Making Concurrent Data Structures Recoverable

July 19, 2018

Abstract

Recent developments foreshadow the emergence of new systems, in which byte-addressable *non-volatile main memory* (*NVRAM*), combining the performance benefits of conventional main memory with the durability of secondary storage, co-exists with (or eventually even replaces) traditional volatile memory. Consequently, there is increased interest in *recoverable* concurrent objects: objects that are made robust to crash-failures by allowing their operations to recover from such failures. This paper presents a principled approach to deriving recoverable versions of several widely-used concurrent data structures, in particular, a linked list and an elimination stack.

1 Introduction (based on our PODC)

Shared-memory multiprocessors are asynchronous in nature. Asynchrony is related to reliability, since algorithms that provide nonblocking progress properties (e.g., lock-freedom and wait-freedom [8]) in an asynchronous environment with reliable processes continue to provide the same progress properties in the presence of *crash failures*. This happens because a process that crashes permanently during the execution of the algorithm is indistinguishable to the other processes from one that is merely very slow. Owing to its simplicity and intimate relationship with asynchrony, the crash-failure model is almost ubiquitous in the treatment of concurrent algorithms.

The attention to the crash-failure model has so far mostly neglected the *crash-recovery* model, in which a failed process may be resurrected after it crashes. Recent developments foreshadow the emergence of new systems, in which byte-addressable *non-volatile main memory* (*NVRAM*), combining the performance benefits of conventional main memory with the durability of secondary storage, co-exists with (or eventually even replaces) traditional volatile memory. Consequently, there is increased interest in *recoverable concurrent objects* (also called *persistent* [4, 5] or *durable* [11]): objects that are made robust to crash-failures by allowing their operations to recover from such failures.

This paper consider an abstract individual-process crash-recovery model for non-volatile memory [2]. Processes communicate via non-volatile shared-memory variables. Each process also has local variables stored in volatile processor registers. [[YF: What about program counter? HA: in this paper, it can be volatile.]] At any point, a process may incur a crash-failure, causing all its local variables to be reset to arbitrary values. Operation response values are returned via volatile processor registers, which may become inaccessible to the calling process if it fails just before persisting the response value. A data structure has an associated *recovery function* that is responsible for restoring it upon the recovery from a crash-failure. The recovery function completes the current outstanding operation on the data structure, if there was any, returning either its *response* or a *fail* indication, if it was unsuccessful. Both responses are consistent with the resulting state of the data structure, to which the operation was applied (in the former case) or not (in the latter case).

We present a principled approach to deriving recoverable versions and describe in in detail in two widely-used concurrent data structures: a linked list [] and elimination stack [].

Several correctness conditions for the crash-recovery model were defined in recent years (see, e.g., [1, 3, 6, 7, 10]). The goal of these conditions is to maintain the state of concurrent objects consistent in the face of crash failures.

2 Model and Definitions (based on our PODC)

We consider a system where N asynchronous *processes* $p_1, \ldots, p_N$ communicate by accessing *concurrent objects*. The system provides *base objects* that support atomic read, write, and compare&swap (CAS). Base objects can be used for implementing more complex concurrent objects (e.g. lists, trees and stacks), by defining access procedures that simulate each operation on the implemented object using operations on base objects.

The state of each process consists of non-volatile *shared-memory variables*, as well as *local variables stored in volatile processor registers*. Each process can incur at any point during the execution a *crash-failure* (or simply a *crash*) that resets all its local variables to arbitrary values, but preserves the values of all its non-volatile variables. A process *p invokes an operation Op* on an object by performing an *invocation step*; upon Op's completion, a *response step* is executed.

Operation Op is *pending* if it was invoked but was not yet completed. For simplicity, we assume that, at all times, each process has at most a single pending operation on any one object.

[[HA: Define data structure D.] Each data structure has an associated *recovery function*, denoted D.`Recover`, which is responsible for restoring the data structure to a consistent state, upon recovery from a crash.

More formally, a *history* H is a sequence of *steps*. There are four types of steps:

1. an *invocation step*, denoted (INV, p, O, Op), represents the invocation by process p of operation Op on object O;

2. an operation Op can be completed either normally or when, following one or more crashes, the execution of Op.`Recover` is completed. In either case, a *response step* s, denoted (RES, p, O, Op, ret), represents the completion by process p of operation Op invoked on object O by some step s' of p, with response ret being written to a local variable of p. We say that *s is the response step that matches s'*;

3. a *crash step* s, denoted $(CRASH, p)$, represents the crash of process p. We call the inner-most recoverable operation Op of p that was pending when the crash occurred the *crashed operation of s*. $(CRASH, p)$ may also occur while p is executing some recovery function Op.`Recover` and we say that Op is the crashed operation of s also in this case;

4. a *recovery step s for process p*, denoted (REC, p), is the only step by p that is allowed to follow a $(CRASH, p)$ step s'. It represents the resurrection of p by the system, in which it invokes Op.`Recover`,[1] where Op is the crashed operation of s'. We say that *s is the recovery step that matches s'*.

For a history H, we let $H|p$ denote the subhistory of H consisting of all the steps by process p in H. H is *crash-free* if it contains no crash steps (hence also no recover steps). We let $H|<p, O>$ denote the subhistory consisting of all the steps on O by p.

Given two operations op_1 and op_2 in a history H, we say that op_1 *happens before* op_2, denoted $op_1 <_H op_2$, if op_1's response step precedes the invocation step of op_2 in H. $H|O$ is a *sequential object history*, if it is an alternating series of invocations and the matching responses starting with an invocation (the history may end by a pending invocation). The *sequential specification* of an object O is the set of all possible (legal) sequential histories over O. A history H is *sequential* if $H|O$ is a sequential object history for all objects O.

Two histories H and H' are *equivalent*, if $H|<p, O> = H'|<p, O>$ for all processes p and objects O. Given a history H, a *completion* of H is a history H' constructed from H by selecting separately, for each object O that appears in H, a subset of the operations pending on O in H and appending matching responses to all these operations, and then removing all remaining pending operations on O (if any).

Definition 1 (Linearizability [9], rephrased) *A finite crash-free history H is* linearizable *if it has a completion H' and a legal sequential history S such that:*

L1. H' is equivalent to S; and

[1] A history does not contain invocation/response steps for recovery functions.

L2. $<_H \subseteq <_S$ *(i.e., if* $op_1 <_H op_2$ *and both ops appear in* S *then* $op_1 <_S op_2$*).*

Thus, a finite history is linearizable, if we can linearize the subhistory of each object that appears in it. Next, we define a more general notion of well-formedness that applies also to histories that contain crash/recovery steps. For a history H, we let $N(H)$ denote the history obtained from H by removing all crash and recovery steps.

Definition 2 (Recoverable Well-Formedness) *A history* H *is recoverable well-formed if the following holds.*

1. *Every crash step in* $H|p$ *is either* p*'s last step in* H *or is followed in* $H|p$ *by a matching recover step of* p*.*
2. $N(H)$ *is well-formed.*

We can now define the notion of nesting-safe recoverable linearizability.

Definition 3 (Nesting-safe Recoverable Linearizability (NRL)) *A finite history* H *satisfies* nesting-safe recoverable linearizability *(NRL) if it is recoverable well-formed and* $N(H)$ *is a linearizable history. An object implementation satisfies NRL if all of its finite histories satisfy NRL.*

2.1 General Overview

Designing a data structure which is persistent in the presence of a crash-recovery is not an easy task. Several different correctness conditions and implementations have been proposed for such data structures [cite...]. However, most of these conditions and implementations does not support detectability, that is, the ability to conclude upon recovery whether the crashed operation took effect or not.

Attiya et al [cite] presented a novel crash-recovery model together with a correctness condition. Roughly speaking, Nesting-safe Recoverable Linearizability (NRL) requires each recoverable operation to supply an recover function, such that invoking it after a crash inside the operation allows the process to complete the operation, as well as restore the response value if needed. Moreover, they presented recoverable implementations for read,write and CAS operations. As suggested by the name NRL, it allows nesting. Therefore, taking any algorithm which uses only read,write and CAS primitives, and replacing each primitive with its recoverable version yields an NRL implementation (some minor changes are still needed in order to use this transformation). In particular, this is the case for all the algorithms presented in this paper. However, this transformation is very costly, in terms of both time and space.

