Composable Recoverable Linearizability: Hierarchical Constructions for Non-Volatile Memory*

Hagit Attiya[1], Ohad Ben-Baruch[2], and Danny Hendler[3]

[1]Department of Computer-Science, Technion, hagit@cs.technion.ac.il
[2]Department of Computer-Science, Ben-Gurion University, ohadben@post.bgu.ac.il, +972(0)524261187 †
[3]Department of Computer-Science, Ben-Gurion University, hendlerd@cs.bgu.ac.il, +972(0)86428038

February 18, 2018

Abstract

We presents a novel abstract individual-process crash-recovery model for non-volatile memory, which enables *composition*, so that complex recoverable objects can be constructed in a modular manner from simpler recoverable base objects. Within the framework of this model, we define *composable recoverable linearizability* (CRL) – a novel correctness condition that captures the requirements for recoverable objects composability. Informally, CRL allows the recovery code to extend the interval of the failed operation until the recovery code succeeds to complete (possibly after multiple failures and recovery attempts). Unlike previous correctness definitions, the CRL condition implies that, following recovery, an implemented (higher-level) recoverable operation is able to complete its invocation of a base-object operation and obtain its response.

We present algorithms for *composable recoverable primitives*, namely, recoverable versions of widely-used primitive shared-memory operations such as read, write, test-and-set and compare-and-swap, which can be used to implement higher-level recoverable objects. We then exemplify how these recoverable base objects can be used for constructing a recoverable counter object.

Finally, we prove an impossibility result on wait-free implementations of recoverable test-and-set (TAS) objects from read, write and TAS operations, thus demonstrating that our model also facilitates rigorous analysis of the limitations of recoverable concurrent objects.

*Partially supported by the Israel Science Foundation (grant 1749/14) and by the Lynne and William Frankel Center for Computing Science at Ben-Gurion University.

†Contact author.

1 Introduction

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 [15]) 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* or *durable*): objects that are made robust to crash-failures by allowing their operations to recover from such failures.

In this paper, we present a novel abstract individual-process crash-recovery model for non-volatile memory, inspired by a model introduced by Golab and Ramaraju for studying the *recoverable mutual exclusion* (RME) problem [13]. Our model enables the composition of recoverable objects, so that complex recoverable objects can be constructed in a modular manner from simpler recoverable base objects. We assume that processes communicate via persistent shared-memory variables. Each process also has local variables stored in volatile processor registers. At any point, a process may incur a crash-failure, causing all its local variables to be reset to arbitrary values. A key challenge with which recovery code must cope in our model is that operation response values are returned via volatile processor registers. Hence, they may become inaccessible to the calling process if it fails just before persisting the response value. Each recoverable operation Op is associated with a *recovery function* that is responsible for completing Op upon recovery from a crash-failure.

Several correctness conditions for the crash-recovery model were defined in recent years (see, e.g., [2, 3, 14, 17]). The goal of these conditions is to maintain the state of concurrent objects consistent in the face of crash failures. However, guaranteeing object consistency is insufficient for composability. To illustrate this point, consider a base object $\mathcal{B}$ that supports atomic *compare&swap* (CAS) and *read* operations. Suppose that an operation Op by some recoverable object $\mathcal{O}$ invokes $\mathcal{B}.compare\&swap(old, new)$ in its algorithm which, upon completion, should return a success or failure response. Suppose also that some process p crashes inside Op, immediately after it invokes $\mathcal{B}.compare\&swap(old, new)$. Since the CAS operation is atomic, the consistency of $\mathcal{B}$ is ensured in spite of p's failure: either the CAS took effect before the failure, or it did not. However, the consistency of $\mathcal{B}$ is insufficient, since once p recovers from the crash, it has no way of knowing whether the CAS succeeded or not, as its response was written to a volatile local variable that was lost because of the crash. Reading $\mathcal{B}$ will not help, since even if *new* was written before the crash, it may have been overwritten by other processes since then. Op may therefore not be able to proceed correctly.

To address this issue, we define within the framework of our model the notion of *composable recoverable linearizability* (CRL), a novel correctness condition that captures the requirements for recoverable objects composability. Informally, CRL allows the recovery code to extend the interval of the failed operation until the recovery code succeeds to complete (possibly after multiple fail-

ures and recovery attempts). CRL implies that, following recovery, an implemented (higher-level) recoverable operation is able to complete its invocation of a base-object operation and obtain its response.

We present several algorithms for *composable recoverable primitive objects*, recoverable versions of widely-used primitive shared-memory operations such as read, write, test-and-set and compare-and-swap, that can be used by higher-level recoverable objects. We also provide an example of how these recoverable base objects can be used for constructing a recoverable counter object.

In addition, we prove an impossibility result on wait-free implementations of recoverable test-and-set (TAS) objects from read, write and TAS operations, thus demonstrating that our model also facilitates rigorous analysis of the limitations of recoverable concurrent objects.

2 Model and Definitions

We consider a system where N asynchronous *unreliable processes* $p_1, \ldots, p_N$ communicate by applying operations to *concurrent objects*. The system provides *base objects* that support atomic read, write, and read-modify-write operations. Base objects can be used for implementing more complex concurrent objects (e.g. counters, queues and stacks), by defining access procedures that simulate each operation on the implemented object using operations on base objects. These may be used in turn similarly for implementing even more complex objects, and so on.

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, in which *Op's response is stored to a local variable of p*. It follows that the response value is lost if p incurs a crash before persisting it.

