# The RedBlue Adaptive Universal Constructions

Panagiota Fatourou
University of Ioannina
& FORTH-ICS, Crete Island

Nikolaos D. Kallimanis
Department of Computer Science
University of Ioannina

**Abstract**

We present the family of RedBlue algorithms, a collection of universal wait-free constructions for linearizable shared objects in an asynchronous shared-memory distributed system with $n$ processes. The algorithms are adaptive and improve upon previous algorithms in terms of their time and/or space complexity.

The first of the algorithms is of theoretical interest only; it achieves better time complexity than all previously presented algorithms but it is impractical since it uses large $LL/SC$ registers. This algorithm comprises the keystone for the design of the rest RedBlue algorithms which are of practical interest. The second algorithm significantly reduces the size of the required registers and it is therefore practical in many cases. The last two algorithms work efficiently for large objects improving previous universal constructions for large-objects presented by Anderson and Moir (PODC 1995).

Since our algorithms are universal, they are obviously more general than algorithms simulating specific types of shared objects (like *f-arrays* introduced by Jayanti in PODC 2002, *closed objects* introduced by Chandra, Jayanti and Tan in PODC 1998, or any other simulation of a restricted class of objects).

## 1 Introduction

In a shared memory system processes communicate by accessing shared objects, data structures that can be accessed concurrently by several processes. In this paper, we present the collection of RedBlue algorithms, wait-free universal constructions, i.e., algorithms that implement any shared object in an asynchronous system; *wait-free* algorithms [12] guarantee that a process completes the operation it executes in a finite number of its own steps despite the failures or the execution speed of other processes. The algorithms use $LL/SC$ registers; Herlihy has proved [12] that such algorithms necessarily use strong primitives (with infinite consensus number) like $LL/SC$.

In shared memory systems it is often the case that the total number of processes $n$ taking part in a computation is much larger than the actual number of processes that concurrently access the shared object. For this reason, a flurry of research [2, 1, 8, 9, 14] has been devoted to the design of *adaptive* algorithms whose time complexity depends on $k$, the maximum number of processes that concurrently access the shared object and not on $n$. All RedBlue algorithms are adaptive.

The algorithms use two perfect binary trees of $\log n+1$ levels each. The first tree, called the *red* tree, is employed for the estimation of any encountered contention. In each of these trees, a process is assigned a leaf node (and therefore also a path from this leaf to the root node, or vice versa). A process that wants to apply an operation on the simulated object, traverses first its path from the root downwards in the red tree looking for an unoccupied node in this path. Once it manages to occupy such a node, it starts traversing the second tree (which is called the *blue* tree) upwards from the isomorphic blue node to the occupied red node, trasfering information about its operation (as well as for other active operations) towards the tree's root. In this way, each operation traverses at most $O(\min\{k, \log n\})$ nodes in each of the two trees. Once information about the operation reaches the root, the operation is applied on the simulated object.

The first algorithm has time complexity $O(\min\{k, \log n\})$ which is better than the one of any previously presented algorithm but it uses big $LL/SC$ registers; thus it is only of theoretical interest. A lower bound of $\Omega(\log n)$ on the time complexity of universal constructions that use $LL/SC$ registers is presented in [16]. It holds even if an infinite number of unbounded-size registers are employed by the implementation. Our algorithm is therefore optimal in terms of time complexity.

The second algorithm (S-RedBlue) is a slightly modified version of RedBlue that uses smaller registers and it is therefore, in many cases, practical. S-RedBlue uses $O(n)$ $LL/SC$ registers, one for each of the trees' nodes and $n+1$ single-writer registers per process. Each register of the red tree has size $\log n + 1$. Each register of the blue tree stores $n$ bits, one for each process. One of the registers (the one corresponding to the blue root) is big. We implement this register by single-word LL/SC using the technique in [18]. In current systems where registers of 128 bits are available, S-RedBlue works with single-word LL/SC objects for up to 128 processes. In fact, even in cases that $n > 128$ but $n/128 = c$, where $c$ is any constant, our algorithm can be implemented by single-word LL/SC registers with the same time complexity (increased by a constant factor) using the implementation of multi-word LL/SC from single-word LL/SC [18].

Most of the universal algorithms presented in past, as well as RedBlue and S-RedBlue, copy the entire state of the object each time an update on it should be performed by some process. This is not practical for large objects that their states may require a large amount of storage to mantain it. Anderson and Moir [7] presented a lock-free and a wait-free universal construction that are practical for large objects. Their algorithm differs from Herlihy's large object approach [13] since it is array-based rather than pointer based; it also overcomes some of the drawbacks of Herlihy's algorithm. We combine some of the techniques introduced in [7] with the techniques employed by the RedBlue algorithms to design two simple wait-free constructions which have the nice properties of the constructions in [7] while achieving better time complexity. More specifically, our algorithms are adaptive. The time complexity of the first algorithm is much better than the wait-free construction presented in [7] but it does not assume an upper bound on the number of processes a process may help as the wait-free construction in [7] does. Our last algorithm (BLS-RedBlue) exhibits all the properties of the wait-free construction in [7] and still achieves better time complexity. In particular, its time complexity is similar to the one in [7] but with $k$ having substituted $n$. The space complexity of the algorithm is the same as the one of the wait-free algorithm in [7]. Our algorithms are much simpler than the constructions presented in [7], and they achieve significantly improved time complexity compared to these algorithms.

Afek, Dauber and Touitou [3] have presented algorithm GroupUpdate which also uses a tree technique to keep track of the list of active processes. They then combine this tree construction with Herlihy's universal algorithm [12, 13] to get a universal construction with time complexity $O(k \log k + W + kD)$, where $W$ is the size (in words) of the simulated object state and $D$ is the time required for performing a sequential operation on it. Our first algorithm is (kind of) a simplified version of GroupUpdate thus achieving better time complexity ($O(\min\{k, \log k\})$). However, it is inferior to GroupUpdate in that it uses big $LL/SC$ registers. Our second algorithm addresses this problem still achieving better time complexity than GroupUpdate.

Although the first of the RedBlue algorithms shares a lot of ideas with GroupUpdate, it exhibits also several differences: (1) it employs two complete binary trees each of which has one more level than the single tree employed by GroupUpdate; in each of these trees, each process is assigned its own leaf node which identifies a unique path (from the root to this leaf) in the tree for the process; (2) processes traverse the red tree first in order to occupy a node and this procedure is faster than a corresponding procedure in GroupUpdate. More specifically, GroupUpdate performs a $BFS$ traversal of its employed tree in order for a process to occupy a node of the tree, while each process in any of the RedBlue algorithms always traverses appropriate portions of its unique path. This results in reduced time complexity for some of the RedBlue algorithms. The rest of our algorithms additionally employ a completely different mechanism for transfering the response values to the active processes, and they present several other differences.

Afek, Dauber and Touitou [3] present a technique that employs indirection to reduce the size of the registers used by GroupUpdate (each tree register stores a process id and a pointer to a list of ids of currently active processes). A similar technique can be applied to the RedBlue algorithms in case $n$ is large enough to have $n$ bits stored in a constant number of $LL/SC$ registers. The resulting algorithms will have just a pointer stored in each of the blue nodes (thus using smaller registers than GroupUpdate which additionally stores and a process id in each of its $LL/SC$ register). However, employing this technique would cause an increase to the step complexity of our algorithms by an $O(k \log k)$ additive factor.

Afek, Dauber and Touitou present in [3] a second universal construction (IndividualUpdate) that has time complexity $O(k(W + D))$. IndividualUpdate stores into registers sequence numbers and therefore it requires unbounded size registers or registers that support the $VL$ operation in addition to $LL$ and $SC$. The first two RedBlue algorithms achieve better time complexity than IndividualUpdate. Some of our algorithms use single-word registers (however, they also employ $LL/VL/SC$ objects).

Afek, Dauber and Touitou [3] discuss a method similar to the one presented in [10] to avoid copying the entire object's state in IndividualUpdate. The resulting algorithm has time complexity

$O(kD \log D)$. The work of Anderson and Moir on universal constructions for long objects [7] follows this work. Our last two algorithms improve in terms of time complexity upon the constructions presented in [7]. They achieve this using single-word registers (and the last algorithm with the same space complexity as the wait-free construction in [7]).

Jayanti presented in [17] f-arrays, a generalized version of a snapshot object which allows the execution of any aggregation function $f$ on the $m$ elements of an array of $m$ memory cells that can be updated concurrently. As our first algorithm, f-arrays has time complexity $O(\min\{k, \log n\})$; the algorithm uses a tree structure similar to the one emplyed by GroupUpdate and our algorithm. Our algorithm is universal, thus achieving wider functionality than f-arrays. Constructions for other restricted classes of objects with polylogarithmic complexity are presented in [11].

Afek *et al.* [4, 5] and Anderson and Moir [6] have presented universal algorithms for multi-object operations that support access to multiple objects atomically. The main difficulty encountered under this setting is to ensure good parallelism in cases where different operations perform updates in different parts of the object's state. We are currently working on whether appropriately modified versions of some of the RedBlue algorithms can work efficiently for multi-object operations and we plan to include multi-object operation RedBlue-like universal constructions in the full version.

## 2 Model

We consider an asynchronous shared-memory system of $n$ processes which communicate by accessing shared objects. A *read-write register* stores a value from some set and supports two operations: `read` returns the value of the register leaving its content unchanged, and `write`$(v)$ writes the value $v$ into the register and returns `ack`. An *LL/SC register* $R$ stores a value from some set and supports the atomic operations $LL$ and $SC$; $LL(R)$ returns the current value of $R$; the execution of $\mathrm{SC}(R, v)$ by a process $p$ must follow the execution of $\mathrm{LL}(R)$ by $p$, and it is successful only if no process has performed a successful SC on $R$ since the execution of $p$'s latest LL on $R$; if $\mathrm{SC}(R, v)$ is successful the value of $R$ changes to $v$ and *true* is returned. Otherwise, the value of $R$ does not change and $false$ is returned. Some $LL/SC$ registers support the operation $VL$ in addition to $LL$ and $SC$; $VL$ returns *true* if no process has performed a successful SC on $R$ since the execution of $p$'s latest LL on $R$, and $false$ otherwise. A register is *multi-writer* if all processes can change its content; on the contrary, a *single-writer* register can be modified only by one process. A register is *unbounded* if the set of values that can be stored in it is unbounded; otherwise, the register is *bounded.*

A *configuration* consists of a vector of $n + r$ values, where $r$ is the number of registers in the system; the first $n$ attributes of this vector describe the state of the processes, and the last $r$ attributes are the values of the $r$ registers of the system. In the *initial configuration* each process is in an initial state and each register contains an initial value. A process takes a *step* each time it accesses one of the shared registers; a step also involves the execution of any local computation that is required before the process accesses some shared register again (this may cause the state of the process to change). An *execution* is a sequence of steps.

Registers are usually used to simulate more complex objects. A *simulation* of an object $O$ (which supports e.g., $l$ operations) by registers uses the registers to store the data of $O$ and provides $l$ algorithms for each process, to implement each of the $l$ operations supported by the simulated object. The *time complexity of an operation* is the maximum number of steps performed by any process to execute the operation in any execution of the simulation. The *time complexity* of the simulation is defined to be the maximum between the time complexities of its operations. A *universal object* simulates all other objects.

A process is *active* if it has initiated but not yet finished the execution of an operation *op*. When this is true, we also say that *op* is *active*. The portion of an execution that starts with the invocation of an operation *op* and ends with *op*'s response is called the *execution interval* of *op*. The *interval contention* of *op* is the total number of processes that take steps during the execution interval of *op*. The *point contention* of *op* is the maximum number of processes that are active at any configuration during the execution interval of *op*. The interval (point) contention of a simulation is the maximum interval (point) contention of any operation performed in any execution of the simulation. An execution is *serial* if for any two steps executed by the same operation, all steps between them are executed also by the same operation.