Cohen et al [cite..] presented a universal construction that implements durably any object in a lock-free manner, using at most one persistent fence per operation, which is optimal. Moreover, the algorithm uses only read, write and CAS operations. However, universal construction by its nature is not efficient. In particular, their implementation requires to keep the entire history of the object in a designated shared queue, as well as per-process persistent log, such that these logs as whole keeps the entire history, and different logs may have a big overlap. In addition, the response of an operation is determine by looking at the entire history up to the linearization point of the operation. Their universal construction can be made detectable in the following way: upon recovery from a crash inside an $Update(op)$ operation, and after the system completes its recovery routine, *op* was linearized if and only if *op* appears in the shared queue (assuming each *op* structure has a unique address). In such case, the process can conclude the response value, since the queue represents the object's history.

Clearly, although the construction has an optimal fence complexity, in practice it is not efficient. Moreover, the crash model they consider is a system-wide crash, in which all processes crash at once. In such case, upon recovery, a single recover function is being executed by the system in order to reconstruct the queue data-structure in a consistent way. The correctness condition they consider is durable linearizability [cite], which requires that after a full system crash, the state of the object must reect a consistent operation sub-history that includes all operations completed by the time of the crash. Therefore, some of the crashed operation may get lost.

Different concurrent implementation for specific data-structures are known. These implementations exploit the structure and requirements of the specific object in order to optimize the resulting implementation. For example, operations that effects different parts of the data structure can be done concurrently without interfering each other. A lot of work and effort has been put in order to develop these implementations, as well as proving their correctness, thus we would like to have a general approach to turn a given implementation to a recoverable one, while preserving its structure and complexity as much as possible. This way, we can enjoy both an efficient algorithm, as well as recoverability, while avoiding the need to design new algorithms.

Ben-David, Blelloch and Wei [ref] presented a transformation for implementations which uses only read, write and CAS operation, which results a recoverable and detectable implementation. The transformation splits the code into capsules, each contains a single CAS operation followed by reads, and then replacing each CAS with its recoverable version. Each capsule can be recovered in case of a crash inside the capsule. In addition, the paper presents an optimisation for normalized algorithms, such that only two capsules are required for any operation. However, not all implementation are given in a normalized form, and the process of converting an implementation into a normalized form may be costly by itself.

Since Ben-David at el transformation is general, it may be improved in certain cases. In more details, each CAS is replaced with a recoverable CAS. This requires each CAS operation to use different arguments (they implement it using an increasing sequence number), which will be stored in the CAS location. This in turn implies that the CAS uses an unbounded word length, even if the original implementation uses a bounded CAS (for example, the operations are over a finite domain). Furthermore, altohugh two capsules are used for each normalized operation, these two capsules are executed repeatedly, in each attempt of performing the operation (the implementation is lock-free, hence a process can fail to complete its operation over and over). In many cases we can avoid it, and not paying the extra complexity of the capsules for a failed attempt. In addition, we would like the transformation to change as little as necessary from the original code, even in the price of having a costly recover function. Assuming crash is a rare event, this may result a more efficient implementation in practice.

Harris' Linked-List presented in details in Section [cite]. The algorithm has the following property, which makes it persistent to crashes - each operation is linearized at its first successful CAS (not including CAS which help other operation). However, as discussed before, this does not imply it is also detectable. The main issue is a crash right after such a CAS. Upon recovery the crashed process needs to know whether the last CAS was successful or not, in order to conclude the right response value. Moreover, two or more processes can try and perform the same operation at the same time, for example, deleting the same node. Therefore, upon recovery p not only needs to know whether the node was deleted, but also to determine if it was performed by it, or by some other process. In order to solve these issues we use the structure of the Linked-List implementation.

First, we notice that, as in many algorithms, data is added to the Linked-List using nodes, and each node is unique (different nodes may have the same key, but must have different references). Therefore, if a process adds a new node nd to the list using a CAS, then from this point on nd is accessible as part of the list, until it is deleted. This implies a recovery function for the Insert operation, by testing the former two conditions p can conclude whether its last attempt to add nd was successful. For the Delete operation, we notice that once a node is logically delete (the marked bit is turned on), it remains so. Hence, if process p tries to delete node nd and crash, upon recovery p can know if the node was logically deleted by looking at the marked bit. However, this is not enough information, as there might be many processes trying to delete nd at the same time, and exactly one needs to succeed. Therefore, processes needs to reach a consensus regarding which process is the one to delete nd. For that, each node is equipped with an extra field named deleter which is initialised to $\perp$. This field is used by processes trying to delete the same node nd in order to reach a consensus regarding which process is the one to delete nd. After nd is logically delete, the first process which writes to deleter its id (using an atomic CAS) is the one to declare its Delete operation as successful, while all others return false. Notice that crashed operation are also trying to win the consensus, and that deleter is written to exactly once. If a crash occurs after deleter was written to, then a process can recover by reading deleter and compare it to its own id.

The same technique can be used for other implementations which share some similarities with the Linked-List implementation. For example, an operation which takes effect using a single CAS, and needs to guarantee exactly one process is the one to perform it, can use a new field as the deleter in order to determine which process executed it. This requires the ability to conclude whether some process successfully performed the CAS upon recovery. In addition, the Insert operation of data-structures in which a new and unique structure is used each time a new key is added (as a node), can be recovered by looking for the new structure, or an indication for it being removed from the data-structure. Such an implementation is Michel-Scott queue for example.

3 Linked-List

In this section, we present a recoverable version of the linked-list set algorithm of Harris [**?**].[2] Harris' algorithm maintains a linked list of nodes sorted in increasing order of keys. The *next* field of each node consists of two components: a reference to the next node and a *marked* bit that is set when the node is logically deleted. Both componenets can be manipulated atomically, either together or individually, using a singe-word CAS operation.[3] For presentation simplicity, we assume in the pseudo-code that reading the *next* field returns the reference only, while invoking the get_data() function on *next* returns both the reference and the mark bit. We also assume that the list always contain a *head* and *tail* sentinel nodes, containing keys $-\infty, \infty$ respectively.

We now provide a brief description of the Harris algorithm. The linked-list set supports the Find, Insert and Delete set operations. Insert and Delete use the Search helper procedure in order to find the node with the smallest key greater than or equal to the input key (this node

[2] Some implementation details follow the algorithm's presentation by [**?**].

[3] In the Java implementation of the algorithm in [**?**], *next* fields are represented by AtomicMarkableReference objects, in whose implementation the marked bit uses the least significant bit of the reference.

is denoted *curr*), as well as its predecessor on the list (denoted *pred*). While traversing the list, Search attempts to physically remove from the list any marked node it encounters. To insert a key α, a process p first calls Search in order to find the position in the list where α should be added. If α is already in the list, Insert returns false, otherwise it tries to set *pred.next* to point to a new node containing α using CAS (which would fail if *pred* has been logically deleted in the interim). To delete a key α, p searches for it using Search and returns false if it not in the set. Otherwise, p tries to logically delete it by marking the *next* field if its node using CAS. If marking is successful, p tries to physically remove the node. To find a key α, a process simply looks for a node in the list with key α which is unmarked.

The linearization point for the Harris algorithm are as follows:

Insert: Either when Lookup finds α in the list, or upon a successful CAS inserting α to the list.

Delete: Upon a successful CAS that marks the node (logical deletion).

Find: Upon the last read of either *curr.key* or *curr.next*.

Following the given linearization points (omitting proof...), Insert and Delete operation are linearized at the point where they affect the system. That is, if an Insert operation performed a successful CAS, then all processes will see the new node starting from this point, and if a node was logically delete, then all processes treat it as if it was removed. Therefore, once a process p recovers following a crash, the list data structure is consistent - if p has a pending operation, either the operation already had a linearization point which affected all other processes, or it did not affect the data structure at all, nor will in any future run.

However, even though the list data structure is consistent, the response of the pending operation is lost. Consider for example a scenario in which process p performs Delete(α) and crash right after applying a successful CAS to mark a node. Upon recovery, p may be able to conclude α was removed, as the node is marked. Nevertheless, even if no other process takes steps, p is not able to determine whether it is the process to successfully delete α, or that it was done by some other process, and therefore it does not able to determine the right response. Moreover, in case the node was physically removed, p is not able to determine whether α has been deleted at all, as it is no longer part of the list.

3.0.1 Linked-List Recoverable Version

To solve the problems mention above, we present a modification for the algorithm such that in case of a process crash, upon recovery it is able to complete its last pending operation if needed, and also return the response value in such case. The algorithm presented in figure 1. Blue lines represents changes comparing to the original algorithm.