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.[1] A *recoverable operation* Op is associated with a *recovery function*, denoted Op.`Recover`, that is responsible for completing Op upon recovery from a crash. The execution of operations (and recoverable operations in particular) may be nested, that is, an operation Op_1 can invoke another operation Op_2. Following a crash of process p that occurs when p has one or more pending recoverable operations, the system eventually resurrects process p by invoking the recovery function of the inner-most recoverable operation that was pending when p failed. This is represented by a *recover step for p*

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'*;

[1] This assumption can be removed, but this would require substantial changes to the notions of sequential executions and linearizability which we chose to avoid in this work.

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`,[2] where Op is the crashed operation of s'. We say that s *is the recovery step that matches* s'.

When a recovery function Op.`Recover` is invoked by the system to recover from a crash represented by step s, we assume it receives the same arguments as those with which Op was invoked when that crash occurred. We also assume that Op.`Recover` has access to a designated per-process non-volatile variable LI_p, identifying the instruction of Op that p was about to execute in the crash represented by s. An object is a *recoverable object* if all its operations are recoverable. In the following definitions, we consider only histories that arise from operations on recoverable objects or atomic primitive operations.

Consider a scenario in which p incurs a crash (represented by a crash step s) immediately after a recoverable operation Op completes (either directly or through the completion of Op.`Recover`) – an event represented by a response step r. In this case, the response value is lost and, moreover, Op.`Recover` will not be invoked by the system, since the crashed operation of s is no longer Op but, rather, the operation Op' that invoked Op. In general (although there are exceptions to the rule as we will see in Section 3), Op' may therefore not be able to proceed correctly. Hence, it is sometimes required to guarantee that a recoverable operation returns only once its response value gets persisted. This is defined formally as follows.

Definition 1. *A recoverable operation Op is a strict recoverable operation if whenever it is completed, either directly or by the completion of Op.*`Recover`*, (an event represented by a* (RES, p, O, Op, ret) *step), ret is stored in a designated persistent variable accessed only by process p.*

For a history H, we let $H|p$ denote the subhistory of H consisting of all the steps by process p in H. We let $H|O$ denote the subhistory of H consisting of all the invoke and response steps on object O in H, as well as any crash step in H, by any process p, whose crashed operation is an operation on O and the corresponding recover step by p (if it appears 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. A crash-free subhistory $H|O$ is well-formed, if for all processes p, $H|<p, O>$ is a sequence of alternative matching invocation and response steps, starting with an invocation step.

Given two operations op_1 and op_2 in a history H, we say that op_1 *happens before* op_2, denoted by $op_1 <_H op_2$, if op_1's response step precedes the invocation step of op_2 in H. If neither $op_1 <_H op_2$ nor $op_2 <_H op_1$ holds then we say that op_1 and op_2 are *concurrent* 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 (that may end by a pending invocation). The *sequential specification* of an object O is the set of all possible (legal) sequential histories over O. H is a *sequential history* if $H|O$ is a sequential object history for all objects O.

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

A crash-free history H is *well-formed* if: 1) $H|O$ is well-formed for all objects O, and 2) For all p, if i_1, r_1 and i_2, r_2 are two matching invocation/response steps in $H|p$ and $i_1 <_H i_2 <_H r_1$ holds, then $r_2 <_H r_1$ holds as well. The second requirement guarantees that if operation Op_1 invokes operation Op_2, Op_2's response must precede Op_1's response. Two histories H and H' are *equivalent*, if $H|<p, O> = H'|<p, O>$ for all processes p and objects O. A history H is *sequential*, if $H|O$ is sequential for all objects O that appear in H. 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 2 (Linearizability [16], 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

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 3 (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 composable recoverable linearizability.

Definition 4 (Composable Recoverable Linearizability (CRL))**.** *A finite history H satisfies* composable recoverable linearizability *(CRL) if it is recoverable well-formed and $N(H)$ is a linearizable history. An object implementation satisfies CRL if all of its finite histories satisfy CRL.*

3 Recoverable Base Objects

In this section, we present algorithms for recoverable base objects that support primitive operations such as read, write, compare-and-swap (CAS) and test-and-set. As described in Section 2, this consists in implementing a recovery function Op.`Recover` for each such operation, which is invoked by the system upon a crash-failure when Op is the crashed operation. In the pseudo-code, we use names that start with a capital letter for shared-memory variables and lower-case names for local variables. We also use capital-letter names for implemented operations and lower-case names for primitive operation names.

In this paper, we only consider recoverable objects or base objects provided by the system that support atomic read, write or read-modify-write (RMW) operations. Before presenting our algorithms we prove that, under these assumptions, our model guarantees that all histories are recoverable well-formed.

Lemma 1. *Let H be a history in which all operations are applied to either recoverable objects or atomic base objects. Then H is recoverable well-formed.*

Proof. Consider an execution α of any algorithm and the corresponding history $H(\alpha)$. If an atomic read, write or RMW is executed in α, then its invocation and response steps appear in H consecutively hence cannot violate well-formedness, so it suffices to consider invocation/response steps on recoverable objects and crash/recovery steps.