We assume that processes may experience *crash failures*, i.e., they may stop running at any point in time. A *wait-free* algorithm [12] guarantees that a process finishes the execution of an operation within a finite number of its own steps independently of the speed of the other processes or the faults they experience. (*Lock-freedom* is a weaker progress property that allows individual processes to starve but guarantees system-wide progress.) We concentrate on *linearizable* implementations [15]. *Linearizability* guarantees that in any execution $\alpha$ of the simulation, each of the executed operations in $\alpha$ appears to take effect at some point, called the *linearization point*, within its execution interval.

## 3 The F-RedBlue Algorithm

### 3.1 Algorithm description

F-RedBlue uses a perfect binary tree (called *blue tree*) of $\log n + 1$ levels, each node of which is an $LL/SC$ register. Each process $p$ owns one of the tree leaves and it is the only process capable to modify this leaf. For each process $p$, there is therefore a unique path $pt(p)$ (called *blue path* for $p$) from the leaf node assigned to $p$ up to the root. Each node stores an array of $n$ operation types (and their parameters), one for each process $p$ to identify the operation that $p$ is currently executing. The root node stores additionally the state of the simulated object (and for each process, the return value for the last operation (being) applied on the simulated object by the process).

Whenever $p$ wants to apply an operation $op$ on the object, it moves up its path until it reaches the root and ensures that the type $op\_tp(op)$ of $op$ is recorded in all nodes of the path by executing two $LL/SC$ on each node. If any of the $LL/SC$ that $p$ executes on a node succeeds, $op\_tp(op)$ is successfully recorded in it; otherwise, it can be proved that $op\_tp(op)$ is recorded for $p$ in the node by some other process before the execution of the second of the two $SC$ instructions executed by $p$. In this way, the type of $op$ is propagated towards the root where $op$ is applied to the object.

Process $p$ also records in each node the operations executed by other active processes in an effort to help them finish their executions. Successful $SC$ instructions executed at the root node may cause the application of several operations by different processes to the simulated object. In this way, the algorithm guarantees wait-freedom.

Once $p$ ensures that $op$ has been applied, it traverses its path from its leaf up to the root once more to eliminate any evidence of its last applied operation by overwriting the operation type of its last operation with the special value $\perp$. This allows $p$ to execute more operations on the simulated object; more specifically, a new operation $op'$ executed by $p$ is applied on the simulated object only if its operation type $op\_tp(op')$ reaches the root and finds the value $\perp$ stored for $p$ in it.

This relatively simple algorithm requires $O(\log n)$ steps to execute. In order to make it adaptive,

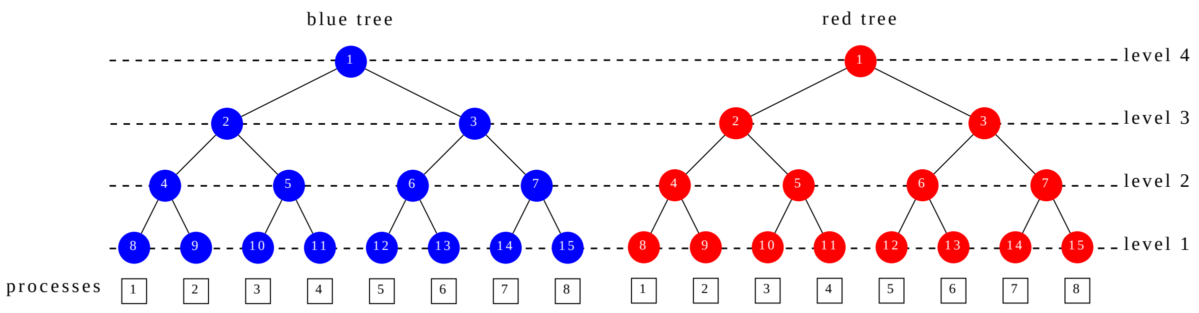

Figure 1: Blue and red tree of F-RedBlue algorithm for $n = 8$.

we use one more tree (the *red tree*), isomorphic to the blue tree. Thus, each process $p$ is assigned a leaf node of the red tree which identifies a unique path from the root to this leaf (*red path for* $p$). The red tree allows processes to obtain information about the encountered contention which is then used to shorten the paths that processes traverse in the blue tree (i.e., the process starts its traversal of its blue path possibly from some internal node of the tree which is at a level that depends on the encountered contention).

Each node of the red tree stores information about only one operation, namely the operation that is applied by the process that "occupies" the node. More specifically, each process $p$ first tries to occupy a node of the red tree and then starts traversing (part of) its blue path. In order to occupy a red node, $p$ traverses its red path downwards, from the root to its assigned leaf, until it finds a *clean node* (i.e., a currently unoccupied node with the value $(\perp, -1)$) and manages to occupy it by recording its operation type and its id in it. We will later prove that each red node is occupied by at most one process at any point in time. An occupied node identifies a process that is currently active, so as long as $p$ reaches occupied nodes, it encounters more contention. It is to prove that $p$ will eventually reach an unoccupied node and will achieve to record the appropriate information there. (This, in the worst case, will be its leaf node). Once $p$ occupies some red node with id $z_r$, it starts each of the two traversals of its blue path from the node of the blue tree that corresponds to $z_r$ up to the root. By employing the red tree, processes traverse shorter paths in the blue tree. This improves the time complexity of the algorithm to $O(\min\{k, \log n\})$, where $k$ is the interval contention of *op*.

We continue to provide a more technical description of the algorithm. Since the blue (red) tree is a perfect binary tree and there is only one such tree with $\log n + 1$ levels, we implement it using an array $bn$ ($rn$) of $2n-1$ elements. The nodes of the tree are numbered so that node $z$ is stored in $bn[z]$ ($rn[z]$, respectively). The root node is numbered with 1, and the left and right children of any node $z$ are nodes $2z$ and $2z+1$, respectively. The two trees for $n = 8$ are illustrated in Figure 1. Process $p$, $1 \leq p \leq n$, is assigned the leaf node numbered $n + p - 1$. We remark that traversing up the path from any node $z$ to the root can be implemented in a straightforward manner: the next node of $z$ in the path is node numbered $\lfloor z/2 \rfloor$. However, the downward traversal of the path requires some more calculations which are accomplished by the lines $5 - 10$ of the pseudocode.

When a process $p$ wants to execute an operation *op* of type *op_tp*(*op*) it first traverses its red path (lines $1 - 10$). For each node $z$ of this path, it checks if the node is unoccupied (line 3) and if this is so, it applies an *SC* instruction on it in an effort to occupy it (line 4). If the *SC* is successful, the traversal of the red path ends (line 4). Otherwise, the next node in the path is calculated (lines

**Algorithm 1** Pseudocode of F-RedBlue.

```
type PINDEX $\{1,\ldots,n\}$;                     struct bnode{
struct rnode{                                          state $st$;          // field used only at the root node
    operation_type $op\_tp$;                             ret_vals $vals[n]$;  // field used only at the root node
    PINDEX $pid$;                                        operation_type $ops\_tp[n]$;
}                                                  }
shared struct rnode $rn[1..2n-1]$ = $\{<\perp,-1>,...,<\perp,-1>\}$;
shared struct bnode $bn[1..2n-1]$ = $\{<\perp,<0,...,0>,<\perp,...,\perp>>,...,<\perp,<0,...,0>,<\perp,...,\perp>>\}$;

ret_val apply_op(operation_type $op\_tp$, PINDEX $p$){
    int $direction$ = n/2, $z$ = 1, $levels$ = lg(n) + 1, l;
    ret_vals $rv$;
1.  for($l$=$levels$;$l \geq 1$;$l$--){    // traversal of red path
2.      LL($rn[z]$);
3.      if($rn[z]$ == $<\perp,-1>$)
4.          if(SC($rn[z]$, <$op\_tp$, $id$>)) break;
5.      if($id \leq direction$){                        // find the next node in the path
6.          $direction$ = $direction$ - $2^{l-3}$;
7.          $z = 2*z$;                                  // move to the left child of $z$
        }
8.      else{
9.          $direction$ = $direction$ + $2^{l-3}$;
10.         $z = 2*z+1$;                                 // move to the right child of $z$
        }
    }
11. propagate($z$, $p$);               // first traversal of blue path: propagating the operation
12. $rv$ = $bn[1].vals[id]$;
13. LL($rn[z]$);
14. SC($rn[z]$, <$\perp$, $p$>);             // indicate that the operation occupying $rn[z]$ starts its deletion phase
15. propagate($z$, $p$);               // second traversal of blue path: propagating $\perp$
16. LL($rn[z]$);
17. SC($rn[z]$, $<\perp,-1>$ );          // re-initiate the occupied red node to $\perp$
18. return $rv$;                       // return the appropriate value
}

void propagate(int $z$, PINDEX $p$){
19. while($z$!=0){                     // traversal of the blue path
20.     for(int $i$=1 to 2)do {        // two efforts to store appropriate information into each node
21.         LL($bn[z]$);
22.         $bt$=calculate($z$, $p$);
23.         SC($z$, $bt$);
        }
24.     z =$\lfloor$z/2$\rfloor$;
    }
}

struct bnode calculate(int $z$) {
    struct bnode tmp=$<\perp,<0,\ldots,0>,<\perp,\ldots,\perp>>$, blue=bn[z], lc, rc;
    struct rnode red = rn[z];
25. if ($2*z+1<2n$) { $lc$ = $bn[2*z]$; $rc$ = $bn[2*z+1]$; }              // actions if $z$ is an internal node
26. if ($z == 1$) { $tmp.ret\_val[1..n]$ = $blue.ret\_val[1..n]$; $tmp.st = blue.st$; } // actions if $z$ is the root node
27. for $q=1$ to $n$ do{
28.     if ($red.pid$ == $q$) $tmp.ops\_tp[q] = red.op\_pt$;                // check if process $q$ occupies node red
29.     else if (is_predecessor($z$,$q$,$2*z$)) $tmp.ops\_tp[q]$=$lc.op\_tp[q]$;
30.     else if (is_predecessor($z$,$q$,$2*z+1$)) $tmp.ops\_tp[q]$=$rc.op\_tp[q]$;
31.     if ($tmp.ops\_tp[q]$ $\neq$ $\perp$ AND $tmp.ops\_tp[q]$ $\neq$ $blue.ops\_tp[q]$)
32.         apply $tmp.ops\_tp[q]$ to $tmp.st$ and store into $tmp.ret\_vals[q]$ the return value;
    }
33. return tmp;
}

boolean is_predecessor(int $z$, int $p$, int $pred$){
34.    int $levels$ = $\log n+1$, $total\_nodes$ = $2^{levels}-1$, $leaf\_node = \lfloor total\_nodes/2\rfloor + p$;
35.    int $pred\_height$=$levels - \lfloor lg(2*z)\rfloor$, $real\_pred$=$\lfloor leaf\_node/2^{pred\_heigth-1}\rfloor$;
36.    if ($pred < 2*n$ AND $real\_pred$ == $pred$) return $true$;
37.    else return $false$;
}
```

$5 - 10$) and one more iteration of the loop is performed.

Once a red node $z_r$ has been occupied, *op* performs two traversals of (a part of) its blue path starting from the corresponding to $z_r$ node of the blue tree up to the root (lines $11 - 15$). This is accomplished by the two calls to `propagate`. Each of these traversals propagates the operation type written into $z_r$ to the root node. Notice that $p$ records $\perp$, as its operation type, into $z_r$ (lines $13-14$) before it starts its second traversal (notice that this occurs by performing one more $LL/SC$, although a `write` instruction would be enough, since we assume that an $LL/SC$ register supports only the operations `read`, $LL$, and $SC$ and not `write`).

On each node $z$ of the traversed path, `propagate` performs twice the following: (1) an $LL$ instruction on $z$ (line 21); (2) calculates the appropriate information to write into $z$ by calling function `calculate` (line 22); (3) an $SC$ to store the result of `calculate` into $z$ (line 23). Finally, it moves up to the next node of the blue path (line 24). Process $p$ re-initiates its occupied red node by writing in it the value $(\perp, -1)$ (lines $16 - 17$) just before it returns.

Function `calculate` computes a (potentially new) operation type for each process $q$ (lines $27 - 32$) as described below. If $q$ occupies the isomorphic to $z$ red node (line 28) then $q$'s new operation type is the one which is recorded into the red node. Otherwise, the operation type for $q$ is found in the previous node of $z$ in $q$'s blue path. In case z is the root node and the calculated operation type for $q$ is not already written in z and it is different than $\perp$ (line 31), then the operation of $q$ is a new one and should be applied to the simulated object (line 32). This is simulated by calling function `apply`.

