An Efficient Universal Construction for Large Objects

Panagiota Fatourou
Institute of Computer Science, Foundation for Research and Technology-Hellas,
Department of Computer Science, University of Crete, Heraklion, Greece
faturu@csd.uoc.gr

Nikolaos D. Kallimanis
Institute of Computer Science, Foundation for Research and Technology-Hellas, Heraklion, Greece
nkallima@ics.forth.gr

Eleni Kanellou
Institute of Computer Science, Foundation for Research and Technology-Hellas, Heraklion, Greece
kanelou@ics.forth.gr

Abstract

This paper presents L-UC, a universal construction that efficiently implements dynamic objects of large state in a wait-free manner. The step complexity of L-UC is $O(n + kw)$, where n is the number of processes, k is the interval contention (i.e., the maximum number of active processes during the execution interval of an operation), and w is the worst-case time complexity to perform an operation on the sequential implementation of the simulated object. L-UC efficiently implements objects whose size can change dynamically. It improves upon previous universal constructions either by efficiently handling objects whose state is large and can change dynamically, or by achieving better step complexity.

2012 ACM Subject Classification Computing methodologies → Concurrent computing methodologies; Theory of computation → Concurrent algorithms; Computing methodologies → Concurrent algorithms; Computing methodologies → Shared memory algorithms; Theory of computation → Shared memory algorithms

Keywords and phrases universal construction, concurrent object, shared memory, simulation, wait-freedom, large object

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.18

Funding This work was supported by the General Secretariat of Research and Technology in Greece through project Sentitour at Scale (T1EDK-02857).

1 Introduction

1.1 Motivation and Contribution

Multi-core processors are nowadays found in all computing devices. Concurrent data structures are frequently used as the means through which processes communicate in multi-core contexts, thus it is important to have efficient and fault-tolerant implementations of them. A *universal construction* [11, 12] provides an automatic mechanism to get a concurrent implementation of any data structure (or object) from its sequential implementation.

In this paper, we present L-UC, an efficient, wait-free universal construction that deals with dynamic objects whose state is large. *Wait-freedom* [11] ensures that every process finishes the execution of each operation it initiates within a finite number of steps. The step complexity of L-UC is $O(n + kw)$, where n is the number of processes in the system, k is the *interval contention*, i.e., the maximum number of processes that are active during the execution interval of an operation, and w is the worst-case time complexity to perform

an operation on the sequential data structure. The step complexity of an algorithm is the maximum number of *shared memory accesses* performed by a thread for applying any operation on the simulated object in any execution.

A large number of the previously-presented universal constructions [1,2,5,7,8,11,12] work by copying the entire state of the simulated object locally, making the required updates on the local copy, and then trying to make the local copy shared by changing one (or a few) shared pointers to point to it. Copying the state of the object locally is however very inefficient when coping with large objects. L-UC avoids copying the entire state of the simulated object locally; in contrast, it applies the required changes directly on the shared state of the object. For doing so, processes need to synchronize when applying the changes. Previous universal constructions that apply changes directly to the shared data structure (e.g., [5]) synchronize on the basis of each operation. However, this results in high synchronization cost. To reduce this cost, L-UC applies a wait-free analog of the combining technique [8,9]: each process simulates, in addition to its own operation, the operations of other active processes. So, in L-UC, processes have to pay the synchronization cost once for a batch of operations and not for each distinct operation.

Sim [8,10] is a wait-free universal construction that implements the combining technique. In Sim, each process p that wants to apply an operation, first announces it in an *Announce* array. Then, p reads all other announced operations, makes a local copy of the shared state, applies all the operations it is aware of on this copy, and tries to update a shared variable to point to this local copy. P-Sim, the practical version of Sim (presented also in [8]) is highly efficient for objects whose state is small. L-UC borrows some of the ideas presented in [8]. Specifically, as P-Sim, L-UC uses an *Announce* array in which processes announce their operations, and employs bit vectors to figure out which processes have active operations at each point in time. However, the bit vector mechanism of L-UC is more elaborated than that of P-Sim, because the active processes have to agree on the set of operations that must be applied on the shared data structure before they attempt to perform any changes. In contrast to Sim, L-UC avoids copying locally the object's state. This makes L-UC appropriate for simulating large objects.

L-UC also borrows some ideas from the universal construction presented in [5] that copes with large objects. As in the universal construction in [5], in L-UC, each process uses a directory to store copies of the shared variables (e.g., the shared nodes) it accesses while executing operations on the data structure. L-UC combines this idea with the idea of implementing a wait-free analog of the combining technique. This way, L-UC achieves step complexity that is $O(n + kw)$. In scenarios of low contention, this bound can be much smaller than the $O(nw)$ achieved by the universal construction in [5]. Moreover, the universal construction in [5] have processes synchronize on the basis of every single operation, whereas in L-UC, processes synchronize once to execute a whole batch of operations.

1.2 Related Work

In [11], Herlihy studied how shared objects can be simulated, in a wait-free manner, using read-write registers and consensus objects. In the proposed universal construction, the simulated object is represented by a list of records. Each record stores information about an operation *op* (its type, its arguments, and its response) that has been performed on the simulated object. It also stores the state of the simulated object after all operations inserted in the list up until *op* (including it) have been applied on the implemented object in the order that they have been inserted in the list. To agree on which record will be inserted in the list next, each record additionally stores an n-consensus object. To ensure wait-freedom,

the algorithm also employs an announce array of n elements, where the n threads running in the system announce their operations, and stores a (strictly increasing) sequence number in each record, which illustrates the order in which this record was inserted in the list. Threads help the record of a thread i to be inserted as the j-th record in the list when $i = j \bmod n$. The step complexity of the algorithm is $O(n^2)$. The space overhead of the algorithm is $O(n^3)$ and each register contains the entire state of the object and a sequence number growing infinitely large. Herlihy revisited wait-free simulation of objects in [12], where it presented a universal construction which uses LL/SC and CAS objects and achieves step complexity $O(n + s)$, where s is the total size of the simulated object. These algorithms [11, 12] are inappropriate for large objects, as they work by copying the entire state of the object locally.

Afek, Dauber and Touitou presented in [1] a universal construction that employs a tree structure to monitor which processes are *active*, i.e. which processes are performing an operation on the simulated object at a given time. This tree technique was combined with some of the techniques proposed in [11, 12] in order to get a universal construction for simulating large objects, which has step complexity $O(kw \log w)$.