Each node is equipped with a new field named deleter. This field is used to determine which process is the one to delete the node. After the node was successfully marked (logical delete), process p tries to announce itself as the one to delete the node by writing its id to deleter using CAS. This way, if a process crash during a delete, it can use deleter in order to determine the response value. We assume deleter is initialised to null when creating a new node.

Each process p has a designated location in the memory, Backup[p]. Before trying to apply an operation, p writes to Backup[p] the entire data needed to complete the operation. Upon recovery, p can read Backup[p], and based on it to complete its pending operation, in case there is such. Formally, Backup[p] contains a pointer to a structure containing all the relevant data. For simplicity, process p creates new such structure for each of its operations, although a more efficient way will be to use two such structures in an alternating way.

Algorithm 5: boolean RECOVER ()

```
Data:   Node *nd := Backup[pid].nd
if Backup[pid].result ≠ ⊥ then                          // operation was completed
    return Backup[pid].result
if Backup[pid].optype = INSERT then
    ⟨pred, curr⟩ := SEARCH(nd.key)                      // search for nd in the list
    if curr = nd || nd.next is marked then              // nd is in the list or marked
        Backup[pid] := true
        return true
    return FAIL
if Backup[pid].optype = DELETE then
    if nd ≠ ⊥ && nd.next is marked then                 // nd was logically deleted
        nd.deleter.CAS(⊥, pid)                          // try to complete the deletation
        if nd.deleter = pid then                        // you are the deleter
            Backup[pid].result := true
            return true
    return FAIL
```

Algorithm 1: ⟨Node, Node⟩ SEARCH (T *key*)

```
Data:   Node *pred, *curr, *succ
        boolean mbit
retry: while true do
    pred := head
    curr := pred.next
    while true do
        ⟨succ, mbit⟩ := curr.next.get_data()
        if mbit then                                    // succ was logically deleted
            if pred.next.CAS(unmarked curr,unmarked succ) = false then   // help physical delete
                go to retry                             // help failed
            curr := succ                                // help succeed
        else
            if curr.key ≥ key then                      // curr is the first unmarked node with key ≥ key
                return ⟨pred, curr⟩
            pred := curr                                // advance pred and curr
            curr := succ
    end
end
```

Shared variables: Node *∗head*

Type Info {
  {Insert, Delete} *optype*
  Node *∗nd*
  boolean *result*
}

Code for process p:

Algorithm 2: boolean Insert (T *key*)

```
Data:   Node *pred, *curr
        Node newnd := new Node (key)
Backup[pid] := new Info (INSERT, newnd, ⊥)
while true do
    ⟨pred, curr⟩ := SEARCH(key)          // search for the right location for insertion
    if curr.key = key then               // key is already in the list
        Backup[pid].result := false
        return false
    else
        newnd.next := unmarked curr
        if pred.next.CAS (unmarked curr, unmarked newnd) then   // try to add newnd
            Backup[pid].result := true
            return true
end
```

Algorithm 3: boolean Delete (T *key*)

```
Data:   Node *pred, *curr, *succ
Backup[pid] := new Info (DELETE, ⊥, ⊥)
⟨pred, curr⟩ := SEARCH(key)              // search for key in the list
if curr.key ≠ key then                   // key is not in the list
    Backup[pid].result := false
    return false
else
    Backup[pid].nd := curr
    while curr.next is unmarked do       // repeatedly attempt logical delete
        succ := curr.next
        curr.next.CAS (unmarked succ, marked succ)
    end
    succ := curr.next
    pred.next.CAS (unmarked curr, unmarked succ)      // physical delete attempt
    res := curr.deleter.CAS(⊥, pid)                   // try to announce yourself as deleter
    Backup[pid].result := res
    return res
```

Algorithm 4: boolean Find (T *key*)

```
Data:   Node *curr := head
while curr.key < key do      // search for the first node with key greater or equal to key
    curr = curr.next
end
return (curr.key = key && curr.next is unmarked)
```

Figure 1: Recoverable Non-Blocking Linked-List

Correctness Argument

In the following, we give an high-level proof for the correctness of the algorithm.

First, notice that quitting the Lookup procedure at any point, or repeating it, can not violet the list consistency. The Lookup procedure simply traverse the list, while trying to physically delete marked nodes. Once curr.next is marked, a single process can perform the physical delete. This follows from the fact that at any point there is a single node in the list which points to curr. Once curr is physically delete, no node in the list points to curr, and thus any CAS operation with curr as the first parameter will fail. This observation relays on the fact that any new allocated node has a different address then curr. As a result, repeating the attempt to physically delete a node does not affect the list.

Assume a process p performs an $insert(key)$ operation. First, p writes to Backup[p], updating it is about to perform an Insert. If a process p does not crash, then, as in the original algorithm, it repeatedly tries to find the right location for the new node, and insert it by performing a CAS changing *pred.next* to point to newnd. In addition, it is clear from the code that a crash after updating $Backup[p].result$ is after the operation had its linearization point, and the Recover procedure will return the right response. Therefore, we need to consider a crash before an update to $Backup[p].result$. There are two scenarios to consider.

Assume p crash without performing a successful CAS in line 27. p is the only process to have a reference to newnd, and it is yet to update any node with this reference, and thus no node points to newnd. As a result, the operation did not affects any other process, nor it will be in the future. Hence, considering the operation as not having a linearization point does not violate the list consistency. Indeed, since no node points to newnd, upon recovery p will see that newnd is not in the list and also not marked, and thus will return FAIL. Notice this argument holds whether key is already in the tree, or not, as the operation in both cases did not affect the system.

Assume now p crash after performing a successful CAS in line 27. In such case, newnd is part of the list, as pred.next points to it. Also notice we did not delete any other node, since pred.next pointed to curr, and after the CAS it points to newnd which points to curr. As a result, when p executes the Recover procedure, either it will see newnd in the list, or that it is no longer part of the list, and it must be some other process deleted it, and hence newnd.next is marked. In any case, p will return true as required. The above argument relies on the fact a marked node can not be unmarked, and that an Insert and Delete can not mistakenly remove nodes from the list. We have claimed it for Insert, and we will prove the same holds for Delete. Therefore, if a node is no longer in the list, it must be marked.

Assume a process p performs a $delete(key)$ operation. First, p writes to Backup[p], updating it is about to perform a Delete. As before, a crash after writing to Backup[p].result will return the right response. Also, a crush before updating any of Backup[p].result or Backup[p].nd implies p is yet to try and mark any node, and thus the operation did not affect the system so far, nor it will be in the future. Therefore, we can consider the operation as not having a linearization point (even in case key is not the list), and indeed, the Recover procedure returns FAIL in such case.

Assume thus p writes to Backup[p].nd. It follows that p completed the lookup procedure and finds a node curr storing key. The lookup procedure guarantees there is a point in time (of the procedure execution) where curr is in the list and curr.next is not marked. If p crash and recovers, and observe that curr is unmarked, then in returns FAIL. Since a marked node can not be unmarked, as there is no CAS changing a marked node, it follows that p did not marked curr. Therefore, the operation did not affects any process, nor it will be, and we consider it as having no linearization point. Otherwise, the Recover function observe curr as marked, and we can conclude the marking point of curr is along the delete operation. We now prove we can linearize the operation, according to its response.

Let q be the process to mark curr. Since once curr.next is marked it will never be changed, the reference of curr.next is fixed to succ (of q at the point of the marking). This also implies q is unique and well defined, and any future CAS on curr.next will fail. As a result, any process leaving the while loop in line 39 reads the same value in line 43, which is this succ. The attempt to physically delete curr in line 44 will succeed only if pred.next points to curr, and as we said, curr point to succ, and any other attempt will fail. Thus, if this attempt succeed, it deletes only curr, and can not delete additional nodes.

In line 45 process p tries to writes its id to curr.deleter. As it is initialised to null, only the first process to perform this CAS will succeed. Also, any p must go through line 45 in order to complete its operation, as the Recover procedure redirect the process to this line. Therefore, if there is a process to complete its delete operation while observing curr.next is marked, there must be a CAS to curr.deleter. Let q' be the first process to perform this CAS. As proved above, q' tries to delete curr, and the point in time where curr is marked must be contained in its operation interval. Moreover, q' is the only process to write to curr.deleter, and the first one to do so, thus q' is the only process to obtain true when testing (curr.deleter $= q'$) in line 46 (and thus to also return true), while any other process will obtain false. We linearize the operation of q' at the point of the marking, and any other attempt to delete curr is linearized after it (in an arbitrary order).