## 3.2 Correctness proof

In this section we prove the correctness of F-RedBlue and analyze its complexity.

We start by studying the properties of the execution portion of an operation *op* (i.e., an instance of `apply_op`) that traverses the red tree. Intuitively, we prove that *op* manages to occupy a red node (Lemma 3.2), and as long as *op* is executed, no other operation succeeds in updating this red node (Lemmas 3.5 and 3.6). Once *op* finishes its execution, it stores into its red node its initial value in order to allow its re-occupation by some other operation.

Call the SC instructions of line 4, SC of type 1, the SC instructions of line 14, SC of type 2, and the SC instructions of line 17, SC of type 3. Let $p$ be any process. By the pseudocode, only process $p$ executes SC instructions on register $rn[n-1+p]$, the red leaf associated to $p$. Thus, all these instructions succeed. Therefore, the condition of line 3 of the pseudocode is evaluated to `TRUE` when executed on $rn[n-1+p]$ and, by the pseudocode, the sequence of SC executed on $rn[n-1+p]$ alternates between SC of type 1, SC of type 2, and SC of type 3.

**Observation 3.1** *Let $p$ be any process. Then,*

1. *only process $p$ executes SC instructions on register $rn[n-1+p]$;*
2. *the condition of line 3, when executed on $rn[n-1+p]$, is evaluated to* `TRUE`*;*
3. *all SC on $rn[n-1+p]$ succeed;*
4. *the sequence of SC executed on $rn[n-1+p]$ alternates between SC of type 1, SC of type 2, and SC of type 3.*

Based on Observation 3.1, it is easy to prove that any operation *op*, executed by some process $p$, performs a successful $SC$ of type 1 at some node of the red tree, since, in the worst case, this will occur at $p$'s red leaf, the last node of $p$'s red path.

**Lemma 3.2** *Any instance op of apply_op executes a successful SC instruction of type 1 at some node of the red tree.*

**Proof:** Let $p$ be the process that executes *op*. Assume, by the way of contradiction, that *op* does not execute a successful SC of type 1 on any node of the red tree. Then, by the pseudocode (lines 1-10), *op* executes an $SC$ instruction of type 1 on any node of the red path of $p$. Since the last node in this path is $rn[n-1+p]$, it follows that the SC executed by $p$ on $rn[n-1+p]$ is not successful, which contradicts Observation 3.1 (claim 3). ■

Let $z$, $1 \leq z \leq 2n-1$, be the id of a node of the red tree. For any $j \geq 1$, denote by $SC_1^j(z)$ the $j$-th successful SC of type 1 executed on $z$ (i.e., on the node with id $z$), and let $op_j(z)$ be the operation that executes $SC_1^j(z)$. We often abuse notation and omit $z$, whenever it is clear from the context. Notice that, by definition, there are no successful SC instructions of type 1 between $SC_1^j$ and $SC_1^{j+1}$.

We say that a red node with id $z$ *is occupied* by a process $p$ at some configuration $C$, if it holds that $rn[z].pid = p$ at $C$. If $p$ is executing operation *op* at $C$, we also say that $z$ is occupied by *op* at $C$. We continue to prove that each red node, occupied by some operation *op*, should first be released by *op* before it can be occupied again by some other operation.

**Lemma 3.3** *For each $j \geq 1$, $op_j$ executes a successful SC instruction of type 3, which we denote by $SC_3^j$, on $rn[z]$ between $SC_1^j$ and $SC_1^{j+1}$.*

**Proof:** First, we prove that at least one successful SC of type 3 is executed on $rn[z]$ between $SC_1^j$ and $SC_1^{j+1}$. We let $SC_3^j$ be the first of these successful SC instructions. Then, we prove, by induction on $j$, that $SC_3^j$ is executed by $op_j$ (i.e., the same operation that executes $SC_1^j$).

1. Assume, by the way of contradiction, that no successful SC of type 3 is executed on $rn[z]$ between $SC_1^j$ and $SC_1^{j+1}$. Recall that $op_j$ is the operation that executes $SC_1^j$ and $op_{j+1}$ is the operation that executes $SC_1^{j+1}$.

   First, we prove that the read of $rn[z]$ (line 3) by $op_{j+1}$ follows $SC_1^j$. Assume, by the way of contradiction, that this read occurs before $SC_1^j$. The execution of the $LL$ of line 2 by $op_{j+1}$ precedes this read, so the execution of this $LL$ occurs before $SC_1^j$. Since the corresponding SC to this $LL$ is $SC_1^{j+1}$, and occurs after the successful $SC_1^j$ instruction, $SC_1^{j+1}$ cannot be successful, which is a contradiction. Therefore, the read at line 3 by $op_{j+1}$ follows the execution of $SC_1^j$.

   Since $op_{j+1}$ executes $SC_1^{j+1}$, by the pseudocode (lines $3-4$), it follows that $op_{j+1}$ has read the value $-1$ in variable $rn[z].pid$ (line 3). Let $SC_l$ be the last successful SC on $rn[z]$ preceding $SC_1^{j+1}$. Recall that we have assumed that no successful SC of type 3 is executed on $rn[z]$ between $SC_1^j$ and $SC_1^{j+1}$. Moreover, by definition of $SC_1^j$ and $SC_1^{j+1}$, no successful $SC$ of type 1 occurs between $SC_1^j$ and $SC_1^{j+1}$. Thus, $SC_l$ must be either $SC_1^j$ or some successful SC of type 2. In either case, it follows by the pseudocode (lines 4, 14), that $SC_l$ writes a value different than $-1$ into $rn[z].pid$, which is a contradiction. Thus, there is at least one successful SC of type 3 executed on $rn[z]$ between $SC_1^j$ and $SC_1^{j+1}$.

Let $SC_3^j$ be the first successful SC of type 3 executed on $rn[z]$ between $SC_1^j$ and $SC_1^{j+1}$.

2. We prove, by induction on $j$, that $SC_3^j$ is executed by $op_j$. Fix any $j \geq 1$ and assume that the claim holds for any $j', 1 \leq j' < j$.

   We prove that the claim holds for $j$. Assume, by the way of contradiction, that $SC_3^j$ is executed by some operation $op \neq op_j$. By the pseudocode and by Lemma 3.2, $op$ executes a successful SC instruction of type 1 on some node of the red tree before $SC_3^j$; let $SC_1$ be this SC instruction. By the pseudocode (lines 4, 18), $SC_1$ and $SC_3^j$ are executed on the same node, namely on node $z$.

   By the definitions of $SC_1^j$ and $SC_1^{j+1}$, no other successful SC of type 1 is executed on $z$ between $SC_1^j$ and $SC_1^{j+1}$. Moreover, $SC_1^j \neq SC_1$ since $SC_1^j$ is executed by $op_j \neq op$. Thus, $SC_1$ is executed before $SC_1^j$.

   If $j = 1$, this is a contradiction, since, by definition, $SC_1^j$ is the first successful $SC$ of type 1 on $rn[z]$. If $j > 1$, let $SC_1'$ be the first successful SC of type 1 on $rn[z]$ following $SC_1$. Notice that $SC_1'$ is either $SC_1^j$ or some earlier successful SC of type 1 on $z$. As proved above (item 1), there is at least one successful SC of type 3 executed on $rn[z]$ between $SC_1$ and $SC_1'$. Let $SC_3$ be the first such SC; obviously, $SC_3$ precedes $SC_3^j$. Then, by the induction hypothesis, $SC_3$ is executed by $op$. By the pseudocode, $op$ executes only one SC of type 3, which contradicts the fact that $op$ executes both $SC_3$ and $SC_3^j$.

■

We continue to prove that the $SC$ instructions executed on $rn[z]$ by any operation $op \neq op_j$ between $SC_1^j$ and $SC_1^{j+1}$ fail.

**Lemma 3.4** *Let $op \neq op_j$ be any operation. Then, no successful SC is executed by $op$ on $rn[z]$ between $SC_1^j$ and $SC_1^{j+1}$.*

**Proof:** By definition, no successful $SC$ of type 1 is executed between $SC_1^j$ and $SC_1^{j+1}$. Assume, by the way of contradiction, that $op$ executes a successful SC of type 2 or 3 on $rn[z]$ between $SC_1^j$ and $SC_1^{j+1}$. Let $SC'$ be the first of these successful SC instructions.

Lemma 3.2 implies that $op$ executes a successful SC of type 1 on some node of the red tree before $SC'$; let $SC_1$ be this instruction. By the pseudocode (lines 4, 17), $SC_1$ is executed on the same node as $SC'$, namely on node $rn[z]$. Since $op \neq op_j$ and $SC_1^j$ is executed by $op_j$, $SC_1 \neq SC_1^j$. Since no successful SC of type 1 is executed between $SC_1^j$ and $SC_1^{j+1}$, it follows that $SC_1$ is executed before $SC_1^j$.

If $j = 1$, this is a contradiction since, by definition, $SC_1^j$ is the first successful $SC$ of type 1 on $rn[z]$. If $j > 1$, let $SC_1'$ be the first successful SC of type 1 on $rn[z]$ following $SC_1$. Then, $SC_1'$ is either $SC_1^j$ or some earlier successful $SC$ of type 1 on $rn[z]$. Lemma 3.3 implies that $op$ executes a successful SC of type 3 on $rn[z]$ between $SC_1$ and $SC_1'$; denote by $SC_3$ this $SC$ instruction. Then, $SC_3$ precedes $SC'$, so $SC_3 \neq SC'$. By the pseudocode, $SC_3$ is the only $SC$ of type 3 executed by $op$. Moreover, by the pseudocode, $op$ executes only one SC of type 2 and it does so between $SC_1$

and $SC_3$; let $SC_2$ be this $SC$ instruction. Then, $SC_2 \neq SC'$. It follows that $SC'$ can be neither the type 2 nor the type 3 $SC$ instruction executed by *op*, which contradicts our assumption. ■

Recall that, by the pseudocode, *op* executes exactly one $SC$ of type 3. Thus, Lemmas 3.3 and 3.4 immediately imply the following observation.

**Observation 3.5** *For each $j \geq 1$, $SC_3^j$, executed by $op_j$, is the only successful SC of type 3 executed on $rn[z]$ between $SC_1^j$ and $SC_1^{j+1}$.*

It is now easy to prove that between any successful SC of type 1 and the following successful SC of type 3 on $rn[z]$, there is exactly one successful SC of type 2 on $rn[z]$ executed by the same process.

**Lemma 3.6** *For each $j \geq 1$, there is exactly one successful SC of type 2, namely $SC_2^j$, on the red node with id $z$ between $SC_1^j$ and $SC_1^{j+1}$, and $SC_2^j$ is executed by $op_j$ between $SC_1^j$ and $SC_3^j$.*

**Proof:** By Observation 3.5, $SC_3^j$ is executed by $op_j$. By the pseudocode, $op_j$ executes exactly one $SC$ of type 2, namely $SC_2^j$, and this happens between $SC_1^j$ and $SC_3^j$. Moreover, the only $SC$ of type 3 executed by $op_j$ is $SC_3^j$. Let $LL_2^j$ be the matching $LL$ instruction to $SC_2^j$. By the pseudocode (lines 4, 13, 14), it follows that $LL_2^j$ is executed after $SC_1^j$. Lemma 3.4 implies that no successful $SC$ is executed on $rn[z]$ by any operation $op \neq op_j$ between $SC_1^j$ and $SC_1^{j+1}$. It follows that $SC_2^j$ succeeds and it is the only successful $SC$ of type 2 executed on $rn[z]$ between $SC_1^j$ and $SC_1^{j+1}$. Since in addition $SC_2^j$ is executed by $op_j$ between $SC_1^j$ and $SC_3^j$, the claim follows. ■

Lemmas 3.5 and 3.6 and the pseudocode (line 4) imply that each operation *op* occupies exactly one red node during its execution. We denote by $z_r(op)$ the id of this red node; whenever *op* is clear from the context, we abuse notation and use $z_r$ instead of $z_r(op)$.