When a process p invokes an operation Op on a recoverable object O, a corresponding invocation step i is appended to $H(\alpha)$. If p crash-fails inside Op, a $(CRASH, p)$ step c is appended to $H(\alpha)$ and its crashed operation is Op. The system eventually resurrects p and invokes Op.`RECOVER`, thus a recovery step r that matches c is appended to $H(\alpha)$. If p undergoes additional failures before Op completes, then additional pairs of a crash step and its matching recovery step are appended to $H(\alpha)$. Otherwise, if and when Op completes, a response step r that matches i is appended to $H(\alpha)$. It follows from Definition 3 that $H(\alpha)$ is a recoverable well-formed history. □

A Recoverable Read-Write Object Our recoverable read-write object algorithm assumes that all values written to the object are distinct. This assumption can be easily satisfied by augmenting each written value with a tuple consisting of the writing process' ID and a per-process sequence number. In some cases, such as the example of a recoverable counter that we provide later, this assumption is satisfied due to object semantics and does not require special treatment.

Algorithm 1 presents pseudo-code for process p of a recoverable read-write object R. It supports non-strict (see Definition 1) recoverable `READ` and `WRITE` operations. Both `READ` and `READ.RECOVER` simply return R's current value (lines 7-9, 18-20.

Our implementation of `WRITE` "wraps" the write primitive with code that allows the recovery function `WRITE.RECOVER` to conclude whether p's write in line 4, or a write by a different process, took place since p's invocation of `WRITE`. This is done using a single-reader-single-write shared-memory variable S_p that stores a pair of values – R's previous value (read in line 2) and a flag that allows the recovery function to infer the location in `WRITE` where the failure occurred. Specifically, $flag = 0$ holds whenever p is either not performing a `WRITE` operation on R (since it is initialised to 0) or p has performed line 5 but `WRITE` was not yet completed.

The intuition for correctness is the following. A crash before line 3 executes implies that both $S_p = <0, curr>$ and $curr \neq val$ hold, since we assume each value written to R is distinct. In this case, `WRITE.RECOVER` simply re-executes `WRITE` (lines 12-13). A failure after line 5 executes implies that $S_p = <0, val>$ holds, in which case `WRITE.RECOVER` returns *ack*. Otherwise neither of the conditions of lines 12,14 holds, hence either p already executed line 4, or another process wrote to R between when R was read by p in `WRITE` (line 2) and in `WRITE.RECOVER` (line 14). In either of these cases, we may linearize `WRITE`, so the recovery function updates S_p and returns *ack* (lines 16-17). A formal correctness proof follows.

Lemma 2. *Algorithm 1 satisfies CRL.*

Proof. First observe that R is a recoverable object since each of its two operations has a corresponding `RECOVER` function. Consider an execution α of the algorithm and the corresponding history $H(\alpha)$. From Lemma 1, $H(\alpha)$ is recoverable well-formed. Hence, $H'(\alpha) = N(H(\alpha))$ is a well-formed (non-recoverable) history. Following definition 4, it is enough to prove that $H'(\alpha)|R$ is linearizable.

Algorithm 1 recoverable read/write object R, program for process p

Shared variables: S_p - pair of values, init $< 0, null >$

```
1: procedure WRITE(VAL)
2:     temp ← R
3:     S_p ←< 1, temp >
4:     R ← val
5:     S_p ←< 0, val >
6:     return ack
7: procedure READ()
8:     temp ← R
9:     return temp
10: procedure WRITE.RECOVER(VAL)
11:     < flag, curr >← S_p
12:     if flag = 0 ∧ curr ≠ val then
13:         proceed from line 2
14:     else if flag = 1 ∧ curr = R then
15:         proceed from line 2
16:     S_p ←< 0, val >
17:     return ack
18: procedure READ.RECOVER()
19:     temp ← R
20:     return temp
```

Since neither of R's recovery functions write to variables read by other processes, they have no effect on their execution. We can therefore ignore crashes that occur during the execution of the recovery functions, as long as S_p is not written (which only occurs in line 16 of `WRITE.RECOVER`).

Assume p applies a `WRITE`(val) operation to R in α. If p does not fail during its execution, then clearly p writes to R exactly once in line 4 and this is the operation's linearization point. Otherwise, assume that p fails when executing the `WRITE` operation. A crash before line 3 implies that p did not yet write to neither S_p nor R, hence `WRITE` was not linearized yet. Upon recovery, p reads $S_p = <0, curr>$, where $curr \neq val$ holds, since we assume all written values are distinct. Hence, `WRITE.RECOVER` re-executes `WRITE`.

A crash between the two writes to S_p (in lines 3 and 5) implies that $S_p = <1, curr>$ and $curr \neq val$ holds. Upon recovery, if the condition of line 14 is satisfied, then $curr = R$ ensures that no process wrote to R between the two reads of R (in lines 2 and 14). In particular, p did not write to R (so `WRITE` was not linearized) and the operation is re-executed.

Otherwise $curr \neq R$, i.e., there was a write to R between the two reads of R by p. If p wrote to R in line 4, then the operation was already linearized at this point. Else, there was a write by a different process q that occurred between the two reads by p, hence within the execution interval of p's `WRITE`. Thus, p's `WRITE` can be linearized immediately before q's, causing q to immediately overwrite val (if it was indeed written by p in line 4), resulting in an execution that is indistinguishable to all processes from one in which p does not write to R at all. It follows that, in both these cases, p's `WRITE` operation can be linearized correctly and `WRITE.RECOVER` returns ack.