A corollary of the analysis is that processes trying to delete the same node curr "helps" each other, in the sense that they all keep trying to mark curr. However, the marking process is not necessarily the one to return true. Also, in the original algorithm, if a process fails to mark a node, it starts the delete operation from the beginning. In our implementation, process can keep trying to mark the node without the need to perform a lookup again after each failed CAS. We guarantee that once curr is marked, exactly one process will return true, while the rest can consider curr as being deleted (in the course of their delete execution), and thus there is a point along their execution is which key is not in the tree, and they can return false.

4 Robust BST

The original BST algorithm does not support the crash-recovery model. It is clear from the code a process does not persist the operation's response in the non-volatile memory, and thus, once a process crash the response is lost. For example, assume a process q apply INSERT(k), performs a successful CAS in line 101 and fails after completing the HELPINSERT routine. In this case, the INSERT operation took effect, that is, the new key appears as a leaf in the tree, and any FIND(k) operation will return it. However, even though the operation must be linearized before the crash, upon recovery process q is unaware of it. Moreover, looking for the new leaf in the tree may be futile, as it might be k has been removed from the tree after the crash.

Furthermore, if no recover routine is supplied, it may result an execution which is not well-formed. Consider for example the following scenario. A process q invoke an Op_1 = INSERT(k_1) operation. q performs a successful CAS in line 101 followed by a crush. After recovering, q invoke an Op_2 = INSERT(k_2) operation. Assume k_1 and k_2 belongs to a different parts of the tree (do not share parent or grandparent). Then, q can complete the insertion of k_2 without having any affect on k_1. Now, a process q' performs FIND(k_1) which returns NULL, as the insertion of k_1 is not completed, followed by FIND(k_2), which returns the leaf of k_2. The INSERT(k_1) operation will be completed later by any INSERT or DELETE operation which needs to make changes to the flagged node. We get that Op_2 must be before Op_1 in the linearization, although Op_1 invoked first.

The kind of anomaly described above can be addressed by having the first CAS of a successful attempt for INSERT or DELETE as the linearization point, as in the Linked-List. For that, the FIND routine should take into consideration future unavoidable changes, for example, a node flagged with IFlag ensures an insertion of some key. A simple solution is to change the FIND routine such that it also helps other operations, as described in figure 2. The FIND routine will search for key k in the tree. If the SEARCH routine returns a grandparent or a parent that is flagged, then it might be that an insert or delete of k is currently in progress, thus we first help the operation to complete, and then search for k again. Otherwise, if *gpupdate* or *pupdate* has been changed since the last read, it means some change already took affect, and there is a need to search for k again. If none of the above holds, there is a point in time where gp points to p which points to l, and there is no attempt to change this part of the tree. As a result, if k is in the tree at this point, it must be in l, and the find can return safely.

The approach described above is not efficient in terms of time. We would like a solution which maintain the desirable behaviour of the original FIND routine, where a single SEARCH is needed. A more refined solution is given in figure 3. The intuition for it is drown from the Linked-List algorithm. In the Linked-List algorithm it was enough to consider a marked node as if it has been deleted, without the need to complete the deletion. Nonetheless, the complex BST implementation is more challenging, as the DELETE routine needs to successfully capture two nodes using CAS in order to complete the deletion. Therefore, if a process p executes FIND(k) procedure, and observes a node flagged with DFlag attempting to delete the key k, it can not know whether in the future this delete attempt will succeed or fail, and thus does not know whether to consider the key k as part of the tree or not. To overcome this problem, in such case the process will first try and validate the delete operation by marking the relevant node. According to whether the marking attempt was successful, the process can conclude if the delete operation is successful or not. In order to easily implement the modified FIND routine there is a need to conclude from IInfo what is the new leaf (leaf *new* in the INSERT routine). For simplicity of presentation, we do not add this field, and abstractly refer to it in the code.

The correctness of the two suggested solutions relies on the following argument. Once a process flags a node during operation *Op* with input key k (either INSERT or DELETE), then if this attempt to complete the operation eventually succeed (i.e., the marking is also successful in the case of DELETE), then any FIND(k) operation invoked from this point consider *Op* as if it is completed.

The suggested modification, although being simple and local, only guarantee the implementation satisfy R-linearzability. However, the problem of response being lost in case of a crash is not addressed. Roughly speaking, the critical points in the code for recovery are the CAS primitives, as a crash right after applying CAS operation results the lost of the response, and in order to complete the operation the process needs to know the result of the CAS. In addition, because of the helping mechanism, a suspended DELETE operation which flagged a node and yet to mark one, may be completed by other process in the future, and may not. Upon recovery, the process needs to distinguish between the two cases, in order to obtain the right response.

To address this issue, we expend the helping mechanism so that it also update the info structure in case of a success. This is done by adding a boolean field, done, to the Flag structure. This way, if a process crash along an operation *Op*, upon recovery it can check to see if the operation was already completed. A crucial point is to update the *done* bit before performing the unflagging. Therefore, if a node is no longer flagged we can be sure done was already updated. If we switch the order, then it might be an operation and unflagging were completed, but the done bit is yet to be updated. Therefore, other processes can change the BST structure. However, if the process crash and recover at this point, the done bit is off, and the BST structure has been changed, so it will be harder for the process to conclude whether the operation took affect.

Before a process q attempt to perform an operation, as it creates the Flag structure *op* describing the operation and its affect on the data structure, the process stores *op* in a designated location in the shared memory (for simplicity, we use an array). As a result, upon recovery q has an access to this information. Now, q can check to see if the operation is still in progress, i.e., if the relevant node (parent or grandparent) is still flagged. If so, it first tries to complete the operation. Otherwise, it implies either the operation was completed, and therefore done bit is updated, or that the attempt was unsuccessful and there wa no write to the done bit. Hence, the done bit can distinguish between the two scenarios. Notice that there is a scenario in which process q recovers and observes an operation *Op* as it in a progress, but just before it retries it, some other process complete the operation. We need to prove that even in such case, the operation will affect the data structure exactly once, and the right response is returned.

The given implementation does not recover the FIND routine, since this routine does not make any changes to the BST, hence it is always safe to consider it as having no linearization point and reissue it. Also, for ease of presentation, we only write to $Announce[id]$ once we are about to capture a node using a CAS. However, writing to $Announce[id]$ at the beginning of the routine may be helpful in case of a crash early in the routine, so that the process will be able to use the data stored in $Announce[id]$ in such case also. The same is true with response value, $Announce[id] \rightarrow done$ is updated only if the routine made changes to the BST.

4.0.2 Correctness

In the following section we give a proof sketch for the algorithm correctness. We assume for simplicity nodes and Flag records are always allocated new memory locations, although it is enough to require no location is reallocated as long as there is a chain of pointers leading to it. The proof relies on the correctness of the original algorithm, which can be found on [....].

The proof relies on several key arguments given below.

[**Arg1**] The original algorithm is anonymous and uniform, i.e., any number of processes can use the BST, and there is no need to know the number of processes in the system in order to use the BST. Notice that all helping routines in the given implementation are completely anonymous, and an execution of such a routine by either the process which invoked *op* or any other helping process executes the exact same code. This observation allows the use of the following argument. If a process crash while executing some helping routine, we can consider it as an helping process which stop taking steps (more formally, there is an equivalent execution in which there is such a process, and it is indistinguishable to all process in the system). Since such process can not cause a wrong behaviour of the algorithm, so does the crash. A corollary of this argument is that repeating an helping routine multiple times by the same process can not violate the BST specification, as there is an equivalent executions in which multiple processes executes the different helping routines.

[**Arg2**] It is easy to verify the post-conditions of the SEARCH routine still holds, as they follow directly from the routine's code, and does not rely on the structure or correctness of the BST. Also, the SEARCH routine does not make any changes to the BST, but rather simply traverse it. Therefore FIND routine, which only uses SEARCH, does not affect any process, and in case of a failure along FIND execution, reissuing it satisfies NRL.

[**Arg3**] If an internal node nd_1 stops pointing to a node nd_2 at some point of the execution, it can not point to nd_2 again. This attributes to the fact an INSERT presents a node with two new children. Therefore, if nd_2 is a leaf, it can either be delete, or replaced by a new copy of an INSERT operation. Otherwise, nd_2 is an internal node, and as such, the pointer to it by nd_1 can not be replaced by an INSERT operation (which only allows to replacement a leaf), and therefore it can only be removed from the tree.