**Observation 3.7** *(1) Assume that $C$ is some configuration at which a process $p$, performing some operation op, has executed the type 1 SC instruction of op but it has not yet executed its type 3 SC. Then, there exists exactly one integer $z_r$, $1 \leq z_r \leq 2 * n - 1$, such that $p$ occupies the red node with id $z_r$ at $C$. (2) Assume that $C$ is some configuration at which a process $p$ does not execute any operation. Then, for each integer $z$, $1 \leq z \leq 2 * n - 1$, $p$ does not occupy the red node with id $z$ at $C$.*

We continue to study the properties of the execution portion of an operation that it traverses the blue tree. Intuitively, we prove that for each operation *op* that occupies a red node with id $z_r$, *op_tp*(*op*), the operation type of *op* will be recorded into all nodes of the path starting from the blue node with id $z_r$ up to the blue root (Lemma 3.9). Therefore, *op_tp*(*op*) is eventually recorded into the root node of the blue tree.

Consider any integer $z$, $1 \leq z \leq 2n-1$, and let $level(z) = lg(n) - \lfloor lg(z) \rfloor + 1$, i.e., $level(z)$ is the level of the node with id $z$ in any of the trees. For the rest of this section, let *op* be any instance of `apply_op` executed by some process $p$. Assume that the operation type of *op* is *op_tp*(*op*). By Lemma 3.2, *op* executes a successful SC of type 1 on some node with id $z_r(op)$ of the red tree. By the pseudocode (line 4), this is the only SC of type 1 executed by *op*. Let $pt(op)$ be the path of the blue tree from the node with id $z_r$ to the root. For each $h, level(z_r) \leq h \leq lg(n) + 1$, denote by $z_h$ the id of the node of $pt(op)$ at level $h$; notice that when $h = level(z_r)$, $z_h = z_r$. The following observation is an immediate consequence of the pseudocode (lines 20, 23).

**Observation 3.8** *Let $\pi$ be the execution of any instance of* `propagate` *by op. Then, $\pi$ executes two SC instructions on every node of pt(op).*

Let $\pi_1(op)$ and $\pi_2(op)$ be the two instances of `propagate`, executed by *op*, in order. By Observation 3.8, for each $i \in \{1, 2\}$, $\pi_i(op)$ executes two SC instructions on each node of $pt(op)$. For each $h$, $level(z_r) \leq h \leq \log n + 1$, denote by $C'_{i,h}(op)$ the configuration immediately following the execution of the second of these SC instructions (line 23) on node $bn[z_h]$ by $\pi_i(op)$. Let $opt[1] = op\_tp(op)$ and $opt[2] = \bot$. Denote by $C_1(op)$ the configuration just after the successful $SC$ by *op* that writes the pair $(op\_tp(op), p)$ into $rn[z_r]$, and let $C_2(op)$ be the configuration just after the successful $SC$ by *op* that writes the pair $(\bot, p)$ into $rn[z_r]$. Similarly, let $C_3(op)$ be the configuration just after the successful $SC$ by *op* that writes the pair $(\bot, -1)$ into $rn[z_r]$. In case $h = level(z_r)$, let $C_{1,h-1}(op) = C_1(op)$ and $C_{2,h-1}(op) = C_2(op)$. For simplicity of presentation, we sometimes omit *op* from the above notation if it is clear from the context.

Obviously, for two distinct operations $op, op'$, it might hold that $op\_tp(op) = op\_tp(op')$. However, whenever we mention $op\_tp(op)$ below, we refer to the specific instance of $op\_tp$ that consists the parameter of the particular instance *op* of `apply_op`.

**Lemma 3.9** *For each $i \in \{1, 2\}$, and for each $h$, $level(z_r) \leq h \leq \log n + 1$, there is a configuration $C_{i,h}(op)$ such that:*

1. *$C_{i,h}(op)$ follows $C_{i,h-1}(op)$ and comes before or at $C'_{i,h}(op)$;*
2. *$C_{1,h}(op)$ is the first configuration at which $op\_tp(op)$ is contained in $bn[z_h].ops\_tp[p]$, and at each configuration between $C_{1,h}(op)$ and $C_{2,h}(op)$, $bn[z_h].ops\_tp[p]$ contains $op\_tp(op)$;*
3. *the value $\bot$ is contained in $bn[z_h].ops\_tp[p]$ at $C_{2,h}(op)$, as well as at each configuration between $C_{2,h}(op)$ and $C_3(op)$.*

**Proof:** The proof is by induction on $h$. Fix any $h$, $level(z_r) \leq h \leq \log n + 1$ and assume that the claim holds for each $h', level(z_r) \leq h' < h$. We prove that the claim holds for $h$.

Fix any $i \in \{1, 2\}$. Recall that $p$ is the process executing *op*. By Observation 3.8, $\pi_i$ executes two SC instructions on the blue node with id $z_h$. Let $SC_{i,1}$, $SC_{i,2}$ be these two SC instructions and let $LL_{i,1}$, $LL_{i,2}$ be the matching $LL$ instructions, respectively. By the pseudocode, $\pi_i$ reads $rn[z_h]$ and $bn[z_{h-1}]$ during the execution of any of the instances of its `calculate`. We prove that *op* calculates the value $opt[i]$ as the new value of $bn[z_h].ops\_tp[p]$.

Assume first that $h = level(z_r)$. By definitions of $z_r$ and $C_{i,h-1}$ (when $h = level(z_r)$), and by the pseudocode (lines 4, 14), the pair $(opt[i], p)$ was written into $rn[z_r]$ at $C_{i,h-1}$, and therefore before the execution of $LL_{i,1}$ and $LL_{i,2}$. By the pseudocode, and by Lemmas 3.5 and 3.6, it follows that $rn[z_r]$ contains the pair $(opt[i], p)$ until the execution by *op* of the SC of type $(i + 1)$ which comes after the final configuration of $\pi_i$. Therefore, each of the reads of $rn[z_r]$ by $\pi_i$ following $LL_{i,1}$ and $LL_{i,2}$ returns $opt[i]$ for process $p$ and, by the pseudocode (lines $27 - 28$), `calculate` writes $opt[i]$ for $p$ in the new set of operation types it calculates.

Assume now that $h > level(z_r)$, so $z_h \neq z_r$. Then, Observation 3.7 (claim 1) implies that $rn[z_h].pid \neq p$. By the pseudocode (lines $29 - 30$), it follows that `calculate` will consider, as the new operation type for $p$, the value read for $p$ in $bn[z_{h-1}]$. By the induction hypothesis, there is a configuration $C_{i,h-1}(op)$ in which $opt[i]$ is written in $bn[z_{h-1}].ops\_tp[p]$, and $C_{i,h-1}(op)$ precedes $C'_{i,h-1}$. We argue that $opt[i]$ is contained into $bn[z_{h-1}].ops\_tp[p]$ from $C_{i,h-1}(op)$ until the execution by $op$ of the SC instruction of type $(i+1)$ (which occurs after the final configuration of $\pi_i$). If $i = 1$, this is implied by the induction hypothesis (claim 2) and because $C_{2,h-1}(op) = C_2(op)$ when $h = level(z_r)$. If $i = 2$ this is immediate from the induction hypothesis (claim 3). By the definition of $C'_{i,h-1}$, and by the pseudocode, it follows that $LL_{i,1}$ and $LL_{i,2}$ occur between $C'_{i,h-1}$ and the final configuration of $\pi_i$. Thus, when $bn[z_{h-1}]$ is read between $LL_{i,1}$ and $SC_{i,1}$ (or $LL_{i,2}$ and $SC_{i,2}$), $opt[i]$ is found in $bn[z_{h-1}].ops\_tp[p]$.

If any of the $SC_{i,1}$ or $SC_{i,2}$ is successful, then $opt[i]$ is recorded into $bn[z_h].ops\_tp[p]$ by this successful SC.

Assume now that both $SC_{i,1}$ and $SC_{i,2}$ fail. Since $SC_{i,2}$ fails, it must be that, between $LL_{i,2}$ and $SC_{i,2}$ there is at least one successful SC instruction on $bn[z_h]$. Let $SC'_{i,2}$ be the first of these instructions, and let $op'_i \neq op$ be the operation that executes $SC'_{i,2}$. Let $LL'_{i,2}$ be the matching $LL$ instruction to $SC'_{i,2}$. Since $SC_{i,1}$ fails, it must be that between $LL_{i,1}$ and $SC_{i,1}$, there is at least one successful SC instruction on $bn[z_h]$; let $SC'_{i,1}$ be any of them. $LL'_{i,2}$ follows $LL_{i,1}$, since otherwise $SC'_{i,1}$, which follows $LL_{i,1}$, would cause $SC'_{i,2}$ to fail.

By the pseudocode, $op'_i$ reads $rn[z_h]$ and $bn[z_{h-1}]$ during the execution of its `calculate` between $LL'_{i,2}$ and $SC'_{i,2}$. Recall that $op$ occupies $z_r$. We prove that $op'_i$ calculates the value $opt[i]$ as the new value of $bn[z_h].ops\_tp[p]$.

Assume first that $h = level(z_r)$, so $z_h = z_r$. Recall that the pair $(opt[i], p)$ was written into $rn[z_r]$ before the execution of $LL_{i,1}$ and $LL_{i,2}$; moreover, $rn[z_r]$ contains the pair $(opt[i], p)$ until the execution of the SC instruction of type $(i+1)$ by $op$ which comes after the final configuration of $\pi_i$. By the pseudocode (line 20), $op'_i$ reads $rn[z_r]$ after $LL'_{i,2}$ (which follows $LL_{i,1}$) and before $SC'_{i,2}$ (which precedes $SC_{i,2}$ and the final configuration of $\pi_i$) . Thus, $op'_i$ reads the pair $(opt[i], p)$ in $rn[z_r]$. So, by the pseudocode (lines $27-28$), the instance of `calculate` executed by $op'_i$ between $LL'_{i,2}$ and $SC'_{i,2}$, stores $opt[i]$ for $p$ in the new set of operation types it calculates and $op'_i$ writes $opt[i]$ into $bn[z_h].ops\_tp[p]$ when it executes $SC'_{i,2}$.

Assume now that $h > level(z_r)$, so $z_h \neq z_r$. Then, Observation 3.7 (claim 1) implies that $rn[z_h].pid \neq p$. By the pseudocode (lines $28-30$), it follows that the instance of `calculate` executed by $op'_i$ between $LL'_{i,2}$ and $SC'_{i,2}$, will consider as the new operation type of $p$ the value read in $bn[z_{h-1}].ops\_tp[p]$. By the induction hypothesis, there is a configuration $C_{i,h-1}$ in which $opt[i]$ is written into $bn[z_{h-1}].ops\_tp[p]$, and $C_{i,h-1}$ precedes $C'_{i,h-1}$; moreover, $opt[1]$ is contained in $bn[z_{h-1}]$ from $C_{1,h-1}$ until $C_{2,h-1}$ (which follows the final configuration of $\pi_1$), and $opt[2]$ is contained in $bn[z_{h-1}]$ from $C_{2,h-1}$ until the execution of the SC instruction of type $(i+1)$ by $op$ (which follows the final configuration of $\pi_2$). Thus, in either case $opt[i]$ is contained in $bn[z_{h-1}].ops\_tp[p]$ from $C_{i,h-1}$ until the final configuration of $\pi_i$. By the definition of $C'_{i,h-1}$, and by the pseudocode, it follows that $LL_{i,1}$ and $SC_{i,2}$ occur between $C'_{i,h-1}$ and the final configuration of $\pi_i$. Thus, when $bn[z_{h-1}]$ is read between $LL'_{i,2}$ (which follows $LL_{i,1}$) and $SC'_{i,2}$ (which precedes $SC_{i,2}$), $opt[i]$ is found in $bn[z_{h-1}].ops\_tp[p]$. So, by the pseudocode (lines $27-28$), $opt[i]$ i stored for $p$ in the new set of operation types calculated by $op'_i$ and is written into $bn[z_h].ops\_tp[p]$ by $SC'_{i,2}$.