The last case to consider is when `WRITE.RECOVER` reads $<0, val>$ in line 11 and then performs lines 16-17 and returns. This can occur either if p crashed after executing line 5, or if it crashed before line 5 but a subsequent `WRITE.RECOVER` failed after updating S_p (in line 16). The latter case can happen only if a previous invocation of `WRITE.RECOVER` by p read $S_p = <1, curr>$, for $curr \neq R$, executed line 16 and then failed. Our previous analysis established that `WRITE` can be linearized also in this case. This concludes our discussion of linearization points of `WRITE` operations. As for `READ`, if and when it returns (in lines 9 or 20), it is linearized when it last read R (in line 8 or line 19, resp.). □

A Recoverable Compare-and-Swap Object A Compare-and-Swap (CAS) object supports the `CAS`(old, new) operation, which atomically swaps the object's value to *new* only if the value it stores is *old*. The operation returns true and we say it is *successful* if the swap is performed, otherwise it returns false and we say it *fails*. A CAS object also supports a `READ` operation which returns the object's value. Algorithm 2 presents pseudo-code for process p of a recoverable CAS object C.

C stores two fields, both initially *null*. The first is the ID of the last process that performed a successful `CAS` and the second is the value it wrote. Both `READ` and `READ.RECOVER` simply return $C.val$. Inside the `CAS` operation, process p first reads C. If $C.val$ was written by process q, then p stores it in $R[q, p]$ which is a SRSW shared variable used by p to inform q. This allows processes to inform each other which `CAS` operations were successful. We assume that `CAS` is never invoked with $old = new$ and that values written to C by the same process are distinct. This assumption can be easily satisfied by augmenting each written value with a per-process sequence number.

A process p first reads C. If it reads a value v other than *old* it returns *false* and is linearized at the read. Otherwise, if $v \neq null$, p informs the process (say, q) that wrote v that its CAS took effect by setting $R[q][p] \leftarrow v$. This mechanism guarantees that, upon recovery, process p is able to determine that its `CAS` operation took effect if the value it wrote is still stored in C or is written in $R[p][j]$, for some j.

If p crash-fails inside `CAS` without modifying C, either because it crashes before line 7 or because it crashed after its *cas* in line 7 failed, then $<p, new>$ is never written to C or to any of $R[p][*]$. In this case, `CAS.RECOVER` simply re-executes `CAS`. A key point in the correctness argument is that a `CAS` operation that crashed after a failed (primitive) *cas* can be re-executed, since it did not affect other processes, hence we may assume it was not linearized yet and re-execute it without violating the sequential specification of CAS.

Lemma 3. *Algorithm 2 satisfies CRL.*

Proof. First observe that C is a recoverable object since each of its two operations has a corresponding `RECOVER` function. Consider an execution α of the algorithm and the corresponding history $H(\alpha)$. From Lemma 1, $H(\alpha)$ is recoverable well-formed. Hence, $H'(\alpha) = N(H(\alpha))$ is a

Algorithm 2 recoverable CAS object C, program for process p

Shared variables: R[N][N] - all elements init *null*

```
1:  procedure CAS(OLD,NEW)
2:      <id, val> ← C.read()
3:      if val ≠ old then
4:          return false
5:      if id ≠ null then
6:          R[id][p] ← val
7:      ret ← C.cas(<id, val >, <p, new>)
8:      return ret
9:  procedure READ()
10:     <id, val> ← C
11:     return val
12: procedure CAS.RECOVER(OLD, NEW)
13:     if C = <p, new> ∨
            new ∈ {R[p][1], ..., R[p][N]}  then
14:         return true
15:     else
16:         proceed from line 2
17: procedure READ.RECOVER()
18:     <id, val> ← C
19:     return val
```

well-formed (non-recoverable) history. Following definition 4, it is enough to prove that $H'(\alpha)|C$ is linearizable. Since none of the recovery functions writes to a variable read by other processes, they have no effect on their executions. We can therefore ignore crashes that occur during the execution of the recovery function.

For presentation simplicity, when we refer to *the value of* C we refer the value of its second field, which represents the state of the object, and when refer to *the content of* C, we refer to both fields as a pair. Since no process writes the same value twice, it follows that the content of C is unique, while the values of C are not necessarily unique, since different processes may write the same value.

Assume p applies a `CAS`(old, new) operation to C in α. First assume p does not crash during the `CAS` operation. In line 2, p reads C and then compares the value of C to *old*. If the values differ, then the operation is linearized at the read in line 2 and indeed at this point the value of C is different from *old*, so p returns $false$ in line 4. Otherwise, p informs the last process q to have performed a successful `CAS` that its operation took effect. It does so by writing to $R[q][p]$ the value it read from C. A two dimensional array R of SRSW registers is used for this purpose. Following that, p tries to change the value of C by performing CAS in line 7.

There are two scenarios to consider. Assume first there is a successful `CAS` applied to C between p's execution of line 2 and line 7. In this case, the first such operation must change C's value to a value other than *old*, and we can linearize p's operation right after it. Since the content of C is unique, p's CAS in line 7 fails, and it returns $false$ in line 8. Otherwise, there was no successful `CAS` to C between the read in line 2 and the CAS in line 7, and thus the CAS in line 7 is successful, and this is also the linearization point and p returns $true$ in line 8.

Assume now that p crashes during a `CAS` operation. If p did not write to C, either because the crash occurs before line 7, or due to a failed CAS operation in line 7, then $<p, new>$ is not written to C. This follows from the fact that p writes distinct values, thus *new* was not written by p before. As processes writes to R only values read from C, no process writes *new* to $R[p][*]$. As a result, `CAS.RECOVER` re-executes `CAS`. A failed CAS operation does not affect other processes, that is, removing the operation is indistinguishable to other processes. Therefore, in both cases, considering the operation as not having a linearization point so far and re-executing it does not violate the sequential specification of CAS.