Anderson and Moir presented in [3] a wait-free universal construction for simulating large objects. In their algorithms, a contiguous array is used to represent the state of the object. Specifically, the object state is stored in B data blocks of size S each. To restrict memory overhead, the algorithms operate under the following assumptions: each operation can modify at most T blocks and each thread can help at most $M \geq 2T$ other threads. The step complexity of the universal construction in [3] is $O((n/\min\{k, M/T\})\ (B + MS + nw))$.

In [7], Fatourou and Kallimanis presented the family of RedBlue adaptive universal constructions. The F-RedBlue algorithm achieves $O(min\{k, logn\})$ step complexity and uses $O(n^2 + s)$ LL/SC registers. However, F-RedBlue uses large registers and it is not able to simulate objects whose state is stored in more than one register. S-RedBlue uses small registers, but the application of an operation requires to copy the entire state of the simulated object and thus it is inefficient for large objects. LS-RedBlue and BLS-RedBlue improve the step complexity of the algorithms presented by Anderson and Moir in [3] for large objects.

In [6], Felber et al. present CX, a wait-free universal construction, suitable for simulating large objects. This universal construction keeps up to $2n$ instances of the object state. In order to perform an update on the shared object, a process first appends its request in a shared request queue and then attempts to obtain the lock of some of the object instances. We remark that each such object instance stores a pointer to a queue node. Subsequently, the process uses this pointer to produce a valid copy of the object by performing all operations that were contained in the shared queue starting from the pointed node. Notice that CX has space complexity $O(ns)$, where n is the number of processes and s is the total size of the simulated object.

1.3 Roadmap

The rest of this paper is organized as follows. Our model is discussed in Section 2. L-UC is presented in Section 3. Section 3.1 provides an overview of the way the algorithm works and its pseudocode. Section 3.2 presents a detailed description of L-UC. A discussion of its complexity is provided in Section 3.3 and a sketch of proof for its correctness in Section 3.4.

2 Model

We consider an *asynchronous* system of n *processes*, $p_1, \ldots, p_n$, each of which may fail by *crashing*. Threads communicate by accessing (shared) base objects. Each *base object* stores a value and supports some primitives in order to access its state. An *LL/SC* object supports the

atomic primitives LL and SC. LL(O) returns the value that is stored into O. The execution of SC(O, v) by a thread p_i, $1 \leq i \leq n$, must follow the execution of LL(O) by p, and changes the contents of O to v if O has not changed since the execution of p's latest LL on O. If SC(O, v) changes the value of O to v, *true* is returned and we say that the SC is successful; otherwise, the value of O does not change, *false* is returned and we say that the SC is not successful or it is failed.L-UC is presented using LL/SC objects (as is the case for Sim [8, 10]). However, in a practical version of it, L-UC will be implemented using CAS objects (as is the case for P-Sim [8, 10]). A CAS object O supports in addition to Read(O), the primitive CAS(O, u, v) which stores v to O if the current value of O is equal to u and returns *true*; otherwise the contents of O remain unchanged and *false* is returned.

A *universal* construction can be used to implement any shared object. A universal construction supports the ApplyOp(req, i) operation, which applies the operation (or *request*) req to the simulated object and returns the return value of req to the calling thread p_i. In this paper, the concepts of an operation and a request have the same meaning and are used interchangeably. A *universal construction* provides a routine, for each process, to implement ApplyOp.

An object O is *linearizable*, if in every execution α, it is possible to assign to each completed operation op (and to some of the uncompleted operations), a point $*_{op}$, called the *linearization point* of op, such that: $*_{op}$ follows the invocation and precedes the response of op, and the response returned by op is the same as the response op would return if all operations in α were executed sequentially in the order imposed by the linearization points.

A *configuration* is a vector that contains the values of the base objects and the states of the processes, and describes the system at some point in time. At the *initial configuration*, processes are in their initial state and the base objects contain initial values. A *step* is taken by some process whenever the process executes a primitive on a shared register; the step may also include some local computation that is performed before the execution of the primitive. An *execution* is a sequence of steps. The *interval contention* of an instance of some operation in an execution is the number of processes that are active during the execution of this instance. The *step complexity* of an operation is the maximum number of steps that any thread performs during the execution of any instance of the operation in any execution. *Wait-freedom* guarantees that every process finishes each operation it executes in a finite number of steps.

3 The L-UC Algorithm

This section presents L-UC, our wait-free universal construction for large objects.

3.1 Overview

The pseudocode for L-UC is provided in Listings 1 and 2. The state of the simulated data structure in L-UC is shared and it can be updated directly by any process. Each process p that wants to apply a request, first announces it in an *Announce* array. In addition to the *Announce* array, L-UC uses a bit vector *Toggles* of n bits, one for each process. A process p_i toggles its bit, *Toggles*[i], after announcing a new request. The use of *Toggles* implements a fast mechanism for informing other processes of those processes that have pending requests.

Each execution of L-UC can be partitioned into phases. In each phase $i \geq 1$, the set of requests that will be executed in the next phase is agreed upon by the processes that are active. Moreover, those requests that have been agreed upon in the previous phase are indeed executed.

Listing 1 Data structures used in L-UC and pseudocode for LSimApplyOp.

```
1  struct NewVar {     // node of list of newly allocated variables
2      ItemSV *var;    // points to the ItemSV struct of the variable
3      NewVar *next;   // points to the next element of the list
4  };

6  struct NewList {
7      ItemSV *first;
8  };

10 struct State {
11     boolean applied[1..n];
12     boolean papplied[1..n];
13     int seq;
14     NewList *var_list;
15     RetVal RVals[1..n]; // return values
16 };

18 struct DirectoryNode {
19     Name name;       // variable name
20     ItemSV *sv;      // data item for the variable
21     Value val;       // value of the data item
22 };

24 struct ItemSV {  // data item for a variable
25     Value val[0..1];// old and new values of data item
26     int toggle;      // toggle shows the current value of data item
27     int seq;
28 };

30 // Toggles is implemented as an integer of n bits; if n is big, more than one
       such integers can be used
31 shared Integer Toggles = < 0,...,0 >;
32 shared State S = < F,...,F >,< F,...,F >,0,< ⊥ >,< ⊥,...,⊥ >>;
33 shared OpType Announce[1..n] = {⊥, ..., ⊥};

35 // Private local variable for process p_i
36 Integer toggle_i = 2^i;

38 RetVal ApplyOp(request req){  // Pseudocode for process p_i
39     Announce[i] = req;         // Announce request req
40     toggle_i = −toggle_i;
41     Add(Toggles, toggle_i);        // toggle p_i's bit by adding 2^i to Toggles
42     Attempt();       // call Attempt twice to ensure that req will be performed
43     Attempt();
44     return S.rvals[i];         // p_i finds its return value into S.rvals[i]
45 }
```