In both cases, we conclude that there is at least one configuration between $SC_{i,1}$ and $SC_{i,2}$ at which the value $opt[i]$ is written into $bn[z_h].ops\_tp[p]$.

Let $SC_{i,h}(op)$ be the first successful SC that writes $opt[i]$ into $bn[z_h].ops\_tp[p]$ and follows $C_{i,h-1}(op)$; let $C_{i,h}(op)$ be the configuration immediately following the execution of $SC_{i,h}(op)$. Then, it follows that $SC_{i,h}(op)$ precedes $SC_{i,2}$. (We remark that $SC_{i,h}(op)$ may also precede $SC_{i,1}$.) Since, by definition, $C'_{i,h}$ follows $SC_{i,2}$, $SC_{i,h}(op)$ precedes $C'_{i,h}$. Thus, $C_{i,h}(op)$ comes before or at $C'_{i,h}$. By definition, $SC_{i,h}(op)$ follows $C_{i,h-1}(op)$, so $C_{i,h}(op)$ follows $C_{i,h-1}(op)$. This concludes the proof of claim 1.

We continue to prove claim 2. We argue that the first time that $opt[1]$ appears in $bn[z_{h-1}].ops\_tp[p]$ is at $C_{i,h-1}$. If $h = lever(z_r)$, this is implied by the pseudocode and by the fact that $C_{1,h-1}(op) = C_1(op)$ in this case. If $h > level(z_r)$, this is implied by the induction hypothesis (claim 2); moreover, in this case, Observation 3.7 implies that $rn[z_h].pid \neq p$ at all configurations starting from $C_1(op)$ until $C_3(op)$ (which comes after $C_{i,h-1}(op)$). Thus, by the pseudocode, it follows that no SC can write $opt[1]$ into $bn[z_h].ops\_tp[p]$ before $C_{i,h-1}(op)$. Since (by definition) $SC_{1,h}(op)$ is the first successful SC that writes $opt[1]$ into $bn[z_h].ops\_tp[p]$ after $C_{i,h-1}(op)$, $C_{1,h}(op)$ is the first configuration at which $opt[1]$ is contained in $bn[z_h].ops\_tp[p]$.

By claim 1 proved above, $C_{1,h}$ precedes the final configuration of $\pi_1$; moreover, $C_{2,h}$ follows $C_2(op)$ which comes after the final configuration of $\pi_1$. Thus, $C_{2,h}$ follows $C_{1,h}$.

Assume, by the way of contradiction, that there is a configuration $C_l$ between $C_{1,h}$ and $C_{2,h}$, such that $bn[z_h].ops\_tp[p]$ contains a value $x \neq opt[1]$ at $C_l$. By definition of $C_l$, there is at least a successful SC instruction that writes $x$ into $bn[z_h].ops\_tp[p]$ and occurs between $C_{1,h}$ and the configuration that precedes $C_{2,h}$. Let $SC'_1$ be the first of these instructions, let it be executed by operation $op''_1 \neq op$ and let $LL'_1$ be its matching $LL$ instruction. Recall that $SC_{1,h}$ is the successful SC instruction executed just before $C_{1,h}$. Since $SC'_1$ is a successful SC instruction, $LL'_1$ must follow $SC_{1,h}$. By the pseudocode, $op''_1$ reads $rn[z_h]$ and $bn[z_{h-1}]$ during the instance of `calculate` executed between $LL'_1$ and $SC'_1$. Recall that $p$ occupies $z_r$.

Assume first that $h = level(z_r)$. Then, the pseudocode (lines 4 or 14), and Lemmas 3.5 and 3.6 imply that $op''_1$ reads either the pair $(opt[1], p)$ or the pair $(opt[2], p)$ into $rn[z_r]$ (depending on whether the read happens between $C_{1,h}$ and $C_{2,h-1}$ or between $C_{2,h-1}$ and the configuration that precedes $C_{2,h}$, respectively) when executing `calculate` between $LL'_1$ and $SC'_1$. Assume now that $h > level(z_r)$, so $z_h \neq z_r$. Then, Observation 3.7 (claim 1) implies that $rn[z_h].pid \neq p$. By the pseudocode (lines $28-30$), it follows that $op''_1$ will consider as the new operation type for $p$, the value read in $bn[z_{h-1}].ops\_tp[p]$ when executing `calculate` between $LL'_1$ and $SC'_1$. By the induction hypothesis (claim 3), $op\_tp(op)$ is contained in $bn[z_{h-1}].ops\_tp[p]$ at all configurations between $C_{1,h-1}(op)$ and $C_{2,h-1}(op)$. Thus, $op''_1$ reads either the pair $(opt[1], p)$ or the pair $(opt[2], p)$ (depending on whether the read happens between $C_{1,h}$ and $C_{2,h-1}$ or between $C_{2,h-1}$ and the configuration that precedes $C_{2,h}$, respectively) in $bn[z_h]$ when executing `calculate` between $LL'_1$ and $SC'_1$. We conclude that this is so in either case.

If $op''_1$ reads the pair $(opt[1], p)$, by the pseudocode, it follows that $op'$ writes the value $opt[1]$ into $bn[z_h].ops\_tp[p]$ when it executes $SC'_1$. This is a contradiction to our assumption that $SC'_1$ writes $x \neq opt[1]$ into $bn[z_h].ops\_tp[p]$. Thus, assume that $op''_1$ reads the pair $(\perp, p)$. Then, by the pseudocode, it follows that $op''_1$ writes $\perp$ into $bn[z_h].ops\_tp[p]$ when it executes $SC'_1$. Recall that, in this case, the read by $op''_1$ that occurs between $LL'_1$ and $SC'_1$ must take place after $C_{2,h-1}(op)$. Therefore, $SC'_1$ occurs after $C_{2,h-1}(op)$ and before $SC_{2,h}(op)$. Then, it is $SC'_1$ (and not $SC_{2,h}(op)$) the first $SC$ after $C_{2,h-1}(op)$ that writes $\perp$ into $bn[z_h].ops\_tp[p]$, which contradicts the definition of $SC_{2,h}(op)$.

We continue to prove claim 3. By definition of $C_{2,h}$, $\perp$ is contained in $bn[z_h].ops\_tp[p]$ at $C_{2,h}$. So, we continue to prove that $\perp$ is contained in $bn[z_h].ops\_tp[p]$ at each configuration between $C_{2,h}$ and $C_3(op)$.

By the definitions of $C'_{2,h}$ and $C_3(op)$, and by claim 1 proved above, $C_3(op)$ follows $C_{2,h}$. Assume, by the way of contradiction, that there is a configuration $C_l$ between $C_{2,h}$ and $C_3(op)$, such that $bn[z_h].ops\_tp[p]$ contains a value $x \neq \perp$ at $C_l$.

By definition of $C_l$, there is at least a successful SC that writes $x$ into $bn[z_h].ops\_tp[p]$ and occurs between $C_{2,h}$ and the configuration that precedes $C_3(op)$. Let $SC'_2$ be the first of these instructions, let it be executed by operation $op''_2 \neq op$ and let $LL'_2$ be its matching $LL$ instruction. Recall that $SC_{2,h}$ is the successful SC instruction executed just before $C_{2,h}$. Since $SC'_2$ is a successful SC instruction, $LL'_2$ follows $SC_{2,h}$.

If $h = level(z_r)$, Lemmas 3.5 and 3.6 imply that the read of $rn[z_h]$ by $op''_1$, which occurs between $LL'_2$ and $SC'_2$ (at the beginning of the execution of its `calculate`) returns $(\perp, p)$. Thus, by the pseudocode, $op''_2$ writes $\perp \neq x$ into $bn[z_h].ops\_tp[p]$ by executing $SC'_2$, which is a contradiction.

Assume now that $h > level(z_r)$. Then, Observation 3.7 (claim 1) implies that $rn[z_h].pid \neq p$. By the pseudocode (lines $28 - 30$), it follows that the instance of `calculate` executed by $op''_2$ between $LL'_2$ and $SC'_2$, will consider as the new operation type of $p$ the value read in $bn[z_{h-1}].ops\_tp[p]$. By the induction hypothesis, there is a configuration $C_{2,h-1}$ at which $\perp$ is written into $bn[z_{h-1}].ops\_tp[p]$, and $C_{2,h-1}$ precedes $C'_{2,h-1}$; moreover, $\perp$ is contained in $bn[z_{h-1}].ops\_tp[p]$ from $C_{2,h-1}(op)$ until $C_3(op)$. Thus, the read of $bn[z_h]$ by $op''_2$, which occurs between $LL'_2$ and $SC'_2$ returns the value $\perp$ for $p$. Thus, by the pseudocode, $op''_2$ writes $\perp \neq x$ into $bn[z_h].ops\_tp[p]$ by executing $SC'_2$, which is a contradiction. ■

We remark that information about $op$ (namely, $op\_tp(op)$ and the id $p$ of the process that executes it) is recorded for the first time into one of the registers when $op$ occupies its red node $z_r$. Lemma 3.9 implies that it is then transferred to the blue node with id $z_r$ (by $op$ or some other operation); moreover, only when it is written there, it can be forwarded to the next node of the blue path of $op$. This transfer continues up to each node of $op$'s blue path until the operation type of $op$ eventually reaches the root. Recall that for each $h$, $level(z_r) \leq h \leq \log n + 1$, $op\_tp(op)$ is written into the node $z_h$ at level $h$ of $pt(op)$ by $SC_{1,h}(op)$ just before $C_{1,h}(op)$, and $\perp$ is written into $bn[z_h]$ by $SC_{2,h}(op)$ just before $C_{2,h}(op)$.

The following claim is an immediate consequence of Lemma 3.9 when $h = \log n + 1$.

**Corollary 3.10** *The operation type of any operation op is successfully recorded in the blue root by $SC_{1,\log n+1}(op)$.*

We continue to prove that the value $\perp$ is contained for some process $p$ in a blue node $z$ from the time that an operation by $p$ writes $\perp$ into $z$ until the time that the subsequent operation by $p$ (for which $z$ is contained in its path) writes its operation type into $z$ (or until the final configuration if such an operation does not exist).

**Lemma 3.11** *Consider any operation op executed by some process p. For each $z_h \in pt(p)$, let $C_h = C_{1,h}(op_{m_h})$ if p executes a subsequent operation $op_{m_h}$ such that $z_h \in pt(op_{m_h})$; let $C_h$ be the final configuration if such an operation does not exist. Then, $bn[z_h].ops\_tp[p] = \perp$ at each configuration between $C_3(op)$ and $C_h$.*

**Proof:** Assume, by the way of contradiction, that the claim does not hold and let $C$ be the first configuration at which the claim is violated. Let $op$ be the operation (let it be executed by some process $p$) and $z_h \in pt(p)$ be the node that causes this violation. More specifically, if $op_{m_h}$ is the first operation executed by $p$ after $op$ for which $z_h \in pt(op_{m_h})$, then $C$ is between $C_3(op)$ and $C_{1,h}(op_{m_h})$ and $bn[z_h].ops\_tp[p] = x \neq \perp$ at $C$ (if such an operation does not exist, then $C$ is between $C_3(op)$ and the final configuration). Assume that $SC'$ is the SC instruction executed just before $C$ which writes the value $x$ in $bn[z_h].ops\_tp[p]$ and let $op'$ be the operation that executes $SC'$. Denote by $LL'$ the corresponding $LL$ instruction to $SC'$.

Assume first that $op'$ reads $x$ in $rn[z_h].op\_tp$. Notice that $op'$ performs this read (let it be $r'$) at some configuration $C'$ that precedes $C$. By the pseudocode, it follows that $rn[z_h].pid = p$ and $rn[z_h].op\_tp = x$ at $C'$. Thus, $p$ is active executing some operation $op''$ at $C'$ such that $z_h$ is the first node in $pt(op'')$.