If p did perform a successful `CAS` in line 7 before the crash, then this is also the operation's linearization point. We argue that the next process q to perform a successful `CAS` on C, if any, must write *new* to $R[p][q]$ before its `CAS` takes effect. Assume there exists such a process q. Then, consider the time when the successful `CAS` of q in line 7 occurs. It must be that q executes line 2 to line 7 without crashing, since any crash before writing to C will cause the re-execution of the `CAS` operation, as we have already shown. In addition, q reads $<p, new>$ from C in line 2, since by our assumption it succeeds in replacing p's value, that is, it performed a successful CAS when C's content was $<p, new>$. As the content of C is unique, reading any other content implies the CAS in line 7 fails, a contradiction. Hence, before q's successful `CAS` in line 7, it writes *new* to $R[p][q]$ in line 6. We assume that the reads in line 13 of `CAS.RECOVER` are done from left to right. As a result, in line 13, p either reads $C = <p, new>$ or, otherwise, another process already replaced C's content but wrote *new* to $R[p][*]$ before that, thus p observes *new* in $R[p][*]$. In both cases, p considers the `CAS` operation as successful and returns true in line 14.

As for `READ`, if and when it returns (in lines 11 or 19), it is linearized when it last read C (in line 10 or line 18, resp.)

□

A Recoverable Test-and-Set Object A non-resettable Test-and-Set (TAS) object is initialized to 0 and supports the `T&S` operation. It atomically writes 1 and returns the previous value. In the following, we present a recoverable non-resettable test-and-set algorithm that uses (non-recoverable) non-resettable TAS objects and read/write shared variables. We first present an impossibility result on such implementations of recoverable TAS algorithms. We say that an operation (or a recovery function) is *wait-free* [15], if a process that does not incur failures during its execution completes it in a finite number of its own steps. Our implementation of TAS has a wait-free `T&S` operation but its recovery code is blocking. The following theorem proves that this is inevitable: any recoverable TAS object from these primitives cannot have both a wait-free `T&S` operation and a wait-free `T&S.RECOVER` function.

Theorem 1. *There exists no recoverable non-resettable TAS algorithm satisfying CRL, from read/write and (non-recoverable) non-resettable TAS base objects only, such that both the `T&S` operation and the `T&S.RECOVER` function are wait-free.*

Proof. We prove the theorem using valency arguments [10]. Let $\mathcal{A}$ be a CRL recoverable non-resettable TAS implementation using the base objects assumed by the theorem, and assume towards a contradiction that both `T&S` and `T&S.RECOVER` are wait-free. We say that a configuration C is p-valent (resp. q-valent) if there exists a crash-free execution starting from C in which p (resp. q) returns 0 or has already returned 0. C is bivalent if it is both p-valent and q-valent, and univalent otherwise. Observe that any configuration C is either p-valent or q-valent (or both), because in a solo execution of p followed by a solo execution of q from C where both complete their operations (if they haven't done so already), exactly one must return (or already returned) 0.

The initial configuration C_0, in which both p and q invoke the TAS operation, is bivalent – a solo execution of each process returns 0. Using valency arguments and as we assume that `T&S` is wait-free, there is an execution starting from C_0 that leads to a bivalent configuration C_1, in which both p and q are about to perform a critical step. This step must be an application of the $t\&s$ primitive to the same base object. Moreover, a step by any of the processes leads to a different univalent configuration. Assume wlog that configuration $C_1 \circ p$ is p-valent whereas $C_1 \circ q$ is q-valent.

Let p and then q perform their next $t\&s$ steps, followed by a crash step of p. Since p's response from the $t\&s$ primitive is lost due to the crash, upon recovery p does not know whether the $t\&s$ primitive was performed, and if it was, what the response value was. Specifically, configurations $C_1 \circ p \circ q \circ CRASH_p$ and $C_1 \circ q \circ p \circ CRASH_p$ are indistinguishable to p. Consequently, an execution of `T&S.RECOVER` by p after both configurations (which will complete since we assume that `T&S.RECOVER` is wait-free) returns the same value ret. This implies that both configurations are u-valent for some u.

Wlog assume $u = p$ (the other case is symmetric), then we consider configuration $C_1' = C_1 \circ q \circ p \circ CRASH_p$. Note that C_1' is indistinguishable from configuration $C_1 \circ q \circ p$ for q, since q is not aware of p's crash. Therefore, C_1' is both q-valent and p-valent, that is, it is bivalent. Since we assume that both processes are only allowed to use read, write and t&s primitives, we can repeat the same argument again and show that there is an extension of C_1' leading to a bivalent configuration C_2, where both p and q are about to perform a critical step. Moreover, this step must be the application of a t&s primitive to the same base object. This must be a TAS base object other than those previously used in the execution, since those would always return 1, contradicting criticality.

Continuing in this manner, we construct a crash-free execution of q in which it executes an infinite number of steps while performing a single `T&S` operation. This is a contradiction. □

Algorithm 3 presents a pseudo-code for process p of a recoverable TAS object T. It supports strict recoverable `T&S` operation. For simplicity we allow a process to return $true$ or $false$, representing 0 and 1 resp. We assume a process invoke `T&S` operation at most once, as any additional operation returns 1.