A process p_i that wants to execute a new request, it first announces it in *Announce*, and then it toggles its bit in *Toggles*. Afterwards, it calls a function, called `Attempt`, twice: After the execution of the first instance of `Attempt` by p_i, it is ensured that the set of requests agreed upon in one of the phases that overlap the execution of the `Attempt`, contains p_i's request. After the execution of the second instance of `Attempt` by p_i, it is ensured that p_i's request has been applied.

L-UC uses an `LL/SC` object S which stores appropriate fields to ensure the required synchronization between the processes in each phase. The first phase (phase 1) starts at the initial configuration and ends when the first successful `SC` is applied on S. Phase $i > 1$ starts when phase $i - 1$ finishes and ends when the i-th successful `SC` is applied on S.

To decide which set of requests will be executed in each phase, S contains two bit vectors, called *applied* and *papplied*, of n bits each (one for each process). The current request initiated by a process p_i has not yet been applied, if $S.applied[i] \neq S.papplied[i]$. When this condition holds, we call the current request of process p_i *pending*.

Listing 2 Pseudocode for L-UC.

```
1  void Attempt(Request req) {                    // pseudocode for process p_i
2    ProcessIndex q, j;
3    State ls, tmp;
4    Set lact;
5    Directory D;
6    NewVar *pvar = new NewVar(), *ltop;
7    ItemSV sv, *psv = new ItemSV();

9    psv→⟨val, toggle, seq⟩ = <<⊥,⊥>,0,0>;
10   pvar→⟨var, next⟩ = <psv, null>;
11   for j=1 to 2 do {
12      D = ∅;                                      // initialize direcory D
13      ls = LL(S);                                 // create a local copy of S
14      lact = Toggles;                             // read active set
15      ltop = ls.var_list→first; // read pointer to the list of newly-allocated
                                                    variables
16      tmp.seq = ls.seq + 1;
17      tmp.papplied[1..n] = ls.applied[1..n];
18      tmp.applied[1..n] = lact[1..n];             // p_i will later attempt to update S
                                                    with tmp, so it sets the fields of tmp
                                                    appropriately
19      tmp.rvals[1...n] = ls.rvals[1..n];
20      for q=1 to n do {
21         if (ls.applied[q] ≠ ls.papplied[q]) { // q's request is pending
22            foreach access of a variable x while applying request Announce[q]{
23               if (x is a newly allocated variable) {
24                  if(CAS(ltop→next, null, pvar)){
25                     psv = new ItemSV();
26                     psv→⟨val, toggle, seq⟩ = <<⊥,⊥>,0,0>;
27                     pvar = new NewVar();
28                     pvar→⟨var, next⟩ = <psv, null>;
29                  }
                    // use node pointed by ltop→next as the new variable's metadata
30                  ltop = ltop→next;
31                  add <x, ltop→var, ltop→var.val[0]> to D;
32               } else {                          // x is not a newly allocated variable
33                  let svp be a pointer to the ItemSV struct for x;
34                  if (this access is a read instruction) {
                       // perform the request on the local copy of x (if any)
35                     if (x exists in D) read x from D;
36                     else {
37                        sv = LL(*svp);
38                        if (tmp.seq==sv.seq) add <x,svp,sv.val[1−sv.toggle]> to D;
39                        else if(tmp.seq>sv.seq) add <x,svp,sv.val[sv.toggle]> to D;
40                        else goto Line 48; // values read from S by p_i obsolete, so
                                                start from scratch
41                     }
42                  } else if (the access is a write instruction) update x in D;
43               }
44            }
45            store into tmp.rvals[q] the return value;
46         }
47      }
48      if (!VL(S)) continue; // value read in S by p_i is obsolete, so start from
                              scratch
49      foreach record <x, svp, v> in D {
50         if(svp→seq > tmp.seq) break;  // all requests have been applied, so leave
                                          the loop
51         else if(svp→seq == tmp.seq) continue; // the variable has been modified,
                                                  so continue
52         else if(svp→toggle == 0) SC(*svp, <<svp→val[0],v>, 1, tmp.seq>);
53         else SC(*svp, <<v, svp→val[1]>, 0, tmp.seq>); // make update visible
54      }
55      tmp.var_list = new List(); tmp.var_list→first = null;
56      SC(S, tmp);                              // try to modify S
57   }
58 }
```

In each instance of `Attempt`, p_i copies the value of S in a local variable ls (line 13), records necessary changes that it makes to its fields in another local variable tmp (lines 16-19, 45, 55), and uses `SC` in an effort to update S to the value contained in tmp (line 56). Specifically,

p_i reads S on line 13 (by performing an LL) and $Toggles$ on line 14. It then copies $S.applied$ into $tmp.papplied$ (line 17) and $Toggles$ into $tmp.applied$ (line 18). Recall that the $applied$ and $papplied$ fields of S encode the requests that are to be performed in each phase. So, if the SC that p_i performs on line 56 succeeds, all processes that will read the value this SC will write to S, will attempt to perform the requests encoded by p_i in those fields.

Next, for each j, $1 \leq j \neq n$, p_i checks whether $ls.applied[j] \neq ls.papplied[j]$ (lines 20-21), and if this is so, it applies the request recorded in $Announce[j]$. To execute the pending requests recorded in S, a process p_i uses a caching mechanism as in [4, 5]: When a process first accesses a shared variable (e.g., a variable of the simulated shared data structure), it maintains a copy of it in a directory, D (which is local to p_i). For each pending request recorded in S, the required updates are first performed by p_i in the local copies of the data items that are residing in the directory (lines 22-45). Read requests executed by p_i are also served using D. Only after it has finished the simulation of all pending requests, p_i applies the changes listed in the elements of its directory to the shared data structure (lines 49-53).

