, August 1989.

[Pow] PowerPc. *Power PC. Reference manual.* IBM.

[RST95] Y. Riany, N. Shavit, and D. Touitou. Towards practical snapshots. In *Proc. of the 3rd IEEE Israeli Symp. on Theory of Computing and Systems*, pages 121–129, January 1995.

[ST94] N. Shavit and D. Touitou. Software transactional memory. Technical Report 290/94, Tel-Aviv University, 1994.

[TSP92] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. In *Proceedings of 1992 Principles of Database Systems*, pages 212–222, 1992.

[TUX90] TUXEDO. system release 4.0 - product review. Unpublished, 1990.