If $z_h \in pt(op)$, let $op_{b_h} = op$; otherwise, let $op_{b_h}$ be the last operation by $p$ preceding $op$ such that $z_h \in pt(op_{b_h})$. (Since $z_h \in pt(op'')$, $op_{b_h}$ is well-defined.) By the pseudocode, $LL'$ happens before $r'$. If $C'$ precedes $C_{2,h}(op_{b_h})$, then $LL'$ precedes $C_{2,h}(op_{b_h})$. Since, by Lemma 3.9, $SC_{2,h}(op_{b_h})$ happens just before $C_{2,h}(op_{b_h})$ and it is a successful $SC$ on $bn[z_h]$, $SC'$ cannot be successful. This is a contradiction. Thus, $C'$ follows $C_{2,h}(op_{b_h})$. Lemmas 3.3 and 3.6 imply that $rn[z_h] = (\bot, p)$ at $C_{2,h}(op_{b_h})$. By definition of $op_{b_h}$, no other operation following $op_{b_h}$ and containing $z_h$ in its path is executed by $p$ before $op_{m_h}$. It follows that, at each configuration between $C_{2,h}(op_{b_h})$ and $C'$, $rn[z_h] \neq (x, p)$. This is a contradiction (since we have assumed that $op'$ reads $(x, p)$ in $rn[z_h]$ at $C'$).

Assume now that $op'$ reads $x$ in $bn[z_{h-1}].ops\_tp[p]$ (where $z_{h-1}$ is the node preceding $z_h$ in $pt(p)$); let $r''$ be this read. Notice that $r''$ results in some configuration $C''$ which precedes $C$. Let $op_{m_{h-1}}$ be the first operation executed by $p$ after $op$ such that $z_{h-1} \in pt(op_{m_{h-1}})$. If $z_{h-1} \in pt(op)$, let $op_{b_{h-1}} = op$; otherwise, let $op_{b_{h-1}}$ be the last operation by $p$ preceding $op$ such that $z_{h-1} \in pt(op_{b_{h-1}})$. In case $op_{b_{h-1}}$ does not exist, denote by $C_{2,h-1}(op_{b_{h-1}})$ the initial configuration. Similarly, in case $op_{b_h}$ does not exist, let $C_{2,h}(op_{b_h})$ be the initial configuration. If $op_{m_{h-1}}$ does not exist, denote by $C_{1,h-1}(op_{m_{h-1}})$ the final configuration. Similarly, in case $op_{m_h}$ does not exist, let $C_{1,h}(op_{m_h})$ be the final configuration.

Since $z_h$ is an ancestor of $z_{h-1}$ in the blue tree, it follows that $z_h \in pt(op_{m_{h-1}})$ and $z_h \in pt(op_{b_{h-1}})$. Thus, either $op_{b_{h-1}} = op_{b_h}$ or $op_{b_{h-1}}$ precedes $op_{b_h}$. Similarly, either $op_{m_{h-1}} = op_{m_h}$ or $op_{m_{h-1}}$ follows $op_{m_h}$.

Assume first that $op_{m_{h-1}}$ follows $op_{m_h}$, so that $C$ precedes $C_{1,h-1}(op_{m_{h-1}})$. By the pseudocode, $LL'$ happens before $r''$. Obviously, $C''$ follows the initial configuration. If $C''$ precedes $C_{2,h}(op_{b_h})$, then $LL'$ precedes $C_{2,h}(op_{b_h})$. Since, by Lemma 3.9, $SC_{2,h}(op_{b_h})$ happens just before $C_{2,h}(op_{b_h})$ and it is a successful $SC$ on $bn[z_h]$, $SC'$ cannot be successful. This is a contradiction. Thus, $C''$ follows $C_{2,h}(op_{b_h})$.

We prove that, at each configuration between $C_{2,h}(op_{b_h})$ and $C_{1,h-1}(op_{m_h})$, $bn[z_{h-1}].ops\_tp[p] = \bot$. In case $op_{b_{h-1}} = op_{b_h}$, Lemma 3.9 imply that $C_{2,h-1}(op_{b_h})$ precedes $C_{2,h}(op_{b_h})$, and at each configuration between $C_{2,h-1}(op_{b_h})$ and $C_3(op_{b_h})$, $bn[z_{h-1}].ops\_tp[p] = \bot$. Otherwise, recall that $op_{b_{h-1}}$ precedes $op_{b_h}$ and they are both executed by $p$, so $C_3(op_{b_{h-1}})$ precedes $C_{2,h}(op_{b_h})$. Since $C$ is the first configuration at which the claim of the lemma is violated, it must be that, at each configuration between $C_3(op_{b_{h-1}})$ and $C_{1,h-1}(op_{m_{h-1}})$, $bn[z_{h-1}].ops\_tp[p] = \bot$.

Assume that $C$ precedes $C_{1,h-1}(op_{m_{h-1}})$. Then, at each configuration between $C_3(op_{b_{h-1}})$ and $C$, $bn[z_{h-1}].ops\_tp[p] = \bot$. It follows that $r''$ reads $\bot \neq x$ in $bn[z_{h-1}].ops\_tp[p]$ and writes $\bot \neq x$ in $bn[z_h].ops\_tp[p]$, which is a contradiction.

Assume now that $C$ follows $C_{1,h-1}(op_{m_{h-1}})$. If $op_{m_{h-1}}$ follows $op_{m_h}$, then $C$ precedes $C_{1,h-1}(op_{m_{h-1}})$, so it must be that $op_{m_{h-1}} = op_{m_h}$. Since (1) at each configuration between $C_{2,h}(op_{b_h})$ and $C_{1,h-1}(op_{m_{h-1}}) = C_{1,h-1}(op_{m_h})$, $bn[z_{h-1}].ops\_tp[p] = \bot$, (2) $r''$ occurs after $C_{2,h}(op_{b_h})$, and (3) $r''$ returns $x \neq \bot$, it must be that $r''$ occurs after $C_{1,h-1}(op_{m_h})$. Thus, $r''$ occurs between $C_{1,h-1}(op_{m_h})$ and $C_{1,h}(op_{m_h})$. Lemma 3.9 implies that, at all configuration between $C_{1,h-1}(op_{m_h})$ and $C_{1,h}(op_{m_h})$,

$bn[z_{h-1}].ops\_tp[p] = op\_tp(op_{m_h})$. Thus, $r''$ reads $op\_tp(op_{m_h})$ in $bn[z_{h-1}].ops\_tp[p]$ and writes this value in $bn[z_h].ops\_tp[p]$ by executing $SC'$. However, Lemma 3.9 implies that the first configuration after $C_{1,h-1}(op_{m_h})$ at which $op\_tp(op_{m_h})$ is written into $bn[z_h].ops\_tp[p]$ is $C_{1,h}(op_{m_h})$. Since we have assumed that $C$ precedes $C_{1,h}(op_{m_h})$, this is a contradiction. ■

**Linearizability.** Linearizability imposes a total order, called *linearization order*, to all operations performed in an execution $\alpha$. If $\alpha$ is linearizable, each of its operations should have the same response as the corresponding operation in the serial execution derived by the linearization order. When this holds for some operation, we say that the response of the operation is *consistent.* When this holds for all operations of $\alpha$, we say that $\alpha$ is *consistent.*

We are now ready to assign linearization points to the operations of F-RedBlue. By Observation 3.10, there is at least one successful SC that records the operation type of *op* in the blue root node. Recall that $SC_{1,\log n+1}(op)$ is the first of these SC instructions. We place the linearization point of *op* just before $SC_{1,\log n+1}(op)$; ties are broken by the order that process identifiers impose.

**Lemma 3.12** *For each operation op, the linearization point of op is placed in its execution interval.*

**Proof:** By Lemma 3.9, the operation type of *op* is recorded in the root node by some SC instruction and this occurs before the execution of the type 2 SC instruction by *op* (line 14). Thus, the linearization point of *op* precedes the end of its execution interval.

Let $SC(op)$ be the first successful SC that records the operation type of *op* in the blue root node. Notice that *op* is invisible to all processes until it performs its first store operation, writing its information into the red node that it occupies. Thus, $SC(op)$ must follow the beginning of the execution interval of *op*. ■

We say that an operation *op is applied* on the simulated object if (1) procedure `calculate`, executed by some operation $op'$ (that might be *op* or another operation), reads in the appropriate child node of the blue root $op\_tp(op)$ (i.e., the operation type written there for *op*) and records it as the new operation type for $p$, (2) `calculate` by $op'$ calls `apply` with this operation type in its parameters, and (2) the execution of the SC of line 23 (let it be $SC_r$) on $bn[1]$ by $op'$ succeeds (thus writing there for $p$ the value $op\_tp(op)$). When these conditions are satisfied, we sometimes also say that $op'$ applies *op* on the simulated object or that $SC_r$ applies *op* on the simulated object. We next prove that each operation *op* is applied on the simulated object exactly once.

**Lemma 3.13** *Each operation op is applied on the simulated shared object exactly once by* $SC_{1,\log n+1}(op)$.

**Proof:** Assume that *op* is executed by process *p*. We first prove that *op* is applied on the simulated object at least once. Lemma 3.9 implies that $op\_tp(op)$, the operation type of *op*, is successfully recorded in the root node of the blue tree at least once and that $SC_{1,\log n+1}$ is the first $SC$ instruction that stores $op\_tp(op)$ into $bn[1].ops\_tp[p]$. Let $op'$ be the operation that executes $SC_{1,\log n+1}$.

In case $p$ executes an operation before *op*, let $C_{b_h} = C_{2,\log n+1}(op_{b_h})$, where $op_{b_h}$ is the last preceding to *op* operation executed by $p$; otherwise, let $C_{b_h}$ be the initial configuration of the algorithm. If $op'$ performs the read of $bn[1]$ that precedes $SC_{1,\log n+1}$ before $C_{b_h}$, then $SC_{1,\log n+1}$ fails. This is so because, by the pseudocode, the corresponding $LL$ to $SC_{1,\log n+1}$ precedes this read, and, by the definition of $C_{b_h}$ and Lemma 3.9, a successful $SC$ on $bn[1]$ (namely $SC_{2,\log n+1}(op_{b_h})$) occurs at $C_{b_h}$, thus causing the failure of $SC_{1,\log n+1}$. This is a contradiction.

Thus, $op'$ performs its read after $C_{b_h}$. Lemmas 3.9 and 3.11, imply that, at each configuration between $C_{b_h}$ and $C_{1,\log n+1}(op)$, $bn[1].ops\_tp[p] = \perp$. It follows that $op'$ reads $\perp$ into $bn[1].ops\_tp[p]$ during the execution of `calculate` that precedes $SC_{1,\log n+1}(op)$. Since $op\_tp(op) \neq \perp$, $op'$ evaluates the condition of the $if$ statement of line 31 to true. We conclude that $op'$ applies $op$.

We now prove that $op$ is applied at most once on the simulated object. Assume, by the way of contradiction, that $op$ is applied at least twice, and let $SC'$ be the first $SC$ after $SC_{1,\log n+1}(op)$ that applies $op$. Let $op''$ be the operation that executes $SC'$ and let $r'$ be the last read of $bn[1]$ executed by $op''$ before $SC'$.

If $r'$ occurs before $SC_{1,\log n+1}(op)$, then the corresponding $LL$ to $SC'$, which precedes $r'$, precedes also $SC_{1,\log n+1}(op)$. Thus, $SC_{1,\log n+1}(op)$ causes $SC'$ to fail, which is a contradiction. Thus, $r'$ follows $SC_{1,\log n+1}(op)$.

If $r'$ occurs between $C_{1,\log n+1}(op)$ and $C_{2,\log n+1}(op)$, then Lemmas 3.9 and 3.11 imply that $r'$ reads the value $op\_tp(op)$ in $bn[1]$. Since $op''$ apply $op$, it must be that $tmp.ops\_tp[p] = op\_tp(op)$ when $op''$ executes line 31 of the instance of `calculate` that precedes $SC'$. Thus, by the pseudocode (line 31), it follows that the condition of the `if` statement of line 31 is evaluated to `FALSE`. Thus, $op'$ does not call `apply`, which contradicts the fact that $op'$ applies $op$.