For each data item x of the simulated object's state, L-UC maintains a record (struct) of type $ItemSV$. This struct stores the old and the current value of the data item in an array val of two elements, a toggle bit that identifies the position in the val array from where the current value for x should be read, and a sequence number that is used for synchronization.

Note that S contains also a field seq that is incremented every time a successful SC on S is performed. This field identifies the current phase of the execution. Before performing an update on the shared data structure (lines 49-53), p_i validates the values of the seq field read in S ($tmp.seq$) and that stored in $ItemSV$ for x ($svp \rightarrow seq$). Only if $svp \rightarrow seq < tmp.seq$ (line 53), the update is performed since otherwise it is already obsolete, i.e., $S.seq$ is already greater than $tmp.seq$ and therefore the SC of line 56 by p_i will fail.

Both the old and the current values of x must be stored in $ItemSV$ in order to avoid the following bad scenario. Consider two processes p_i and p_j that simulate the same request req. Assume that p_i is ready to execute line 37 for some variable x, whereas p_j has finished the simulation of req (lines 49-53) and has started updating the shared data structure. Then, it might happen that p_i reads the updated version for x although it should have read the old version. For this reason, p_j stores the old value (in addition to the new value) in one of the entries of the val array and appropriately updates the toggle bit to indicate which of the two values is the new one. If p_i discovers that it is too slow (line 38), it reads the old value for x stored in the $1 - toggle$ entry of its val array. Notice that, to ensure wait-freedom, p_i should continue executing req (to cope with the case that p_j fails before performing all the required updates to the shared data structure).

When a new data item x is allocated while executing a set of requests, additional synchronization between the processes that execute this set of requests is required to avoid situations where several processes allocate, each, a different record for x. We use a technique similar to that presented in [5] to ensure that all these processes use the same allocated ItemSV structure for x. Specifically, L-UC stores into S a pointer (called var_list) to a list of newly created data items shared by all processes that read this instance of S. Each time a process p_i needs to allocate the j-th, $j \geq 1$, such data item, it tries to add a structure of type $NewVar$ as the j-th element of the list (line 24). If it does not succeed, some other process has already done so, so p uses this structure (by moving pointer $ltop$ to this element on line 15, and by inserting $ltop \rightarrow var$ in its dictionary on line 31).

We remark that the fields of $ItemSV$ must be updated in an atomic way using SC. This requires that registers in the system store two words which is impractical. However, we can utilize single-word registers by using indirection. Indirection can also be used to implement S using single-word registers.

3.2 Detailed Description of `Attempt`

In the following, we detail the implementation of function `Attempt`, presented in Algorithm 2. When `Attempt` is executed by some process p_i, p_i executes two iterations (line 11) of checking whether there are pending requests and of attempting to apply them, as follows. It initializes its local directory D (line 12), creates in ls a local copy of the state of the simulated object (line 13), and reads in $lact$ the value of $Toggles$ (line 14), thus obtaining a view of which processes have pending requests at the current point in time (i.e., calculating the set of pending requests). Furthermore, it locally stores into $ltop$ a pointer to the current variable list of the simulated object (line 15). Recall that the state of the object is copied into local variable ls using an `LL` primitive. In case this instance of `Attempt` is successful in applying the pending requests, it will update the shared state of the system using an `SC` primitive. For this purpose, the local variable tmp is prepared in lines 16 to 18, to serve as the value that will be stored into the shared state in case of success.

After having read the state of the simulated object, as well as the state of the requests of the other processes, p_i can detect which requests are pending. For this purpose, it iterates through the (locally stored) state of each process (line 20) and checks whether the values of *papplied* and *applied* differ for this process (line 21). If so, the request of this process was still pending when `Attempt` read the value of $Toggles$ and therefore, `Attempt` intents to apply it. Notice that the iteration through the *papplied* and *applied* values consist of local steps. Notice also that at most k out of n processes have active requests, meaning what the request application contributes to step complexity depends on k rather than n.

We remark that the request of a process is expressed as a piece of sequential code. Therefore, in order to apply the request of some process, an instance of `Attempt` has to run through the sequential code of this request and carry out the variable accesses that this request entails, i.e. `Attempt` has to apply the modifications that this request incurs on the simulated object's variables (line 22). We distinguish three cases, namely the case where an access creates a new variable, the case where an access reads a variable, and the case where an access modifies an already existing variable.

In the first case (line 24), the new variable, which was created and stored in local variable $pvar$, must be added to the shared list of variables of the simulated object. Recall that a pointer to the top of the variable list has been read by p_i and stored in local variable $ltop$. Recall also that all processes are trying to perform the announced requests in the same order. As with each instance of `Attempt`, so also the p_i instance of `Attempt` tries to add $pvar$ to the top of the list using a `CAS` primitive (line 24). In case this is successful, the metadata of this variable is initialized. In case the `CAS` is unsuccessful, then some other process has updated the object's variable list since this instance of `Attempt` read it into $ltop$. Given that all processes follow the same order when trying to insert newly-allocated variables, the failure means that the variable has already been inserted in the shared list of variables of the simulated object. In either case, i.e. either successful or unsuccessful insertion by p_i, $ltop$ is updated to point to the data item of the newly allocated variable. Furthermore, the newly added variable is included into the local variable dictionary (line 31).

In the second case (line 34), the access to be performed is a read to a variable of the simulated object. If `Attempt` already has a local copy of this variable in its dictionary, it reads the value from there. If no local copy is present in the dictionary (line 37), then `Attempt` reads the variable using an `LL` primitive (line 37). At the same time, it checks the sequence number of the value that it has read, and in case this sequence number is less or equal to the local sequence number stored in tmp, then `Attempt` considers that it is reading a valid value. This value is then added to the local dictionary. However, in case the variable's

sequence number is larger than the local sequence number, this hints that this instance of `Attempt` has been rendered obsolete by some other process that has already applied all requests that this instance of `Attempt` is applying. In order to find out if this is the case, `Attempt` verifies whether the state of S has changed since it last read it (line 48) and if so, it gives up the current iteration of the for loop of line 11.

Finally, in the third case (line 42), where the access is a write to an already existing variable. In case that the accessed variable already exists in the local dictionary, the update on the local dictionary (line 42), updates the variable's value stored in the local dictionary. Otherwise, the update (line 42) creates a new entry and stores the value of the variable. Once the sequential code for the current request has all been run through and all variable accesses for the request have been performed, the request returns a return value, which is stored by `Attempt` for the process to access (line 45).