A doorway mechanism guarantees any operation invoke after a process sets the doorway will returns 1. This implies an operation which return 0 can be linearized before any other operation. The doorway can be replaced with reading T in line 3, in case of a TAS object which supports also a read operation. After passing the doorway, a process tries to win the TAS object in line 8, and the winning process declare itself as so by writing to a designated variable $Winner$. Once a process writes to $Winner$ any process can recover by simply read $Winner$. Hence, the main difficulty is to make sure a single process writes to $Winner$.

A crash before line 6 executes implies the operation did not affect any other process so far, and `T&S.RECOVER` re-executes `T&S`. A crash after setting $R[p] \leftarrow 3$, in either line 12 or 33 implies p

Algorithm 3 recoverable TAS object T, program for process p

Shared variables:

- R[N]: an array, init $[0, \ldots, 0]$
- Winner, Doorway: read/write registers, init $null$, $true$ resp.

```
1:  procedure T&S()
2:      R[p] ← 1
3:      if Doorway = false then
4:          ret ← 1
5:          proceed from line 11
6:      R[p] ← 2
7:      Doorway ← false
8:      ret ← T.t&s()
9:      if ret = 0 then
10:         Winner ← p
11:     Res_p ← ret
12:     R[p] ← 3
13:     return ret

14: procedure T&S.RECOVER()
15:     if R[p] < 2 then
16:         proceed from line 2
17:     if R[p] = 3 then
18:         ret ← Res_p
19:         return ret
20:     if Winner ≠ null then
21:         proceed from line 31
22:     Doorway ← false
23:     R[p] ← 4
24:     T.t&s()
25:     for i from 0 to p − 1 do
26:         await(R[p] = 0 or R[p] = 3)
27:     for i from p + 1 to N do
28:         await(R[p] = 0 or R[p] > 2)
29:     if Winner = null then
30:         Winner ← p
31:     ret ← (Winner ≠ p)
32:     Res_p ← ret
33:     R[p] ← 3
34:     return ret
```

already computed its response and wrote it to Res_p. In such case, `T&S.RECOVER` returns the value stored in Res_p. A crash after some process wrote to $Winner$ implies a winner already declared itself, thus in `T&S.RECOVER` p check to see if it is the winner and answer accordingly.

If none of the above cases holds then p is still competing to win the TAS. Therefore, it closes the doorway in line 22, in case it is still open, and announce it is recovering and competing for the TAS by writing 4 to $R[p]$ in line 23. Following that, it has to pass two for loops in lines 25 and 27. In the first one it waits for any running process with lower id to complete its operation. In the later, it waits for any running process with higher id to either complete or announce itself as recovering. The for loops guarantee that if there is a process writing to $Winner$ in line 10, then p must wait for it to do so. Also, only the smallest id process which gets to the for loops can complete them and write to $Winner$ in line 30, while the rest must wait for it to complete. This implies at most a single process writes to $Winner$, and eventually one must do so. As argued before, once a process writes to $Winner$, upon completing its operation it returns 0.

Claim 1. *Algorithm 3 satisfies CRL.*

Proof. First observe that T is a recoverable object since each of its two operations has a corresponding `RECOVER` function. Consider an execution α of the algorithm and the corresponding history $H(\alpha)$. From Lemma 1, $H(\alpha)$ is recoverable well-formed. Hence, $H'(\alpha) = N(H(\alpha))$ is a well-formed (non-recoverable) history. Following definition 4, it is enough to prove that $H'(\alpha)|T$ is linearizable.

The proof relies on the following simple observations:

1. If there is a process which complete its operation, there must be a write to $Doorway$.
2. Once a process writes the $Doorway$, any operation yet to execute line 2 returns 1 if given enough time with no crash. Moreover, such an operation can only set $R[p]$ to either 1 or 3.
3. Once a process writes 3 to $R[p]$ (in line 12 or 33), its response is persistent in Res_p. In addition, $R[p]$ is fixed for the rest of the execution, and if p given enough time with no crash it returns the value stored in Res_p.
4. A process which returns 0 must also write to $Winner$.
5. Assume a process p writes to $Winner$, and it is the only process to do so. Then if given enough time with no crash p eventually returns 0.

If no `T&S` operation completes in α, then H' is obviously linearizable. Thus, assume there is a complete `T&S` operation, either normally or by completing a `T&S.RECOVER` execution. By observation 1, there is a write to $Doorway$ in α. Let t be the time of the first such write. There is no operation to complete before time t, otherwise there is an earlier write to $Doorway$, in contradiction. Also, by observation 2, any operation yet to execute line 2 at time t can only return 1, and set $R[p]$ to either 1 or 3.

The proof composed of two claims - there can be no two operations returning 0, and if any process is given enough time with no crash there must be such an operation. It follows that either there is a single operation pending at time t which returns 0 in α, or there is no such operation, but there is a pending operation at time t that is yet to complete in α. In both cases we can linearize one operation at time t as returning 0, while the rest can be linearize after it, and they all return 1. This proves H' is linearizable.