[**Arg4**] The field update of a node *nd* can have any value only once along an execution. Any attempt to perform an operation creates a new record in the memory. If $nd \rightarrow update$ is marked, it can not be unmarked or changed. Otherwise, any attempt to flag it uses a new created record *op*. If the attempt succeed, then eventually it will be unflagged while still referring to *op*. In order to replace the value again, there must be an operation reading $nd \rightarrow update$ after it was unflagged (as any operation first help a flagged node). This operation must create a new record, and thus we can use the same argument again. As a corollary, if a process successfully flag or mark a node, there was no change to the node since the last time it read the update field of the node.

Proof Sketch Assume a process q performs an operation Op (either INSERT or DELETE). If q does not crash, the algorithm is identical to the original algorithm, except for the additional write to $Announce[q]$ and $op \rightarrow done$, and thus the correctness of the original algorithm can be applied. Otherwise, q crash at some point, and upon recovery it reads op from $Announce[q]$. This record represent the last attempt of q to complete Op. We split the proof based on the type of operation.

Op = INSERT. Consider the read of $op \rightarrow p \rightarrow update$ upon recovery, and denote this value by *pupdate*. If $pupdate = \langle \text{IFlag}, op \rangle$, this implies the iflag CAS in line 101 was successful and the operation is yet to complete. It might be that INSERT already took affect, that is, the new key is part of the tree, but the unflagging is yet to happen. In such case, q calls HELPINSERT(op) in order to try and complete the operation. Considering arg1, this call can not violate the BST correctness, even if it not the first time q executes it. Moreover, during HELPINSERT there is a write to $op \rightarrow done$, and thus after completing the routine q returns TRUE, as required.

Else $pupdate \neq \langle \text{IFlag}, op \rangle$. There are two scenarios to consider. Either the iflag CAS of q in line 101 was successful or not. If it was successful, then $p \to update = \langle \text{IFlag}, op \rangle$ at this point. The only way to change it is to first unflag p. To do so, a process needs to complete an HELPINSERT(op) routine, and in particular must write to $op \to done$. In such case, the INSERT operation was completed, and q returns TRUE. Otherwise, the CAS was not successful, either because it failed, or the crash was before the CAS. In both cases, the INSERT operation will not be completed, as op is not stored in $p \to update$, and thus no process has an access to it. Consequently, no process can update $op \to done$, and q returns FAIL.

Op = DELETE. Consider the read of $op \to gp \to update$ upon recovery, and denote this value by *gpupdate*. If $gpupdate = \langle \text{DFlag}, op \rangle$, this implies the dflag CAS of q in line 128 was successful, and the operation is yet to complete. As in the INSERT, it might be the operation already changed the tree. After reading *gpupdate* q invokes HELPDELETE(op) routine. Again, following arg1, executing this multiple times by q can not violate the BST correctness. The first process to try and mark $op \to p \to update$ during an HELPDELETE(op) routine is the one to determine the outcome of it. If it is successful, then p is marked, and the *update* field can not be changed. That is, any HELPDELETE(op) execution will obtain true in line 139, and will call HELPMARKED(op) routine. Otherwise, the CAS fails, and so $p \to update$ is no longer equal to $op \to pupdate$. By arg4 it will never get this value again, and thus any marking CAS during a HELPDELETE(op) execution will fail, and there is no call to HELPMARKED(op). In the first case, any HELPDELETE(op) routine must first complete a HELPMARKED(op), and thus must write to $op \to done$, while in the later case, there is no write to $op \to done$, as no HELPMARKED(op) is ever invoked. Therefore, in both cases, when q completes HELPMARKED(op) it reads $op \to done$ and returns the right response.

Otherwise $gpupdate \neq \langle \text{DFlag}, op \rangle$, and there are two scenarios to consider. If the dflag CAS of q in line 128 never took affect, because it either failed, or the crash preceded it, then op is never written to $gp \to update$, or to any update field. Thus, no process is aware of it, and $op \to done$ remains FALSE, resulting q returning FAIL as required. Else, the CAS was successful, and $gp \to update$ was flagged. The only way to change it is to first unflag it, and this in turn can be done only during an HELPDELETE(op) routine. In this case, it can be unflagged in either the HELPMARKED routine in line 154, or in line 145 of the HELPDELETE routine. As mention before, the first CAS in line 138 of an HELPDELETE(op) execution determines the outcome for all HELPDELETE(op). If it is successful, $p \to update$ is forever marked, and all HELPDELETE(op) must invoke HELPMARKED(op). Therefore, the only option to unflag $gp \to update$ is at the end of HELPMARKED(op) routine, and this done only after setting $op \to done$. In such case, the DELETE operation took affect, and q will return TRUE. On the other hand, if the CAS was not successful, then any HELPDELETE(op) will fail to mark $p \to update$, and hence no HELPMARKED(op) is ever invoked. As a result, there is no write to $op \to done$. In such case, the DELETE operation did not took affect, nor will be, and indeed q will return FAIL.

```
Find(Key k) : Leaf* {
    Internal *gp, *p
    Leaf *l
    Update pupdate, gpupdate

    while (True) {
        ⟨gp, p, l, pupdate, gpupdate⟩ := Search(k)
        if gpupdate.state ≠ Clean then Help(gpupdate)
        else if pupdate.state ≠ Clean then Help(pupdate)
        else if gp → update = gpupdate and p → update = pupdate then {
            if l → key = k then return l
            else return Null
        }
    }
}
```

Figure 2: Solution 1: R-linearizable Find routine

```
Find(Key k) : Leaf* {
    Internal *gp, *p
    Leaf *l
    Update pupdate, gpupdate

    ⟨gp, p, l, pupdate, gpupdate⟩ := Search(k)
    if l → key ≠ k then {
        if (pupdate.state = IFlag and pupdate.info attempt to add key k) then
            return leaf with key k from pupdate.info
        else return Null
    }
    if (pupdate.state = Mark and pupdate.info → l → key = k) then return Null
    if (gpupdate.state = DFlag and gpupdate.info → l → key = k) then {
        op := gpupdate.info
        result := CAS(op → p → update, op → pupdate, ⟨Mark, op⟩)              ▷ mark CAS
        if (result = op → pupdate or result = ⟨Mark, op⟩) then return Null ▷ op → p is successfully marked
    }
    return l
}
```

Figure 3: Solution 2: R-linearizable Find routine

```
type Update {                  ▷ stored in one CAS word
    {Clean, DFlag, IFlag, Mark} state
    Flag *info
}
type Internal {                ▷ subtype of Node
    Key ∪ {∞_1, ∞_2} key
    Update update
    Node *left, *right
}
type Leaf {                    ▷ subtype of Node
    Key ∪ {∞_1, ∞_2} key
}
type IInfo {                   ▷ subtype of Flag
    Internal *p, *newInternal
    Leaf *l
    boolean done
}
type DInfo {                   ▷ subtype of Flag
    Internal *gp, *p
    Leaf *l
    Update pupdate
    boolean done
}
▷ Initialization:
shared Internal *Root := pointer to new Internal node
    with key field ∞_2, update field ⟨Clean, Null⟩, and
    pointers to new Leaf nodes with keys ∞_1 and
    ∞_2, respectively, as left and right fields.
```

Figure 4: Type definitions and initialization.

```
Recover() {
    Flag *op = Announce[id]

    if op of type IInfo then {
        result := op → p → update
        if result = ⟨IFlag, op⟩ then HelpInsert(op)            ▷ Finish the insertion
    }
    if op of type DInfo then {
        result := op → gp → update
        if result = ⟨DFlag, op⟩ then HelpDelete(op)            ▷ Either finish deletion or unflag
    }
    if op → done = True then return True
    else return Fail
}
```