Recall that any update to a variable of the simulated object is performed locally by `Attempt`. Therefore, once all active requests have been applied, `Attempt` has to write back the local updates to the shared variables of the simulated object (lines 49 - 53). Notice that once again, the sequence numbers of the local and shared copies are instrumental in detecting whether a variable has already been updated or not (lines 51 - 53). More specifically, the condition of line 51 checks if another process has already updated or not the value of the shared variable while trying to apply the same set of operations calculated in lines 17 - 19. In case that a process is very slow and the whole set of operations calculated in lines 17 - 19 is applied, the condition of line 52 fails, and the process breaks the execution (line 50) of the *for-loop* of lines 49 - 53. Finally, once the updates have been performed, `Attempt` tries to update S, before performing any remaining iteration of the for loop of line 11.

3.3 Step Complexity

By inspection of the pseudocode of ApplyOp, it becomes apparent that its step complexity is determined by the step complexity of `Attempt`. In a practical version of L-UC where S is implemented using indirection, lines 13 and 14 contribute $O(n)$ to performance, since the size of the data records that are read is $O(n)$. The body of the if statement of line 21 (i.e., lines 22-42) is executed $O(k)$ times, each time contributing a factor of $O(w)$ (because of the foreach statement of line 22). Note that searching an element in the dictionary, adding an element to it or removing an element from it does not cause any shared memory accesses, i.e., it causes only local computation. So, the cost of executing lines 23-45 is $O(1)$. Note also that at most $O(kw)$ elements are contained in each dictionary. Therefore, the *foreach* of line 49 contributes $O(kw)$ to the total cost. The rest of the code lines access only local variables and thus they do not contribute to the step complexity of the algorithm. We conclude that the step complexity of ApplyOp is $O(n + kw)$.

3.4 Sketch of Correctness Proof

This section provides a sketch of the correctness proof of L-UC.

We start with some useful notation. Let α be any execution of L-UC and assume that some thread p_i, $i \in \{1, ..., n\}$, executes $m_i > 0$ requests in α. Let req_j^i be the argument of the j-th call of L-UC by p_i and let π_j^i be the j-th instance of `Attempt` executed by p_i (Figure 1). Let C_0 be the initial configuration. Define as Q_j^i the configuration after the execution of the `Add` instruction of line 41; let $Q_0^i = C_0$. We use $Toggles[i]$, $i \in \{1, \ldots, n\}$, to denote the i-th bit of $Toggles$, and let $toggle_j^i$ be the value of p_i's local variable $toggle_i$ at the end of req_j^i.

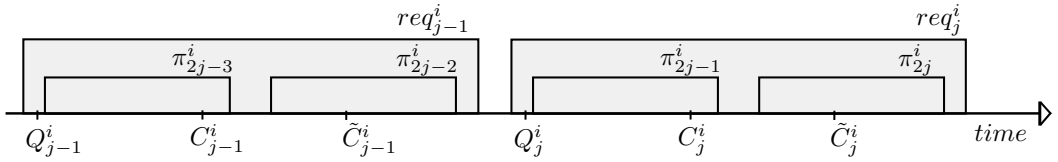

Figure 1 An example of an execution of L-UC.

In the following lemma, we argue that during the execution of each of the two iterations of the for loop of line 11 of any instance of `Attempt`, at least one successful `SC` instruction is performed.

▶ **Lemma 1.** *Consider any j, $0 < j \leq m_i$. There are at least two successful `SC` instructions in the execution interval of π^i_j.*

We continue with two technical lemmas. The first argues that the value of p_i's bit in the *Toggles* array is equal to $j \mod 2$ after the execution of the j-th `Add` instruction of line 41 by p_i. It also shows that no process other than p_i can change this bit.

▶ **Lemma 2.** *For each j, $0 \leq j \leq m_i$, it holds that (1) $Toggles[i] = j \mod 2$ at Q^i_j, and (2) $Toggles[i]$ has the same value between Q^i_{j-1} and Q^i_j.*

The next lemma studies the value of $S.applied[i]$ after the execution of the j-th instance of `Attempt` by p_i.

▶ **Lemma 3.** *Consider any execution π^i_j, $j > 0$, of function `Attempt` by some thread p_i. $S.applied[i]$ is equal to $v = \lceil j/2 \rceil \mod 2$ just after the end of π^i_j.*

For each $l > 0$, let C_l be the configuration resulting after the execution of the l-th `Add` instruction in α. At C_0, $S.applied[i]$ is equal to *false*. Lemma 3 implies that just after π^i_1, $S.applied[i]$ is equal to *true*. Let C^i_1 be the first configuration between C_0 and the end of π^i_1 at which $S.applied[i]$ is equal to *true*. Consider any request req^i_j, $j > 1$. Lemma 3 implies that just after π^i_{2j-2}, $S.applied[i]$ is equal to $\lceil (j-2)/2 \rceil \mod 2 = (j-1) \mod 2$, while just after π^i_{2j-1}, $S.applied[i]$ is equal to $\lceil (2j-1)/2 \rceil \mod 2 = j \mod 2 \neq (j-1) \mod 2$. Let C^i_j be the first configuration between the end of π^i_{2j-2} and the end of π^i_{2j-1} such that $S.applied[i]$ is equal to $j \mod 2$. Figure 1 illustrates the above notation.

Since the value of $S.applied[i]$ can change only by the execution of `SC` instructions on S, it follows that just before C^i_{j-1} a successful `SC` on S is executed. Let SC^i_j be this `SC` instruction and let LL^i_j be its matching `LL` instruction. Let T^i_j be the read of *Toggles* that is executed between LL^i_j and SC^i_j by the same thread.

Lemma 4 states that T^i_j is performed at the proper timing and returns the anticipated value.

▶ **Lemma 4.** *Consider any j, $0 < j \leq m_i$, it holds that T^i_j is executed after Q^i_j and reads $j \mod 2$ in $Toggles[i]$.*

Proof. Assume, by the way of contradiction, that T^i_j is executed before Q^i_j. Let π_x be the `Attempt` that executes T^i_j.