Assume finally that $r'$ follows $C_{2,\log n+1}(op)$. Lemma 3.6 and Observation 3.7 imply that $rn[1] \neq op\_pt(op)$ at all configurations after the configuration at which $op$ executes its type 2 $SC$ instruction. Moreover, if any of the root children belongs to $pt(op)$, then Lemma 3.9 imply that $C_{2,\log n}(op)$ precedes $C_{2,\log n+1}(op)$, and $bn[\log n].ops\_pt[p] \neq op\_tp(op)$ after $C_{2,\log n}(op)$. By the pseudocode, it therefore follows that procedure `calculate`, executed by $op''$ before $SC'$, calculates as the new value of $bn[1].ops\_tp[p]$ a value other than $op\_tp(op)$ and therefore it does not apply $op$, which is a contradiction. ■

Denote by $SC_i$ the $i$-th successful $SC$ instruction on the root node of the blue tree. Obviously, between $SC_i$ and $SC_{i+1}$, the *state* field of the root node is not modified.

We are now ready to prove consistency. Let $a$ be any execution of RedBlue. Denote by $a_i$, the prefix of $a$ which ends at $SC_i$. Let $a_0$ be the empty execution, and let $l_i$ be the linearization order of $a_i$.

**Lemma 3.14** *For each $i \geq 0$, $a_i$ is consistent.*

**Proof:** We prove the claim by induction on $i$.
**Base case (i=0):** Execution $a_0$ is empty, so the claim holds trivially.
**Induction hypothesis:** Fix any $i > 0$ and assume that the claim holds for $i-1$.
**Induction step:** We prove that the claim holds for $i$. From the induction hypothesis, it holds that $a_{i-i}$ is consistent with linearization order $l_{i-1}$. Let $op$ be the operation that executes $SC_i$ and assume that $op$ applies $j \leq 0$ operations on the simulated object. Denote by $op_1, ..., op_j$, for some $j \geq 0$, the sequence of these operations ordered with respect to the ids of the processes that initiate them.

We prove the following claim: For each $l$, $0 \leq l \leq j$, operation $op_l$ returns a consistent response. The claim is proved by induction on $l$.

**Base case (l=0):** The claim holds vacuously.

**Induction hypothesis:** Fix any $l > 0$ and assume that the claim holds for $l-1$.

**Induction step:** We prove that the claim holds for $l$. By the induction hypotheses (inner and outer inductions), it follows that all operations in $l_{i-1}op_1, \ldots, op_{l-1}$ return a consistent response. By

Lemma 3.13, $op_l$ is applied exactly once by $SC_{1,\log n+1}(op_l)$. By the way function `apply` operates, it applies $op_l$ on the simulated object just after it has applied $op_1, \ldots, op_{l-1}$ on it. Therefore, the response calculated for $op_l$ by $op$ is consistent, as needed. ■

By the pseudocode and by Observation 3.7, it follows that when the $SC$ of type 1 of $op$ fails on some red node, this node is occupied by some other active operation. Thus, if an operation $op$ occupies a red node at depth $k$, its interval contention is at least $k$. Since the height of the red-tree is $\log n$, it follows that the time complexity of F-RedBlue is $O(min\{k, \log n\})$, where $k$ is the interval contention.

## 4 Modified Version of F-RedBlue that Uses Small Registers

We present S-RedBlue, a modified version of F-RedBlue that uses small registers. Each red node now stores $\log n + 1$ bits, and a blue node, other than the root, stores $n$ bits. The blue root stores $n$ bits, a process id and the state of the object. This $LL/SC$ register is implemented by single-word $LL/SC$ registers using the implementation in [18].

In S-RedBlue, a process $p$ uses a single-writer register to record its currently active operation (line 1). Like in RedBlue, the process starts the execution of any of its operations by traversing the red tree. However, to occupy a red node, the process just records its id and sets the bit of the node to *true*. Similarly, each process, moving up the path to the root of the blue true, just sets a bit in each node of the path to identify that it is currently executing an operation. Thus, the bit array of the root identifies all processes that are currently active.

To avoid storing the return values in the root node, each process $p$ keeps an array of $n$ single-writer registers, one for each process. When $p$ reaches the root node (during the application of one of its operations), it first records in its appropriate single-writer registers the responses for those processes that are currently active (lines $25-26$). Then, it tries to store the new state of the object in the root of the blue tree together with its id, and the set (bit vector) of active processes. A process finds the response for its current operation in the appropriate single-writer register of the process with id the one recorded in the root node.

The state is updated only at the root node and only when the bit value for a process changes from $false$ $(F)$ to *true* $(T)$ in the blue root's bit array (line 23). This guarantees that the operation of each process is applied only once to the simulated object. However, all processes reaching the root, record responses for each currently active process $p$ in their single-writer registers, independently of whether they also apply $p$'s operation to the simulated object. This is necessary, since the operation of $p$ may be applied to the object by some process $q$ and later on (and before $p$ reads the root node for finding its response) another process $q'$ may overwrite the root contents. Process $q'$ will include $p$ in its calculated active set but it will not re-apply $p$'s operation to the object, since it will see that $p$'s bit in the active set of the root node is already set. Still $q'$ should record a response for $p$ in its single-writer registers since $p$ may read $q'$ and not $q$ in $bn[1].pid$ when seeking for its response.

The proof that S-RedBlue is correct closely follows the correctness proof of F-RedBlue. The main difference of the two algorithms is on the way that response values are calculated. If $q$ is the process that applies some operation $op$, the response for $op$ is originally stored in $rvals[q][p]$ and the id of $q$ is written into the root node. The next process to update the root node will find the id of $q$ in the root node and (as long as $op$ has not yet read its response by executing line 8), it will see that $tmp.ops[p] = T$. Therefore, it will copy the response for $op$ from $rvals[q][p]$ (line 26) to its

**Algorithm 2** Pseudocode of S-RedBlue: functions that are different from those of F-RedBlue.

```
struct rnode{
    boolean $op$;
    PINDEX $pid$;
}

struct bnode{
    state $st$; // used only at the root node
    PINDEX $pid$; // used only at the root node
    boolean $ops[n]$;
}

shared struct rnode $rn[1..2n-1]$ = $\{< F, -1 >, ..., < F, -1 >\}$;      // $F$ stands for $False$ and $T$ for $True$
shared struct bnode $bn[1..2n-1]$ = $\{< \perp, -1, < F, ..., F >>, ..., < \perp, -1, < F, ..., F >>\}$;
shared ret_val $rvals[1..n][1..n]$ = $\{\{\perp, \ldots, \perp\}, \ldots, \{\perp, \ldots, \perp\}\}$;
shared operation_type $ops[1..n]$ = $\{\perp, \ldots, \perp\}$;
```

```
ret_val apply_op(operation_type $op\_tp$, PINDEX $p$){
    int $direction$ = $n/2$, $z = 1$;
    int $levels = \lg(n) + 1$, $l$;
    ret_vals $rv$;

1.  $ops[p] = op\_tp$;
2.  for($l$=$levels$;$l \geq 1$;$l$--){
3.      $LL(rn[z])$;
4.      if($rn[z]$ == $< F, -1 >$)
5.          if(SC($rn[z]$, <$op\_tp$, $id$>)) break;
        //find the next node
6.       lines 5-10 of algorithm 1;
    }
7.  propagate($z$,$p$);
8.  $rv$ = $rvals[bn[1].pid][p]$;
9.  $LL(rn[z])$;
10. SC($rn[z]$, <$F$, $p$>);
11. propagate($z$,$p$);
12. $LL(rn[z])$;
13. SC($rn[z]$, $< F, -1 >$ );
14. return $rv$;
}
```

```
struct bnode calculate(int $z$, PINDEX $p$) {
    struct bnode $blue$ = $bn[z]$, $lc$, $rc$;
    struct bnode $tmp$ = $< \perp, -1, < F, \ldots, F >>$;
    struct rnode $red$ = $rn[z]$;

15. if $(2 * z + 1 < 2n)$ { $lc$=$bn[2 * z]$; $rc$=$bn[2 * z + 1]$; }
16. if ($z$ == 1) { $tmp.st$ = $blue.st$; $tmp.pid$ = $p$; }
17. for(int $q = 1$ to $n$ do){
18.     if ($red.pid$ == $q$) $tmp.ops[q] = red.op$;
19.     else if (is_predecessor($z$,$q$,$2 * z$))
20.         $tmp.ops[q] = lc.ops[q]$;
21.     else if (is_predecessor($z$,$q$,$2 * z + 1$))
22.             $tmp.ops[q] = rc.ops[q]$;
23.     if ($z$==1 AND $tmp.ops[q]$==$T$ AND $blue.ops[q]$==$F$)
24.         apply $ops[q]$ to $tmp.st$
            and store into $rvals[p][q]$ the return value;
25.     else if($z$==1 AND $tmp.ops[q]$==T)
26.         $rvals[p][q] = rvals[b.pid][q]$;
    }
27. return $tmp$;
}
```

appropriate single-writer register. So, when $p$ seeks for the response of $op$ it will find the correct answer in the single-writer register of the process recorded at the root node.

S-RedBlue uses $O(n)$ multi-writer $LL/SC$ registers and $O(n^2)$ single-writer $read - write$ registers. One of the multi-writer registers is large and it is implemented using the implementation of a $W$-word $LL/SC$ object from single-word $LL/VL/SC$ objects presented in [18]. This implementation achieves time complexity $O(W)$ for both $LL$ and $SC$ and has space complexity $O(nW)$. Thus, the number of registers used by S-RedBlue is $O(n^2 + nW)$. In common cases where $n$ bits fit in a constant number of single-word registers, the time complexity of S-RedBlue is $O(k + W)$ since `calculate` pays $O(k)$ to record $k$ response values in the single-writer registers of the process executing it, and $O(W)$ for reading and modifying the root node.

## 5 Adaptive Universal Constructions for Large Objects

In the universal constructions for long objects presented by Anderson and Moir in [7] the object is treated like if it were stored in a contiguous array, and the user is supposed to provide sequential implementations of the object's operations which call appropriate `read` and `write` procedures (described in [7]) to perform read or write operations in the contiguous array (see [7, Section 4] for more information on how the user code should look like and an example). The universal constructions partition the contiguous array into $B$ blocks of size $S$ each, and during the application of an operation to the object, only the block(s) that should be modified are copied locally (and not the entire object's state). The authors assume that each operation modifies at most $T$ blocks.

S-RedBlue can easily employ the simple technique of the *lock-free* construction in [7] to provide a simple, adaptive, *wait-free* algorithm (called LS-RedBlue) for large objects. As illustrated in Algorithm 5, only routine `propagate` requires some modifications. Also, similar data structures as those in [7] are needed for storing the array blocks, and having processes making "local" copies of them and storing back the changed versions of these blocks. More specifically, array $BLK$ stores the $B$ blocks of the object's state, as well as a set of *copy* blocks used by the processes for performing their updates without any interference by other processes. Since each operation modifies at most $T$ blocks, a process reaching the blue root, requires at most $kT$ copy blocks in order to make copies of the $kT$ state blocks that it should possibly modify. So, $BLK$ contains $nkT + B$ blocks; initially, the object's state is stored in $BLK[nkT + 1], \ldots, BLK[nkT + B]$ (the blocks storing the state of the object are called *active*). The blue root node now stores an array named $BANK$ of $B$ indices; the $i$th entry of this array is the pointer (i.e., the index in $BLK$) of the $i$th active block. Each process has a private variable $ptrs$ which uses to make a local copy of the $BANK$ array (line 9).

The application of an active operation to the object is now done by calling (in `calculate`) the appropriate sequential code provided by the user. The codes of the `read` and `write` routines used by the user code are also presented in Algorithm 5 (although they are the same as those presented in [7]). These routines take an index *addr* in the contiguous array as a parameter. From this index, the block number *blkidx* that should be accessed is calculated as $blkidx = addr$ div $S$, and the offset in this block as $addr$ mod $S$. The actual index in $BLK$ of the *blkidx*-th block can be found through the $BANK$ array. However, the process uses its local copy $ptrs$ of $BANK$ for doing so. Thus, line 15 simply access the appropriate word of $BLK$. If the execution of the $VL$ instruction of line 16 by some process $p$ does not succeed, the $SC$ instruction of line 11 by $p$ will also not succeed. So, we use the `goto` to terminate the execution of its `calculate`.