Figure 5: Recover routine

```
Search(Key k) : ⟨Internal*, Internal*, Leaf*, Update, Update⟩ {
    ▷ Used by Insert, Delete and Find to traverse a branch of the BST; satisfies following postconditions:
    ▷ (1) l points to a Leaf node and p points to an Internal node
    ▷ (2) Either p → left has contained l (if k < p → key) or p → right has contained l (if k ≥ p → key)
    ▷ (3) p → update has contained pupdate
    ▷ (4) if l → key ≠ ∞1, then the following three statements hold:
    ▷     (4a) gp points to an Internal node
    ▷     (4b) either gp → left has contained p (if k < gp → key) or gp → right has contained p (if k ≥ gp → key)
    ▷     (4c) gp → update has contained gpupdate
    Internal *gp, *p
    Node *l := Root
    Update gpupdate, pupdate                                  ▷ Each stores a copy of an update field

    while l points to an internal node {
        gp := p                                               ▷ Remember parent of p
        p := l                                                ▷ Remember parent of l
        gpupdate := pupdate                                   ▷ Remember update field of gp
        pupdate := p → update                                 ▷ Remember update field of p
        if k < l → key then l := p → left else l := p → right ▷ Move down to appropriate child
    }
    return ⟨gp, p, l, pupdate, gpupdate⟩
}

Find(Key k) : Leaf* {
    Leaf *l

    ⟨−, −, l, −, −⟩ := Search(k)
    if l → key = k then return l
    else return Null
}

Insert(Key k) : boolean {
    Internal *p, *newInternal
    Leaf *l, *newSibling
    Leaf *new := pointer to a new Leaf node whose key field is k
    Update pupdate, result
    IInfo *op

    while True {
        ⟨−, p, l, pupdate, −⟩ := Search(k)
        if l → key = k then return False                      ▷ Cannot insert duplicate key
        if pupdate.state ≠ Clean then Help(pupdate)           ▷ Help the other operation
        else {
            newSibling := pointer to a new Leaf whose key is l → key
            newInternal := pointer to a new Internal node with key field max(k, l → key),
                update field ⟨Clean, Null⟩, and with two child fields equal to new and newSibling
                (the one with the smaller key is the left child)
            op := pointer to a new IInfo record containing ⟨p, l, newInternal, False⟩
            Announce[id] := op
            result := CAS(p → update, pupdate, ⟨IFlag, op⟩)   ▷ iflag CAS
            if result = pupdate then {                        ▷ The iflag CAS was successful
                HelpInsert(op)                                ▷ Finish the insertion
                return True
            }
            else Help(result)               ▷ The iflag CAS failed; help the operation that caused failure
        }
    }
}

HelpInsert(IInfo *op) {
    ▷ Precondition: op points to an IInfo record (i.e., it is not Null)
    CAS-Child(op → p, op → l, op → newInternal)               ▷ ichild CAS
    op → done := True                                         ▷ announce the operation completed
    CAS(op → p → update, ⟨IFlag, op⟩, ⟨Clean, op⟩)            ▷ iunflag CAS
}
```

Figure 6: Pseudocode for Search, Find and Insert.

```
DELETE(Key k) : boolean {
    Internal *gp, *p
    Leaf *l
    Update pupdate, gpupdate, result
    DInfo *op

    while TRUE {
        ⟨gp, p, l, pupdate, gpupdate⟩ := SEARCH(k)
        if l → key ≠ k then return FALSE                                      ▷ Key k is not in the tree
        if gpupdate.state ≠ CLEAN then HELP(gpupdate)
        else if pupdate.state ≠ CLEAN then HELP(pupdate)
        else {                                                               ▷ Try to flag gp
            op := pointer to a new DInfo record containing ⟨gp, p, l, pupdate, FALSE⟩
            Announce[id] := op
            result := CAS(gp → update, gpupdate, ⟨DFlag, op⟩)                  ▷ dflag CAS
            if result = gpupdate then {                                      ▷ CAS successful
                if HELPDELETE(op) then return TRUE                           ▷ Either finish deletion or unflag
            }
            else HELP(result)                  ▷ The dflag CAS failed; help the operation that caused the failure
        }
    }
}

HELPDELETE(DInfo *op) : boolean {
    ▷ Precondition: op points to a DInfo record (i.e., it is not NULL)
    Update result                                                            ▷ Stores result of mark CAS

    result := CAS(op → p → update, op → pupdate, ⟨MARK, op⟩)                   ▷ mark CAS
    if result = op → pupdate or result = ⟨MARK, op⟩ then {                     ▷ op → p is successfully marked
        HELPMARKED(op)                                                       ▷ Complete the deletion
        return TRUE                                                          ▷ Tell DELETE routine it is done
    }
    else {                                                                   ▷ The mark CAS failed
        HELP(result)                                                         ▷ Help operation that caused failure
        CAS(op → gp → update, ⟨DFlag, op⟩, ⟨CLEAN, op⟩)                         ▷ backtrack CAS
        return FALSE                                                         ▷ Tell DELETE routine to try again
    }
}

HELPMARKED(DInfo *op) {
    ▷ Precondition: op points to a DInfo record (i.e., it is not NULL)
    Node *other

    ▷ Set other to point to the sibling of the node to which op → l points
    if op → p → right = op → l then other := op → p → left else other := op → p → right
    ▷ Splice the node to which op → p points out of the tree, replacing it by other
    CAS-CHILD(op → gp, op → p, other)                                         ▷ dchild CAS
    op → done := TRUE                                                        ▷ announce the operation completed
    CAS(op → gp → update, ⟨DFlag, op⟩, ⟨CLEAN, op⟩)                             ▷ dunflag CAS
}

HELP(Update u) {                                                             ▷ General-purpose helping routine
    ▷ Precondition: u has been stored in the update field of some internal node
    if u.state = IFlag then HELPINSERT(u.info)
    else if u.state = MARK then HELPMARKED(u.info)
    else if u.state = DFlag then HELPDELETE(u.info)
}

CAS-CHILD(Internal *parent, Node *old, Node *new) {
    ▷ Precondition: parent points to an Internal node and new points to a Node (i.e., neither is NULL)
    ▷ This routine tries to change one of the child fields of the node that parent points to from old to new.
    if new → key < parent → key then
        CAS(parent → left, old, new)
    else
        CAS(parent → right, old, new)
}
```

Figure 7: Pseudocode for DELETE and some auxiliary routines.

5 Elimination Stack

For simplicity, we assume a value $\perp$, which is different from NULL and any other value the stack can store. Since NULL is used as a legit return value, representing the value of POP operation (when exchanging values using the elimination array), NULL can not be used to represent an initialization value, different then any stack value. The same holds for a Node, since a NULL node represent an empty stack, the value $\perp$ is used to distinguish between initialization value and empty stack.

For simplicity, we split the RECOVER routine into sub-routines, based on which operation (PUSH, POP, EXCHANGE) is pending, or needs to be recover. This can be concluded easily by the type of record stored in *Announce*[*pid*] (ExInfo or OpInfo), thus there is no need to explicitly know where exactly in the code the crash took place. Also, the RECOVER routine returns FAIL in case the last pending operation did not took affect (no linearization point), nor it will take in any future run. In such case, the user has the option to either re-invoke the operation, or to skip it, depends on the needs and circumstances of the specific use of the data structure.

The given implementation ignores the log of failures and successes of the exchange routine when recovering. That is, in case of a crash during an EXCHANGE, a process is able to recover the EXCHANGE routine, however, the log of successes and failures is not update, since it might be the process already updated it. In addition, in case of a FAIL response, we do not know whether the time limit (timeout) was reached, or that the process simply crashed earlier in the routine without completing it. The given implementation can be expanded to also consider the log. Nonetheless, for ease of presentation we do not handle the log in case of a crash. Assuming crash events are rare, the log still gives a roughly good approximation to the number of failures and successes, thus our approach might be useful in practice.

5.1 A Lock-Free Exchanger

An exchanger object supports the EXCHANGE procedure, which allows exactly two processes to exchange values. If process A calls the EXCHANGE with argument a, and B calls the EXCHANGE of the same object with argument b, then A's call will return value b and vice versa.

On the original algorithm [cite the book?!], processes race to win the exchanger using a **CAS** primitive. A process accessing the exchanger first reads its content, and act according to the state of it. The first process observe an EMPTY state, and tries to atomically writes its value and change the state to WAITING. In such case, it spins and wait for the second process to arrive. The second, observing the state is now WAITING, tries to write its value and change the state to BUSY. This way, it informs the first one a successful collision took place. Once the first process notice the collision, it reads the other process value and release the exchanger by setting it back to EMPTY. In order to avoid an unbounded waiting, if a second process does not show up, the call eventually timeout, and the process release the exchanger and return.