Assume first that $j = 1$. Then, by its definition, SC^i_1 (which is executed by π_x after T^i_1) writes to $S \rightarrow applied[i]$ a value equal to $\lceil j/2 \rceil \mod 2$; the code (lines 14, 18) implies that, in this case, T^i_1 reads 1 in $Toggles[i]$. Lemma 2 implies that $Toggles[i] = 0$ between C_0 and Q^i_1. Thus, T^i_1 could not read 1 in $Toggles[i]$, which is a contradiction.

Assume now that $j > 1$. By our assumption that T^i_j is executed before Q^i_j, it follows that LL^i_j, which is executed before T^i_j, precedes Q^i_j. In case that T^i_j follows Q^i_{j-1}, Lemma 2 implies that T^i_j reads $(j-1) \mod 2 \neq j \mod 2$ in $Toggles[i]$. By the pseudocode (lines 14, 18 and 56), it follows that π_x writes the value $(j-1) \mod 2$ into $S.applied[i]$. By its definition, SC^i_j stores $j \mod 2$ into $S.applied[i]$, which is a contradiction. Thus, T^i_j is executed before Q^i_{j-1}. By its definition, π^i_{2j-3} starts its execution after Q^i_{j-1} and finishes its execution before C^i_j. Lemma 1 implies that at least two successful `SC` instructions are executed in the execution interval of π^i_{2j-3}. Recall that LL^i_j precedes T^i_j and therefore also the beginning of π^i_{2j-3}, while by definition SC^i_j follows the end of π^i_{2j-3}. It follows that SC^i_j is not a successful `SC` instruction, which is a contraction. ◀

We next argue that, between certain configurations (namely C^i_{j-1} and C^i_j), the value of $S.applied[i]$ has the anticipated value and this value does not change in the execution interval defined by the two configurations.

▶ **Lemma 5.** *Consider any j, $0 < j \leq m_i$. At each configuration C between C^i_{j-1} and C^i_j, it holds that $S.applied[i] = (j-1) \mod 2$.*

Proof. Assume, by the way of contradiction, that there is at least one configuration between C^i_{j-1} and C^i_j such that $S \to applied[i]$ is equal to some value $v_x \neq (j-1) \mod 2$. Let C_x be the first of these configurations. Since only `SC` instructions of line 56 write on base object S, it follows that there is a successful `SC` instruction, let it be SC_x, executed just before C_x that stores v_x at $S.applied[i]$. Let π_x be the `Attempt` that executes SC_x and let T_x be the read instruction that π_x executes on line 14 of the pseudocode. By the definition of C^i_{j-1} and Q^i_{j-1}, it is implied that C^i_{j-1} follows Q^i_{j-1} and precedes Q^i_j. Lemma 2 implies that $Toggles[i] = (j-1) \mod 2 \neq v_x$ in any configuration between Q^i_{j-1} and Q^i_j. Since SC_x writes v_x into $S.applied[i]$, the pseudocode (lines 14 and 56) imply that T_x precedes Q^i_{j-1}. It follows that LL_x precedes Q_{j-1}, since LL_x precedes T_x. Therefore LL_x precedes C_{j-1}. This implies that there is a successful `SC` instruction, which is SC^i_{j-1}, between LL_x and SC_x. Thus, SC_x is a failed `SC` instruction, which is a contradiction. ◀

By Lemma 5 and the pseudocode (line 17), it follows that $S.papplied[i] = 1 - (j \mod 2)$ at C^i_j. Denote by $\tilde{C}^i_j$ the first configuration after C^i_j such that a successful `SC` instruction is executed.

The next lemma studies properties of $\tilde{C}^i_j$.

▶ **Lemma 6.** *$\tilde{C}^i_{j-1}$ precedes C^i_j and follows C^i_{j-1}.*

We next argue that the *applied* and *papplied* arrays of S indicate that p_i does not have a pending request between $\tilde{C}^i_{j-1}$ and C^i_j.

▶ **Lemma 7.** *$S.papplied[i] = S.applied[i]$ in any configuration between $\tilde{C}^i_{j-1}$ and C^i_j (C^i_j is not included).*

By Lemma 7, and by line 17, it follows that $S.papplied[i] = 1 - S.applied[i]$ at C^i_j. This and the definition of $\tilde{C}^i_j$ imply:

▶ **Lemma 8.** *$S.papplied[i] = 1 - S.applied[i]$ in any configuration between C^i_j and $\tilde{C}^i_j$ ($\tilde{C}^i_j$ is not included).*

We continue to define what it means for a process to apply a request on the simulated object. We say that a request *req* by some thread p_i *is applied* on the simulated object if (1) the `Read` instruction on *Toggles* (line 14), executed by some request *req*′ (that might be *req*

or any other request), includes p_i in the set of threads it returns, (2) procedure `Attempt`, executed by req' reads in $Announce[i]$, the request type written there by p_i for req and considers it as the new request type for p_i, (3) `Attempt` by req' calls `apply` for req (lines 22 - 45), and the execution of the `SC` at line 56 (let it be SC_r) on S succeeds. When these conditions are satisfied, we sometimes also say that req' applies req on the simulated object or that SC_r applies req on the simulated object.

▶ **Lemma 9.** *req_j^i is applied to the simulated object at configuration C_{3j-1}^i.*

Proof. Let p_h be the `Attempt` that executes the successful `SC` instruction (let it be SC_h this `SC` instruction) just before $\tilde{C}_j^i$. Let LL_h be the matching `LL` of SC_h. Since, SC_h is a successful `SC` instruction, it is implied that LL_h follows C_j^i. Observation 8 implies that LL_h reads for $S.applied[i]$ a value different from that stored in $S.papplied[i]$. Therefore, the if statement of line 21 returns *true*. Thus, a request for thread p_i is applied at $\tilde{C}_j^i$. Let req' be this request and assume, by the way of contradiction, that $req' \neq req_j^i$. Lemma 4 implies that π_h executes its read T_h on $Toggles$ after Q_j^i. By the pseudocode (lines 14, 22), π_h reads $Announce[i]$ after T_h, thus the reading of $Announce[i]$ by π_h is executed between Q_j^i and $\tilde{C}_j^i$. Since req_j^i writes its request to $Announce[i]$ before Q_j^i, the reading of $Announce[i]$ by π_h returns req_j^i. Thus, π_h applies req_j^i as the request of p_i in the simulated object. ◀

We are now ready to assign linearization points. For each $i \in \{1, ..., n\}$ and $j \geq 1$, we place the linearization point of req_j^i at $\tilde{C}_j^i$; ties are broken by the order imposed by identifiers of threads.