Following observation 4, it suffices to prove there can be no two processes writing to $Winner$. Assume there is a process p writing to $Winner$ in line 10. Then, no other process can write to $Winner$ in line 10, since T returns 0 exactly once. Assume towards a contradiction there exists a

process q writing to $Winner$ in line 30. Then, at time t, q is before executing line 23 by definition. In order to write to $Winner$ q must go through the for loops in the `T&S.RECOVER` function, and thus have to wait for p to set $R[p] > 2$ in any case. By observation 2, at time t p is after executing line 2. Also, p can not set $R[p]$ to 4 in line 23, as this implies p crash after executing line 6 and before writing to $Winner$ (since it reads *null* in line 20). In particular, p does not re-execute any part of `T&S` upon recovery, contradicting the fact p writes to $Winner$ in line 10. Therefore, it must be that p sets $R[p] \leftarrow 3$, and this happens only after it writes to $Winner$. As a result, q gets to line 29 only after p writes to $Winner$, and therefore does not write to $Winner$ in line 30, in contradiction.

Assume now there is a process p writing to $Winner$ in line 30. As proved above, no process can write to $Winner$ in line 10. Assume towards a contradiction there exists another process q which writes to $Winner$ in line 30. WLOG, assume $p < q$. At time t both processes are after executing line 2 and before executing line 25. Hence, in order to get to line 30 q has to wait for p to set $R[p] \leftarrow 3$, that is, to complete its operation. This happens only after p writes to $Winner$, thus in line 20 q observes a value different then *null* and does not write to $Winner$, in contradiction.

We now prove that if any process is given enough time with no crash, then eventually some process writes to $Winner$. Assume towards a contradiction no process writes to $Winner$. If no operation crash after setting $R[p] \leftarrow 2$ in line 6, then the first operation to execute line 8 also writes to $Winner$, in contradiction. Thus, there must be such an operation, and it eventually executes `T&S.RECOVER` and set $R[p] \leftarrow 4$ in line 23. Let q be the process with the smallest id to do so. Any operation which does not execute line 23, when given enough time with no crash, eventually sets $R[p]$ to 3 and returns. Then, at some point the following holds: any $i < q$ satisfy $R[i] \in \{0, 3\}$ (since q is the smallest to execute line 23), and any $i > q$ satisfy $R[i] \in \{0, 3, 4\}$. Therefore, eventually p complete the for loops in `T&S.RECOVER`, and writes to $Winner$ in line 30, in contradiction.

□

Using Recoverable Base Objects: an Example A *Counter* object supports an `INC` operation that atomically increments its value and a `READ` operation. We start by describing a simple linearizable implementation of a *Counter* object and then discuss the changes required for making it a recoverable *Counter* satisfying CRL. Each process p has its own entry $R[p]$ in an array R of integers, initialized to 0. To perform `INC`, p simply increments $R[p]$. In the `READ` operation, p reads all array entries, sums up the values and returns the sum.

Claim 2. *The Counter algorithm described above is linearizable.*

Proof. The correctness argument is as follows. Assume process p completes a `READ` operation. Let v be the value it read from $R[q]$. Since $R[q]$ is monotonically increasing, it must be that $R[q] \leq v$ immediately after the `READ`'s invocation and $R[q] \geq v$ immediately before the `READ`'s response. Therefore, at most v `INC` operations by q have been linearized before the `READ`'s invocation, and at least v `INC` operations were linearized before the `READ`'s response. As this is true for any q, it implies that, as p computes value *val*, then there are at most *val* `INC` operations that were linearized before its `READ` invocation, and at least *val* `INC` operations that were linearized before its `READ` response. In particular, there is a point during `READ`'s execution when exactly *val* `INC` operations were linearized, so we linearize the `READ` operation at such a point. □

Making this implementation recoverable necessitates equipping both operations with a `RECOVER` function. The pseudo-code of the recoverable implementation is shown by Algorithm 4. As we described in Section 2, our system model assumes that a recovery function *Op*.`RECOVER` has access

to LI_p - a designated per-process non-volatile variable identifying the instruction of Op that p was about to execute when it incurred the crash that triggered Op.`RECOVER`. Our implementation of the `INC` operation, which we now describe, exemplifies how LI_p is used.

The `INC` operation's write to $R[p]$ in line 4 is its linearization point. In order to ensure that $R[p]$ is incremented exactly once in each `INC` operation, we use an array R of recoverable read/write objects, such as the one we described previously, instead of an array of (non-recoverable) read/write variables. Recall that our implementation of recoverable `WRITE` requires that distinct values will be written, but this imposes no overhead here since it is ensured by the counter's semantics.

Consider a crash that occurs between `INC`'s invocation and response. If p crashes inside `WRITE` (invoked by `INC` in line 4), then `WRITE` is the inner-most recoverable operation that was pending, so `WRITE.RECOVER` is invoked by the system and, once its recovery is completed, `INC` returns in line 5 (unless it crashes again). Otherwise, the system invokes `INC.RECOVER`, which uses LI_p in line 7 to determine whether the last crash inside `INC` occurred before line 4 – in which case `INC` is re-executed, or after it – in which case `INC.RECOVER` simply returns.

We chose to implement the counter's `READ` operation as strict recoverable. This was accomplished in our implementation by having `READ` write its response value, immediately before it returns, to a shared-memory variable Res_p, used by p only. This ensures that a recoverable operation that invokes the counter's `READ` operation is able to access its response even if the process fails immediately after N.`READ` returns.

Algorithm 4 demonstrates how recoverable base objects that satisfy the CRL condition can be used in our model for constructing more complex recoverable objects (satisfying the same condition) relatively easily. Modular constructions, such as that of Algorithm 4, leverage the following key property guaranteed by CRL: base recoverable operations are guaranteed to be linearized correctly before they return, even in the presence of multiple crashes, allowing the implemented recoverable operation to proceed correctly.