Assume a process p successfuly capture the exchanger by setting its status to WAITING, followed by a crash. Now, some other process q complete the exchange by setting the exchanger to BUSY. Upon recovery, p can conclude some exchange was completed, but it can not tell whether its value is part of the exchange, and thus it can not complete the operation. Moreover, p and q must agree, otherwise q will return p's value, and thus the operation of p must be linearized together with q operation.

In order to avoid the above problem, we take an approach resembling the BST implementation. Instead of writing a value to the exchanger, processes will use an info record, containing the relevant information for the exchange. This way, processes use the exchanger in order to exchange info records (more precisely, pointers to such records), and not values. To overcome the problematic scenario described earlier, if a process q observe the exchanger state is WAITING with some record *yourop*, it first update its own record *myop* it is about to try and collide with *yourop*, and only then performs the **CAS**. This way, if the collision is successful, the record *myop* which now stored in the exchanger implies which two records collide. Also, the fact that different processes uses different records guarantee that at most one record can collide with *yourop*.

Using records instead of values, when using wisely, allows us to farther improve the algorithm. First, there is no need to store the exchanger's state in it (by using 2 bits of it to mark the state), but we can rather have this info in the record. Second, if there is a BUSY record in the exchanger, it contains the info of the two colliding records. Therefore, a third process, trying to also use the exchanger, can help the processes to complete the collision, and then can try and set the exchanger back to EMPTY, so it can use it again. In the original implementation, a process observaing a BUSY exchanger, have to wait for the first process to read the value and release the exchanger. Therefore, if the first process crash after the collision, the exchanger will be hold by it forever. The helping mechanism avoids this scenario, making the exchange routine non-blocking.

Notice that no exchange record with EMPTY state is ever created, except for the $default$ record. Therefore, reading EMPTY state is equivalent to the exchanger storing a pointer to $default$. A process p creates a new record *myop* when accessing the exchanger, with a unique address. As long as p fails to perform a successful **CAS**, and thus fails to store *myop* in *slot*, it is allowed to try again. However, once a process performs a successful **CAS** and stores *myop* in *slot*, the only other **CAS** it is allowed to do are in order to try and store $defualt$ in *slot*. Thus, *myop* can be written exactly once to *slot*. It follows that a collision can occur between two processes exactly - once a WAITING record stored in *slot*, only a single **CAS** can replace it with a BUSY record. As the two records can not be written again to *slot*, no other process can collide with any of the records.

The EXCHANGE-RECOVER routine relies on the following argument. If a process p successfully wrote op_p to *slot* using the **CAS** in line 81, the only way to overwrite it by a different process q, is by a **CAS** in line 104 with a record op_q such that its state is BUSY, and $op_q.partner = op_p$. In addition, the only way to overwrite op_q is by a **CAS** replacing it with $default$, and this is done only after SWITCHPAIR(op_p, op_q) is completed, and thus both *result* fields are updated.

The correctness of the EXCHANGE-RECOVER routine is based on the above argument. There are few scenarios to consider. If p crash after a successful **CAS** in line 81, then op_p state is WAITING. Therefore, when reading *slot* in the EXCHANGE-RECOVER one of the following must hold. If *slot* contains op_p, then no process collide with p, and p continue to run as if the time limit has been reached. Otherwise, there was a collision. From the above argument, it must be that either op_q that collide with op_p is stored in *slot*, in this case $op_q.partner = op_p$, and p will try to complete the collision and release *slot*, or that op_q has been overwritten, and in this case the $result$ field of op_p is updated. In both cases, p returns $op_p.result$. If p crash after a successful **CAS** in line 104, then op_p state is BUSY. It follows from the argument that the only way to overwrite op_p is only after completing the collision by SWITCHPAIR. Thus, either upon recovery p reads op_p from *slot*, and in this case it tries to complete the the operation, or that $op_p.result$ was already updated. In both cases, p returns it. If non of the above holds, then op_p was not involved in any collision, because either no successful **CAS** was done by p, or p reached the time limit while no process show up, and was able to set *slot* back to $defualt$. In any case, after the crash of p, op_p will never be written again to *slot*, nor any other op_q such that $op_q.partner = op_p$, as any such op_q tries to perform

CAS (op_p, op_q) that will fail. Also, as no process can collide with op_p, no SwitchPair with op_p as parameter is ever invoked, and in particular $op_p.result = \perp$ for the rest of the execution. This in turn implies that upon recovery p will return Fail, as required.

5.2 Lock-Free Stack

The stack implementation is due to [....]. The TryPush routine tries to atomically have a new node pointing to the old top, and then updating the top to be the new node. The TryPop routine tries to atomically read the top of the stack, and change the top to the next node of it. The two routines uses **CAS** in order to gurantee no change for the top was made between the read and write. Push (resp. Pop) routine is alternating between a TryPush (TryPop) routine, which access the central stack, and the Exchange, trying to collide with an opposite operation.

In order to make the implementation recoverable, we need a way to infer whether a Pop or Push already took affect, in case of a crash. Moreover, in case of a Pop, we also need to infer which process is the one to pop the node. For that, we use an approach similar to the Linked-List implementation. Each node contains a new field *popby* which is used to identify a Push of the node completed, as well as a Pop of the node was completed, and who is the process to pop it. Consider the following scenario. Assume a process p performs a Push operation with node nd, and using a **CAS** succeed to update the stack top to point to nd, followed by a crash. Now, process q performing a Pop operation performs a **CAS** causing the removal of nd from the stack (by changing top to the next node). In this case, once p recovers, nd is no longer part of the stack, and it is also not marked as deleted. This is indistinguishable from a configuration in which the Push of nd was yet to take affect (a crash before **CAS**), and thus p can not know what the right response is.

One way to solve this issue is by first marking a node for removal, and only then remove it. This way, if a node is no longer part of the stack it must be marked, and thus we can conclude it was in the stack, and the Push routine was successful. However, such an implementation, in addition for the need of to system to support a markable reference, also requires process to help each other. If a node is marked for delete, then a process trying to perform a different operation first needs to complete the deletion, before applying its own operation, otherwise the physical delete of the node may not take place, leaving the node forever in the stack. As the original algorithm avoids any marking, and simply tries to swing the Top pointer, we would like to maintain this property.

A field *popby* is initialised to $\perp$ when a node nd is created. Once the node is successfully insert to the stack by a Push operation, the inserting process tries to mark it by changing *popby* to Null using a **CAS**. Before a process tries to remove the node from the stack during a Pop routine, it first mark it as part of the stack by doing the same thing, helping the inserting process conclude the node is in the stack. This replace the logic delete of the node, as we only need to know the node was part of the stack if it is removed. After a successful **CAS** to remove nd from the stack, another **CAS** is used in order to try and set *popby* to the identifier of the process who performed the **CAS**. The use of **CAS** to change *popby* from $\perp$to Null, and from Null to an identifier guarantee that only the first process to perform each of these **CAS** will succeed. Note that before writing an identifier to *popby* a process must try and set it to Null, and thus it can not store two different identifiers along in any execution.

The correctness proof follows the same guidelines as of the proof for the Linked-List. If a Push operation did not introduce a new node nd into the stack, then no process but p is aware of nd. Thus, upon recovery the Search routine will not find nd in the stack, nor its *popby* field has been changed, and the PUSH-ROCEOVER returns FAIL. Otherwise, *nd* was successfully inserted to the stack. As discessed above, the only way to delete *nd* from the stack is by first changing its *popby* field to NULL. Thus, upon recovery p will either find *nd* in the stack, using the SEARCH routine, or that *popby* is different then $\perp$ in case it was deleted, and in both cases it returns **true**. For the POP routine, if p tries to remove a node *nd* from the top of the stack and crash, then upon recovery it first check if *nd* is still in the stack using the SEARCH routine. If it is so, then clearly *nd* was yet to delete, and it returns FAIL. Otherwise, *nd* was deleted, either by p or by some other process. Only the first process of which to performs a **CAS**, writing its identifier to *popby* will return the value stored in *nd*, while the others return either $\perp$ (in the TRYPOP routine) or FAIL (in the POP-RECOVER routine).