It is not difficult to argue that the linearization point of each request is placed in the execution interval of the request.

▶ **Lemma 10.** *Each request req_j^i is linearized within its execution interval.*

To prove consistency, denote by SC_l the l-th successful SC instruction on base object S. Let it_i be any iteration of the for loop of line 11 that is executed by a thread p_i. Let $SV_r(it_i)$ be the sequence of base objects read by the `LL` instructions of line 37 in it_i. Denote by $|SV_r(it_i)|$ the number of elements of $SV_r(it_i)$.

For each $1 \leq j \leq |SV_r(it_i)|$, denote by $SV_r^j(it_i)$ the prefix of $SV_r(it_i)$ containing the j first elements of $SV_r(it_i)$, i.e. $SV_r^j(it_i) = \langle sv_r^1(it_i), \ldots, sv_r^j(it_i)\rangle$, where $sv_r^j(it_i)$ is the j-th `LL` instruction performed by it_i on any base object. Let $SV_r^0(it_i) = \lambda$ be the empty sequence.

Let $V_r(it_i)$ be the sequence of insertions in directory D (lines 38-39) by it_i. Denote by $|V_r(it_i)|$ the number of elements of $V_r(it_i)$. Obviously, it holds that $|SV_r(it_i)| = |V_r(it_i)|$. For each $1 \leq j \leq |V_r(it_i)|$, denote by $v_r^i(it_i)$ the prefix of $V_r(it_i)$ containing the j first elements of $V_r(it_i)$, i.e. $V_r^j(it_i) = \langle v_r^1(it_i), \ldots, v_r^j(it_i)\rangle$, where $v_j(it_i)$ is the j-th value inserted to directory D. Let $V_r^0(it_i) = \lambda$ be the empty sequence.

Let $SV_w(it_i)$ be the sequence of shared base objects accessed by it_i while executing lines 51-52 (we sometimes abuse notation and say that a code line is executed by it_i to denote that the code line is executed by p_i during the execution of p_i). Denote by $|SV_w(it_i)|$ the number of elements of $SV_w(it_i)$. For each $1 \leq j \leq |SV_w(it_i)|$, denote by $SV_w^j(it_i)$ the prefix of $SV_w(it_i)$ that contains the j last elements of $SV_w(it_i)$, i.e. $SV_w^j(it_i) = \langle svw_1(it_i), \ldots, svw_j(it_i)\rangle$, where $svw_j(it_i)$ is the j-th request (lines 51-52) by it_i. Let $SV_w^0(it_i) = \lambda$ be the empty sequence.

Let $SV_a(it_i)$ be the sequence of shared base objects allocations during it_i iteration (lines 23-30). Denote by $|SV_a(it_i)|$ the number of elements of $SV_a(it_i)$. For each $1 \leq j \leq |SV_a(it_i)|$, denote by $SV_a^j(it_i)$ the prefix of $SV_a(it_i)$ that contains the j first elements of $SV_a(it_i)$, i.e. $SV_a^j(it_i) = \langle sva_1(it_i), \ldots, sva_j(it_i)\rangle$, where $sva_j(it_i)$ is the j-th base object allocation by it_i.

Let $SV_{arw}(it_i)$ be the sequence of allocations/reads/writes that it_i performs on base objects in lines 23-53 of the pseudocode. Denote by $|SV_{arw}(it_i)|$ the number of elements of $SV_{arw}(it_i)$. Obviously, it holds that $|SV_{arw}(it_i)| = |SV_a(it_i)| + |SV_r(it_i)| + |SV_w(it_i)|$. For each $1 \leq j \leq |SV_{arw}(it_i)|$, denote by $SV^j_{arw}(it_i)$ the prefix of $SV_{arw}(it_i)$ that contains the j first elements of sequence $SV_{arw}(it_i)$ (i.e. $SV^j_{arw}(it_i) = \langle svarw_1(it_i), \ldots, svarw_j(it_i)\rangle$) where $svarw_j(it_i)$ is the j-th base object allocations/reads/writes of base objects performed by it_i.

The next lemma states that for any process p_i that has a pending request, the i-th element of the *Announce* array stores the pending request of p_i for an appropriate time interval.

▶ **Lemma 11.** *Let $l > 0$ be any integer such that $S.applied[i] \neq S.papplied[i]$ at configuration C_{l-1}. Let req^i_j be the value of $Announce[i]$ at C_{l-1}. In any configuration between C_{l-1} and C_l, it holds that $Announce[i] = req^i_j$.*

▶ **Lemma 12.** *Let r be any shared base object other than S. For any $l > 0$, the following claims are true:*

1. *At most one successful* `SC` *instruction is executed on r between C_{l-1} and C_l.*
2. *In case that a successful* `SC` *instruction SC_w is executed on r, it holds that $r.seq < l$ just before SC_w and $r.seq = l$ just after SC_w.*
3. *Let it_i be some iteration of the loop of line 11 executed by a thread p_i that executes at least one successful* `SC` *instruction SC_r on r. If LL_r is the* `LL` *instruction of line 13 executed by it_i, then LL_r is executed after C_{l-1}.*
4. *Let it_i, $it_{i'}$ be two iterations of the for loop of line 11 executed by threads p_i and $p_{i'}$ respectively, such that that both it_i, $it_{i'}$ execute their* `LL` *instructions of line 13 somewhere between C_{l-1} and C_l, $l > 0$, and $|SV_{arw}(it_i)| \geq |SV_{arw}(it_{i'})|$. If both it_i, $it_{i'}$ execute line 49, just before C_l it holds that $SV_{arw}(it_i) = SV_{arw}(it_{i'})$.*

Proof. We prove the claims by induction on l. Fix any $l \geq 1$ and assume that the claims hold for l. We prove that the claims hold for $l + 1$.