4 Related Work

Golab and Ramaraju [13] define an abstract individual-process crash-recovery model to study the *recoverable mutual exclusion* (RME) problem. RME is a generalization of the standard mutual

Algorithm 4 recoverable Counter object N, program for process p

Shared variables: R[N]: an array of recoverable read/write objects, init $[0, \ldots, 0]$

```
1:  procedure INC()
2:      temp ← R[p].READ
3:      temp ← temp + 1
4:      R[p].WRITE(temp)
5:      return ack
6:  procedure INC.RECOVER()
7:      if LI_p < 4 then
8:          proceed from line 2
9:      else
10:         return ack
11: procedure READ()
12:     val ← 0
13:     for i from 1 to N do
14:         val ← val + R[i].READ
15:     Res_p ← val
16:     return val
17: procedure READ.RECOVER()
18:     proceed from line 12
```

exclusion problem, whereby a process that crashes while accessing a lock is resurrected eventually and allowed to execute recovery actions. The RME problem was further studied in [12, 18]). Some additional works investigated lock recoverability in different models [1, 5, 6, 7, 23].

A *consistency* condition specifies how to derive the semantics of a concurrent implementation of a data structure from its *sequential specification*. This requires to disambiguate the expected results of concurrent operations on the data structure, as done for example in *linearizability* [16]. Linearizability guarantees a locality property required for composability, but it cannot be directly used for specifying recoverable objects, since it has no notion of an aborted or failed operation, and hence, no notion of recovery.

Several consistency conditions for crash-recovery model were suggested, aiming to maintain the consistent state of concurrent objects. However, none of them ensures that a process is able to infer whether the failed operation completed and, if so, to obtain its response value.

Aguilera and Frølund [2] proposed *strict linearizability* as a correctness condition for recoverable objects. It treats the crash of a process as a response, either successful or unsuccessful, to the interrupted operation. Strict linearizability preserves both *locality* [15] and program order. Guerraoui and Levy [14] define *persistent atomicity*. It is similar to strict linearizability, but allows an operation interrupted by a failure to take effect before the subsequent invocation of the same process, possibly after the failure. They also proposed *transient atomicity*, which relaxes this criterion even further and allows an interrupted operation to take effect before the subsequent write response of the same process. Both conditions ensure that the state of an object will be consistent in the wake of a crash, but they do not provide locality.

Berryhill et al. [3] present an alternative condition, called *recoverable linearizability*, which achieves locality but may compromise program order following a crash. Whereas [2, 3, 14] consider an *individual process failures* model in which any subset of the processes may crash-fail at any time, Izraelevitz et al. [17] consider a *simultaneous failures* model, in which processes always crash-fail together. Under this model, persistent atomicity and recoverable linearizability are indistinguishable.

Several previous works investigated transactional access of persistent shared objects and NVRAM-based system recovery [4, 19, 20, 22, 21, 24, 25]. Coburn et al. [8] presented *NV-heaps*, a software user-level library that allows using NVRAM directly for storing object state and recovery data. Cohen et al. [9] recently presented an efficient logging protocol for NVRAM. Friedman et al. [11] presented three concurrent lock-free queue algorithms exhibiting different tradeoffs between consistency and efficiency.

References

[1] U. Abraham, S. Dolev, T. Herman, and I. Koll. Self-stabilizing l-exclusion. *Theor. Comput. Sci.*, 266(1-2):653–692, 2001.

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

[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] H.-J. Boehm and D. R. Chakrabarti. Persistence programming models for non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM, pages 55–67, 2016.

[5] P. Bohannon, D. F. Lieuwen, and A. Silberschatz. Recovering scalable spin locks. In *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing, SPDP 1996, New Orleans, Louisiana, USA, October 23-26, 1996.*, pages 314–322, 1996.

[6] P. Bohannon, D. F. Lieuwen, A. Silberschatz, S. Sudarshan, and J. Gava. Recoverable user-level mutual exclusion. In *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing, SPDP 1995, San Antonio, Texas , USA, October 25-28, 1995*, pages 293–301, 1995.

[7] D. R. Chakrabarti, H. Boehm, and K. Bhandari. Atlas: leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 433–452, 2014.

[8] 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.

[9] N. Cohen, M. Friedman, and J. R. Larus. Efficient logging in non-volatile memory by exploiting coherency protocols. *PACMPL*, 1(OOPSLA):67:1–67:24, 2017.

[10] M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[11] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank. Brief announcement: A persistent lock-free queue for non-volatile memory. In *Distributed Computing - 31st International Symposium (DISC)*, 2017.

[12] W. M. Golab and D. Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing PODC*, pages 211–220, 2017.

[13] 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.

[14] 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.

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

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

[17] 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.

[18] P. Jayanti and A. Joshi. Recoverable FCFS mutual exclusion with wait-free recovery. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, pages 30:1–30:15, 2017.

[19] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 399–411, 2016.

[20] V. J. Marathe, M. Seltzer, S. Byan, and T. Harris. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2017.

[21] D. Narayanan and O. Hodson. Whole-system persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 401–410, 2012.

[22] D. Schwalb, M. Dreseler, M. Uflacker, and H. Plattner. Nvc-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics, IMDM@VLDB 2015, Kohala Coast, HI, USA, August 31, 2015*, pages 4:1–4:8, 2015.

[23] G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[24] 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.

[25] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 91–104, 2011.