Notice that both PUSH-ROCEOVER and POP-RECOVER are wait-free. Due to the structure of stack, no *next* pointer of any node in the stack is ever changed. Therefore, once a process reads Top at the beginning of its RECOVER routine, the chain of pointers from this Top to the last node in the stack is fixed for the rest of the execution, and thus traversing it using the SEARCH routine is wait-free.

```
Type Node {
    T value
    int popby
    Node *next
}

Type PushInfo {                ▷ subtype of Info
    Node *pushnd
}

Type PopInfo {                 ▷ subtype of Info
    Node *popnd
}

Type ExInfo {                  ▷ subtype of Info
    {EMPTY, WAITING, BUSY} state
    T value, result
    ExInfo *partner, *slot
}
```

Figure 8: Type definition

ExInfo *default* - global static ExInfo object with state = EMPTY

Algorithm 6: T EXCHANGE (ExInfo *slot, T *myitem*, long *timeout*)

```
long timeBound := getNanos() + timeout
ExInfo myop := new ExInfo(WAITING, myitem, ⊥, ⊥, slot)
Announce[pid] := myop
while true do
    if getNanos() > timeBound then
        myop.result := TIMEOUT                                      // time limit reached
        return TIMEOUT
    yourop := slot
    switch yourop.state do
        case EMPTY
            myop.state := WAITING                                   // attempt to replace default
            myop.partner := ⊥
            if slot.CAS(yourop, myop)  then                         // try to collide
                while getNanos() < timeBound do
                    yourop := slot
                    if yourop ≠ myop then                           // a collision was done
                        if youop.parnter = myop then                // yourop collide with myop
                            SWITCHPAIR(myop, yourop)
                            slot.CAS(yourop, default)               // release slot
                        return myop.result
                end
                // time limit reached and no process collide with me
                if slot.CAS(myop, default) then                     // try to release slot
                    myop.result := TIMEOUT
                    return TIMEOUT
                else                                                // some process show up
                    yourop := slot
                    if yourop.partner = myop then
                        SWITCHPAIR(myop, yourop)                    // complete the collision
                        slot.CAS(yourop, default)                   // release slot
                    return myop.result
            end
            break
        case WAITING                                                // some process is waiting in slot
            myop.partner := yourop                                  // attempt to replace yourop
            myop.state := BUSY
            if slot.CAS(yourop, myop)  then                         // try to collide
                SWITCHPAIR(myop, yourop)                            // complete the collision
                slot.CAS(myop, default)                             // release slot
                return myop.result
            break
        case BUSY                                                   // a collision in progress
            SWITCHPAIR(yourop, yourop.parnter)                      // help to complete the collision
            slot.CAS(yourop, default)                               // release slot
            break
    endsw
end
```

Algorithm 7: void SwitchPair(ExInfo *first*, ExInfo *second*)

```
/* exchange the valus of the two operations                                  */
first.result := second.value
second.result := first.value
```

Algorithm 8: T Visit (T *value*, int *range*, long *duration*)

```
/* invoke Exchange on a random entery in the collision array                 */
int cell := randomNumber(range)
return Exchange(exchanger[cell], value, duration)
```

Algorithm 9: T Exchange-Recover ()

```
ExInfo *myop := Announce[pid]                     // read your last operation record
ExInfo *slot := myop.slot                          // and the slot on which it acts
if myop.state = Waiting then
    /* crash while trying to exchange defualt, or waiting for a collision    */
    yourop := slot
    if yourop = myop then                           // still waiting for a collision
        if slot.CAS(myop, default) then                       // try to release slot
            return Fail
        else                                                   // some process show up
            yourop := slot
            if yourop.partner = myop then
                SwitchPair(myop, yourop)                      // complete the collision
                slot.CAS(yourop, default)                              // release slot
            return myop.result
    else if yourop.partner = myop then                 // yourop collide with myop
        SwitchPair(myop, yourop)                              // complete the collision
        slot.CAS(yourop, default)                                      // release slot
        return myop.result
if myop.state = Busy then
    /* crash while trying to collide with myop.partner                       */
    yourop := slot
    if yourop = myop  then                  // collide was successful and in progress
        SwitchPair(myop, myop.partner)                        // complete the collision
        slot.CAS(myop, default)                                        // release slot
        return myop.result
if myop.result ≠ ⊥  then
    return myop.result                          // collide was successfuly completed
else
    return Fail
```

Figure 9: Elimination Array routines

Algorithm 10: boolean TryPush (Node *nd)

```
/* attempt to perform Push to the central stack                         */
Node *oldtop := Top
nd.next := oldtop
if Top.CAS(oldtop, nd) then              // try to declare nd as the new Head
    nd.popby.CAS(⊥, Null)                          // announce nd is in the stack
    return true
return false
```

Algorithm 11: boolean Push (T myitem)

```
Node *nd = new Node (myitem)
nd.popby := ⊥
PushInfo data := new PushInfo (nd)
while true do
    Announce[pid] := data                 // declare - trying to push node nd
    if TryPush(nd) then                 // if central stack Push is successful
        return true
    range := CalculateRange()            // get parameters for collision array
    duration := CalculateDuration()
    othervalue := Visit(myitem, range, duration)                // try to collide
    if othervalue = Null then        // successfuly collide with Pop operation
        RecordSuccess ()
        return true
    else if othervalue = Timeout then                        // failed to collide
        RecordFailure ()
end
```

Algorithm 12: boolean Push-Roceover ()

```
Node *nd := Announce[pid].pushnd
if nd.popby ≠ ⊥ then                         // nd was announced to be in the stack
    return true
if Search(nd) || nd.popby ≠ ⊥ then   // nd in the stack, or was announced as such
    nd.popby.CAS(⊥, Null)                          // announce nd is in the stack
    return true
return Fail
```

Algorithm 13: boolean Search (Node *nd)

```
/* search for node nd in the stack                                      */
Node *iter := Top
while iter ≠ ⊥ do
    if iter = nd then
        return true
    iter := iter.next
end
return false
```

Figure 10: Push routine

Algorithm 14: T TryPop()

```
Node *oldtop := Top
Node *newtop
Announce[pid].popnd := oldnop                  // declare - trying to pop node oldtop
if oldtop = ⊥ then                             // stack is empty
|   return Empty
newtop := oldtop.next
oldtop.popby.CAS(⊥, Null)                      // announce oldtop is in the stack
if Top.CAS(oldtop, newtop) then                // try to pop oldtop by changing Top to newtop
|   if newtop.popby.CAS(Null, pid) then        // try to announce yourself as winner
|   |   return oldtop.value
else
|   return ⊥
```

Algorithm 15: T Pop ()

```
Node *result
PopInfo data := new PopInfo (Top)
while true do
|   Announce[pid] := data                      // declare - trying to perform Pop
|   result := TryPop()                         // attempt to pop from central stack
|   if result ≠ ⊥ then                         // if central stach Pop is successful
|   |   return result
|   range := CalculateRange()                  // get parameters for collision array
|   duration := CalculateDuration()
|   othervalue := Visit(Null, range, duration) // try to collide
|   if othervalue = Timeout then               // failed to collide
|   |   RecordFailure ()
|   else if othervalue ≠ Null then             // successfuly collide with Push operation
|   |   RecordSuccess ()
|   |   return othervalue
end
```

Algorithm 16: T Pop-Recover()

```
Node *nd := Announce[pid].popnd                // crash while trying to pop node nd
if nd = ⊥ then                                 // pop from an empty stack
|   return Empty
if Search(nd) then                             // nd was not removed from the stack
|   return Fail
nd.popby.CAS(Null, pid)                        // nd was removed.  Try to complete the operation
if nd.popby = pid then                         // you are the process to win the pop of nd
|   return nd.value
return Fail
```

Figure 11: Pop routine

References

[1] M. K. Aguilera and S. Frølund. Strict linearizability and the power of aborting. In *Tech. Rep. HPL-2003-241*, 2003.

[2] H. Attiya, O. Ben-Baruch, and D. Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *Proceedings of the Symposium on Principles of Distributed Computing*, PODC, 2018.

[3] R. Berryhill, W. M. Golab, and M. Tripunitara. Robust shared objects for non-volatile main memory. In *19th International Conference on Principles of Distributed Systems, OPODIS*, pages 20:1–20:17, 2015.

[4] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[5] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 105–118, 2011.

[6] W. M. Golab and A. Ramaraju. Recoverable mutual exclusion: [extended abstract]. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC*, pages 65–74, 2016.

[7] R. Guerraoui and R. R. Levy. Robust emulations of shared memory in a crash-recovery model. In *24th International Conference on Distributed Computing Systems (ICDCS)*, pages 400–407, 2004.

[8] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[9] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[10] J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Distributed Computing - 30th International Symposium, DISC*, pages 313–327, 2016.

[11] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, volume 11, pages 61–75, 2011.