We first prove Claim 1. Let SC' be the first of the successful `SC` instruction on r between C_{l-1} and C_l. We prove that $r.seq = l$ just after the execution of SC'. Assume by the way of contradiction that $r.seq = l' \neq j$. Let it_h be the iteration of line 13 executed by some thread p_h that executes SC'. Let LL' be the matching `LL` instruction of SC'. Since it_i executes successfully line 52 of the pseudocode, the pseudocode (lines 48 and 52) implies that the `VL` instruction of line 48 returns *true*. Since LL' is executed by it_i before this `VL` instruction, it follows that LL' precedes $SC_{j'}$. Thus, the `VL` instruction of line 48 is executed before $SC_{j'}$. Let $it_{i'}$ be the iteration of the loop of line 13 at which $SC_{j'}$ is executed and let $p_{i'}$ be the thread that executes $SC_{j'}$. Obviously, $LL_{j'}$ has been executed between $C_{l'-1}$ and $C_{l'}$. Since LL' is also executed between $C_{l'-1}$ and $C_{l'}$, the induction hypothesis (Claim 2.ii) implies that $SV_w(it_h) = SV_w(it_q)$. Thus, it_q has also executed an `SC` instruction on r. By lines 37, 49-52 and 56 of the pseudocode, it follows that there is a successful `SC` instruction on r between $SC_{l'-1}$ and $SC_{l'}$. Let SC_r be this instruction. By induction hypothesis (claim 1), it follows that $r.seq = j'$ just after the execution of SC_r. Since SC' is a successful `SC` instruction, LL' follows SC_r. By the pseudocode (lines 51-52), it follows that SC' is not executed, which is a contradiction. Therefore $r.seq = j$ just after the execution of SC_r. We now prove that there is no other successful `SC` instruction between SC' and C_l on r. Assume by the way of contradiction that at least one successful `SC` instruction takes place between SC' and C_l. Let SC'' be the first of these instructions. Since, SC'' is a successful `SC` instruction, it follows that its matching `LL` instruction LL'' follows SC'. By the pseudocode (lines 51-52), it follows that SC'' is not executed since $r.seq = S.seq$, which is a contradiction.

Claim 2 is proved using a similar argument as that above for Claim 1.

We now prove Claim 3. Assume by the way of contradiction that LL_p is executed between $SC_{j'-1}$ and $SC_{j'}$, $j' < j$. Let p_i be the thread that executes $SC_{j'}$ on some iteration it_i. By Claim 1 and by Claim 2, it follows that $r.seq \leq j'$ just before $SC_{j'}$. Thus SC_r is not executed, which is a contradiction. Thus, Claim 3 holds.

To prove Claim 4, it is enough to prove that $svarw_{l'}(it_i) = svarw_{l'}(it_{i'})$, for any $l' \leq |SV_{arw}(it_i)|$. We prove this claim by induction on the number $l' \leq |SV_{arw}|$ of elements of $SV_{arw}(it_i)$ (see appendix). ◀

Denote by α_i, the prefix of α which ends at SC_i and let C_i be the first configuration following SC_i. Let α_0 be the empty execution. Denote by l_i the linearization order of the requests in α_i.

We are now ready to prove that a_i is linearizable. This require to prove that the object state is consistent after the execution of each successful SC on S.

▶ **Lemma 13.** *For each $i \geq 0$, the following claims hold:*

1. *object's state is consistent at C_i, and*
2. *α_i is linearizable.*

Proof. We prove the claim by induction on i. The claim holds trivially; we remark that α_i is empty in this case. Fix any $i > 0$ and assume that the claim holds for $i - 1$. We prove that the claim holds for i.

By the induction hypothesis, it holds that: (1) object's state is consistent at C_{i-1}, and (2) α_{i-1} is consistent with linearization l_{i-1}. Let req be the request that executes SC_i. If req applies no request on the simulated object, the claim holds by induction hypothesis. Thus, assume that req applies $j > 0$ requests on the simulated object. Denote by $req_1, ..., req_j$ the sequence of these requests ordered with respect to the identifiers of the threads that initiate them.

Notice that req performs LL_i after C_{i-1} since otherwise SC_i would not be successful. By the induction hypothesis, object's is consistent at C_{i-1}. By Lemma 7, Observation 8, Lemma 9, and of the definition of $\tilde{C}^i_j$, it follows that each request req is applied exactly once. Thus, Lemma 12 imply that all threads that are trying to apply a set of requests between C_{i-1} and C_i do the following (1) apply the same set of requests with the same order, (2) all read the same consistent state of the object, (3) write the same set of base objects with the same values (although only one write succeeds), and (4) none of $req_1, \ldots, req_j$ have been applied in the past.

Given that $req_1, ..., req_j$ are executed by req sequentially, the one after the other in the order mentioned above, it is a straightforward induction to prove that (1) for each f, $0 \leq f \leq j$, request req_f returns a consistent response; moreover, $S \to st$ is consistent and once line 14 has been executed by req for all these requests. Therefore, $S \to st$ is consistent after the execution of req's successful SC. This concludes the proof of the claim. ◀

Lemma 13 implies that L-UC is linearizable. The discussion in Section 3.3 implies that L-UC is also wait-free and its step complexity is $O(n + kw)$. Thus:

▶ **Theorem 14.** *L-UC is a linearizable, wait-free implementation of a universal object. The number of shared memory accesses performed by L-UC is $O(n + kw)$.*

References

1. Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *Proceedings of the 27th ACM Symposium on Theory of Computing*, pages 538–547, 1995.
2. James H. Anderson and Mark Moir. Universal constructions for multi-object operations. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 184–193, 1995.
3. James H. Anderson and Mark Moir. Universal Constructions for Large Objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317–1332, December 1999.
4. Greg Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.
5. Phong Chuong, Faith Ellen, and Vijaya Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proceedings of the 22nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 335–344, 2010.
6. Andreia Correia, Pedro Ramalhete, and Pascal Felber. A Wait-Free Universal Construct for Large Objects, 2019. `arXiv:1911.01676`.
7. Panagiota Fatourou and Nikolaos D. Kallimanis. The RedBlue Adaptive Universal Constructions. In *Proceedings of the 23rd International Symposium on Distributed Computing*, pages 127–141, 2009.
8. Panagiota Fatourou and Nikolaos D. Kallimanis. A Highly-Efficient Wait-Free Universal Construction. In *Proceedings of the 23nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 325–334, 2011.
9. Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 257–266. ACM, 2012.
10. Panagiota Fatourou and Nikolaos D Kallimanis. Highly-Efficient Wait-Free Synchronization. *Theory of Computing Systems*, pages 1–46, 2013.
11. Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13:124–149, January 1991.
12. Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, November 1993.