The first time that $p$ executes a `write` to the *blkidx*-th block, it copies it to one of its copy blocks (line 21). Array *dirty* is used to identify whether a block is written for the first time by $p$. In this case, the appropriate block is copied into the appropriate copy block of $p$ (line 21). Indices to the $kT$ copy blocks of $p$ are stored in $p$'s private array *copy*. The dirty bit for this block is set to *true* (line 22). Counter *dcnt* counts the number of different blocks written by $p$ thus far in the execution of its current operation (line 25). The appropriate entry of $ptrs$ changes to identify that the *blkidx*-th block is now one of the copy blocks of $p$ (line 23). The write is performed in the copy block at line 27. A process $p$ uses its copy blocks to make copies of the blocks that it will modify. If later $p$'s $SC$ at line 11 is successful, some of $p$'s copy blocks become active blocks (substituting those that have been modified by $p$). These old active blocks (that have been substituted) consist the new copy blocks of $p$ which it will use to perform its next operation. This is accomplished with the code of line 12.

LS-RedBlue is wait-free; it has space overhead $\Theta(n^2 + n(B + kTS))$ and time complexity $\Theta(B + k(D + TS))$. The wait-free universal construction presented in [7] assumes that each process has enough copy blocks to perform at most $M/T$ other operations in addition to its own, where $M \geq 2T$ is any fixed integer. The algorithm uses a quite complicated helping mechanism with return values written into return blocks which should then be recycled in order to keep the memory requirements low. The construction has time complexity $O((n/min\{n, M/T\})(B + nD + MS))$. LS-RedBlue achieves much better time complexity ($\Theta(B + k(D + TS))$) and is adaptive. However, it assumes that processes have enough copy blocks to help any number of other active processes.

LS-RedBlue can be slightly modified to disallow processes to help more than $M/T$ other pro-

**Algorithm 3** Pseudocode of LS-RedBlue.

```
type INDEX $\{1,\ldots,nkT+B\}$;
struct bnode{
    INDEX $BANK[B]$;
    PINDEX $pid$;
    boolean $ops[n]$;
}
void propagate(int $z$, PINDEX $p$){
    bnode $b$;
1.  while($z$!=0){
2.    for(int $i=1$ to 2) do {
3.       if($z$==1){
4.          for(int $j=1$ to $B$) do
5.             $dirty[j]$=$F$;
6.          $dcnt=0$;
7.       }
8.       $b$=$LL(bn[z])$;
9.       if ($z$ == 1) $ptrs = b.BANK$;
10.      $bt$=calculate($z$,$p$);
11.      if(SC($bn[z]$, $bt$) AND $z$==1)
12.         for(int $l=1$ to $dcnt$) do
13.            $copy[i] = oldlst[i]$;
      }
14.   $z$ =$\lfloor$z/2$\rfloor$;
    }
}

shared word $BLK[1..B+kN*T][1..S]$;
private INDEX $copy[1..kT]$, $oldlst[1..kT]$;
private pointer $ptrs[1..B]$;
private boolean $dirty[1..B]$;
private INDEX $dcnt$, $blkidx$;
private word $v$;

wordtype read(int $addr$){
15.   $v$=$BLK[ptrs[addrdivS]][addrmodS]$;
16.   if($VL(BANK)$==$F$)
17.      goto line 27 of calculate (Algorithm 2);
18.   else return $v$;
}

void write(int $addr$, wordtype $val$){
19.  $blkidx$=$addr$ div $S$;
20.  if($dirty[blkidx]$==$F$){;
21.     memcpy($BLK[copy[dcnt]]$, $BLK[ptrs[blkidx]]$, sizeof($blktype$));
22.     $dirty[blkidx]$=$T$;
23.     $oldlsl[dcnt]$=$ptrs[blkidx]$;
24.     $ptrs[blkidx]$=$copy[dcnt]$;
25.     $dcnt$=$dcnt+1$;
26.  }
27.  $BLK[ptrs[blkidx]][addrmodS]$=$val$;
}
```

cesses. The resulting algorithm (BLS-RedBlue) is much simpler than the wait-free construction of [7] since it does not require the complicated mechanisms presenting in [7] for returning values and verifying the application of an operation. These tasks are performed in BLS-RedBlue in the same way as in S-RedBlue.

The BLS-RedBlue algorithm is presented in Algorithm 4. Procedure `propagate` executes the same code as S-RedBlue for all nodes other than the root. The code executed by some process $p$ when it reaches the blue root is presented in lines $27-36$ and it is similar to the one of LS-RedBlue. However, the execution of lines $32-36$ may have to occur more times in order to ensure that $p$'s operation has been applied to the object. Only when this has occurred, $p$'s `propagate` returns. To speed up this process, we store one more field in the blue root node, named *help*. Each process, applying a successful $SC$ on the root node, writes there the index of the last active process it has helped, and next time processes start their helping effort from the next to this process. This has as a result, the body of the `while` loop of line 31 to execute at most $\min\{k, 2M/T\}$ times. Each time that the loop is executed twice, $M/T$ more active processes are helped. Therefore, after $2k/(\min\{k, M/T\})$ iterations, the operation of $p$ will have been applied to the object.

Each iteration of the loop requires $O(B)$ time to execute lines $27-28$, 32, 36 and 34. Each execution of `calculate` applies at most $\min\{k, M/T\}$ operations. The cost of applying these operations is $O(MS + \min\{k, M/T\}D)$. Finally, the cost of calculating the return values at each execution of `calculate` is $O(k)$. So, the cost of executing the while loop is $O(k/(\min\{k, M/T\})(B+MS+k+\min\{k, M/T\}D))$. Given that each process requires only $O(\log k)$ steps to reach the root node and it holds that $\log k \in O((k/\min\{k, M/T\})(B+MS+k+\min\{k, M/T\}D))$, it follows that the time complexity of BLS-RedBlue is $O((k/\min\{k, M/T\})(B + MS + k + \min\{k, M/T\}D))$.

Obviously, BLS-RedBlue achieves better time complexity than the wait-free construction of [7] and it is adaptive. This is achieved without any increase to the required space overhead which is

$O(n^2 + n(MS + B))$ for both algorithms.

In case each return value has size larger than a single word, i.e., it is at most $R$ words, our algorithms can still work with single-word registers by substituting the array of single-writer registers held by each process with a bidimensional array of $nR$ words. Then, the time complexity of BLS-RedBlue becomes $O((k/\min\{k, M/T\})(B + MS + kR + \min\{k, M/T\}D))$. This assumption is done at the wait-free universal construction in [7] which then has time complexity $O(n/min\{n, M/T\}(B + nR + nD + MS))$ (still worse than BLS-RedBlue in this case).

If $n$ is very large, a technique like the one used by GroupUpdate [3] can be employed to store a single pointer instead of the bit vector in each blue node. Then, the time complexity of BLS-RedBlue becomes $O(k \log k + (k/\min\{k, M/T\})(B + MS + kR + \min\{k, M/T\}D))$. Given that we focus on large objects, we expect that $k \log k \in O((k/\min\{k, M/T\})(B + MS + kR + \min\{k, M/T\}D))$ in most of the cases.

## References

[1] Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proc. of the 18th ACM Symposium on Principles of Distributed Computing*, pages 91–103, 1990.

[2] Y. Afek, P. Boxer, and D. Touitou. Bounds on the shared memory requirements for long-lived & adaptive objects. In *Proc. of the 19th ACM Symposium on Principles of Distributed Computing*, pages 81–89, 2000.

[3] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proc. of the 27th ACM Symposium on Theory of Computing*, pages 538–547, 1995.

[4] Y. Afek, M. Merritt, and G. Taubenfeld. The power of multi-objects. *Information and Computation*, 153:213–222, 1999.

[5] Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou. Disentangling multi-object operations. In *Proc. of the 16th ACM Symposium on Principles of Distributed Computing*, pages 262–272, 1997.

[6] J. H. Anderson and M. Moir. Universal constructions for multi-object operations. In *Proc. of the 14th ACM Symposium on Principles of Distributed Computing*, pages 184–193, 1995.

[7] J. H. Anderson and M. Moir. Universal constructions for large objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317–1332, Dec. 1999.

[8] H. Attiya and A. Fouren. Adaptive and efficient wait-free algorithms for lattice agreement and renaming. *SIAM Journal on Computing*, 31(2):642–664, 2001.

[9] H. Attiya and A. Fouren. Algorithms adapting to point contention. *Journal of the ACM (JACM)*, 50:444–468, July 2003.

[10] 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.

**Algorithm 4** Algorithm BLS-RedBlue.

```
struct bnode{
    INDEX $BANK[B]$;             // used only at the root node
    PINDEX $pid$;                // used only at the root node
    PINDEX $help$;               // used only at the root node
    boolean $ops[n]$;
}

void propagate(int $z$, int $p$){
    bnode $b$;
1.  while($z$!=1){
2.      for (int $i = 1$ to 2) do {
3.          $b$=$LL$($bn[z]$);
4.          $b$t=calculate($z$, $p$);
5.          SC($bn[z]$, $bt$);
        }
6.      z =$\lfloor$z/2$\rfloor$;
    }

     // operations to be performed at the root node
7.   $b$=$LL$($bn[1]$);
8.   while ($b.ops[p]$ == $F$) {
9.       for (int $j$=1 to $B$) do
10.          $dirty[j]$=$F$;
11.      $dcnt = 0$;
12.      $b$=$LL$($bn[1]$);
13.      $ptrs = b.BANK$;
14.      $bt$=calculate(1, $p$);
15.      if (SC($bn[1]$, $bt$))
16.          for ($l = 1$ to $dcnt$) do $copy[i] = oldlst[i]$;
     }
}

struct bnode calculate(int $z$, int $p$) {
    struct bnode $tmp =< \perp, -1, < F, \ldots, F >>$, $blue$ = $bn[z]$, $lc$, $rc$;
    struct rnode $red$ = $rn[z]$;
    int $help$=0,$q$;

17. if ($2 * z + 1 < 2n$) { // check if $z$ is an internal node
18.         $lc$=$bn[2 * z]$;
19.         $rc$=$bn[2 * z + 1]$;
    }

20. if ($z$==1) {$q$ = $blue.help$; $tmp.pid$ = $p$; }
21. else $q$ = 1;
22. for (int $i = 1$ to $n$) do{
23.     if ($red.pid$ == $q$) // check if process $q$ occupies node red
24.         $tmp.ops[q] = red.op$;
25.     else if (is_predecessor($z$,$q$,$2 * z$))
26.         $tmp.ops[q] = lc.ops[q]$;
27.     else if (is_predecessor($z$,$q$,$2 * z + 1$))
28.         $tmp.ops[q] = rc.ops[q]$;
29.     if ($z == 1$ AND $tmp.ops[q]$==$T$ AND $blue.ops[q]$==$F$) {
30.         if ($help < M/T$) {
31.              apply $ops[q]$ and store into $rvals[p][q]$ the return value;
32.              $help$ = $help + 1$;
33.         }
34.          else $tmp.ops[q]$ = $F$;
35.     }
36.     else if($z == 1$ AND $tmp.ops[q]$==$T$) $rvals[p][q] = rvals[b.pid][q]$;
37.      $q = (q + 1)$ MOD $n$
    }
38. return $tmp$;
}
```

[11] T. D. Chandra, P. Jayanti, and K. Tan. A polylog time wait-free construction for closed objects. In *Proc. of the 17th ACM Symposium on Principles of Distributed Computing*, pages 287–296, 1998.

[12] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13:124–149, Jan. 1991.

[13] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, Nov. 1993.

[14] M. Herlihy, V. Luchangco, and M. Moir. Space and time adaptive non-blocking algorithms. *Electronic Notes in Theoretical Computer Science*, 78, 2003.

[15] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12:463–492, 1990.

[16] P. Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *Proc. of the 17th ACM Symposium on Principles of Distributed Computing*, pages 201–210, 1998.

[17] P. Jayanti. f-arrays: implementation and applications. In *Proc. of the 21th ACM Symposium on Principles of Distributed Computing*, pages 270–279, 2002.

[18] P. Jayanti and S. Petrovic. Efficient wait-free implementation of multiword ll/sc variables. In *Proc. of the 25th IEEE International Conference on Distributed Computing Systems*, pages 59–68, 2005.