NVRAM - response recover linearzability *

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 3, 2018

Abstract

abstract goes here...

*Partially supported by the Israel Science Foundation (grants 1227/10, 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 now prevalent anywhere from high-end server machines, through desktop and laptop computers to smartphones, accelerating the shift to concurrent, multi-threaded software. Concurrent software involves a collection of threads, each running a separate piece of code, possibly on a different core. Since threads may be delayed because of a variety of reasons (such as cache-misses, interrupts, page-faults, and scheduler preemption), shared-memory multiprocessors are *asynchronous* in nature.

Asynchrony is also related to reliability, since asynchronous algorithms that provide nonblocking progress properties (e.g., lock-freedom and wait-freedom) in an asynchronous environment with reliable processes continue to provide the same progress properties when *crash failures* are introduced. Informally speaking, this property holds because a process that crashes permanently at an arbitrary point in the execution of its 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 non-blocking shared memory 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. The reason is that the crash-recovery model is poorly matched to multi-core architectures with volatile SRAM-based caches and DRAM-based main memories: Any state stored in main memory is lost entirely in the event of a system crash or power loss, and recording recovery information in non-volatile secondary storage (e.g., on a hard disk drive or solid state drive) imposes overheads that are unacceptable for performance-critical tasks, such as synchronizing threads inside the operating system kernel.

This separation between volatile main memory and non-volatile secondary storage has led to a partitioning of the program state into operational data stored using in-memory data structures, and recovery data stored using sequential on-disk structures such as transaction logs. In the event of a system-wide failure, such as a power outage, in-memory data structures are lost and must be reconstructed entirely from the recovery data, making the software system that relies on them temporarily unavailable. As a result, the design of in-memory data structures emphasizes disposable constructs optimized for parallelism, in contrast to the structures that hold recovery data, which cannot be discarded upon a failure and which benefit less from parallelism since their performance is limited by the secondary storage.

Recent developments in non-volatile main memory (NVRAM) media foreshadow the eventual convergence of primary and secondary storage into a single layer in the memory hierarchy, combining the performance benefits of conventional main memory with the durability of secondary storage. Traditional log-based recovery techniques can be applied correctly in such systems but fail to take full advantage of the parallelism enabled by allowing processing cores to access recovery data directly using memory operations rather than slow block transfers from secondary storage.

In this paper we investigate how the performance benefits of NVRAM can be harnessed for improving the robustness of shared-memory programs by allowing concurrent algorithms to efficiently recover from crash failures. This challenge requires a careful rethinking of recovery mechanisms and involves addressing both foundational and algorithmic research questions. Our emphasis would be on direct, non-transactional, approaches of implementing algorithms for *recoverable concurrent* (also called *persistent*) objects. More specifically, we formulate an abstract model of NVRAM shared-memory multiprocessors, and discuss known correctness and progress conditions, while proposing a new property called *recoverable-response linearizability*. We then discuss the theoretic limitations

of these systems, on the one hand, and construction of concurrent algorithms that support effective recovery from crash failures, on the other hand. We focus on construction of recoverable versions of primitive memory operations such as read, write, swap, compare-and-swap and fetch-and-add, deriving upper and lower bounds, and how these recoverable versions can be used to implement various applications.

1.1 Related work

When an application implements a concurrent data structure, it is necessary to specify its *semantics*, namely, the properties provided by values it returns, which determine the feasibility and complexity of implementing it. The *correctness* condition of a concurrent data structure specifies how to derive the semantics of concurrent implementations of the data structure from the corresponding *sequential specification* of the data structure. This requires to disambiguate the expected results of concurrent operations on the data structure.

Two common ways to do so are *sequential consistency* [12] and *linearizability* [9]. Both require that the values returned by the operations appear to have been returned by a sequential execution of the same operations; sequential consistency only requires this order to be consistent with the order in which each individual thread invokes the operations, while linearizability further requires this order to be consistent with the real-time order of non-overlapping operations. The standard shared-memory model assumes that shared memory is linearizable or at least sequentially consistent.

Linearizability is ill-equipped to specify the correctness criteria for implementations that support crash-recovery failures, since linearizability has no notion of an aborted or failed operation, and requires that a process finish one operation before it invokes the next. Extended versions of linearizability or, more generally, alternative definitions are required for specifying correctness for such implementations.

Aguilera and Frølund [1] proposed *strict linearizability* as a correctness condition for persistent concurrent objects, which treats the crash of a process as a response, either successful or unsuccessful, to the interrupted operation. A successful response means that the operation takes effect at some point between its invocation and the crash failure, and an unsuccessful response means that the operation does not take effect at all. Strict linearizability preserves both *locality* [8] and program order. Although this property seems like a natural extension for linearizability, Aguilera and Frølund proved that unlike linearizability it precludes the implementation of some wait-free objects for certain machine models: there is no wait-free implementations of multi-reader single-writer (MRSW) registers from single-reader single-writer (SRSW) registers under strict linearizability.

Guerraoui and Levy [7] proposed *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 fail to provide locality: correct histories of separate objects, when merged, will not necessarily yield a correct composite history.

Guerraoui and Levy properties capture, in some sense, the flexibility of implementations which uses helping mechanism, where an interrupted operation by some failed process p can be completed by a different process after p crashes. Hence, we would like to have the option to set the linearization point after the crash of p, where strict linearizability does not allow so. Moreover, Censor-Hillel et al. [3] formalised the notion of helping, and proved that some objects, when implemented wait-free,

must use an helping mechanism.

In addition, it is not clear how to generalize transient atomicity to a general object which does not support a write operation, as the definition explicitly uses write operation in order to determine the allowed interval for the linearization point. Moreover, transient atomicity allows a scenario where a process p crash during an operation on object X, then recovers and complete a read operation on X, while the linearization point of the interrupted operation is set to after the read. That is, we get a history where p executes two overlapping operations on the same object, causing a program order inversion, and violets the well-formedness property (defined in Section...), which is in the heart of the linearizability definition.

In order to overcome the above drawbacks Berryhill et al. [2] proposed an alternative condition, called *recoverable linearizability*, which achieves locality but may sacrifice program order after a crash. It is a relaxed version of persistent atomicity, which requires the operation to be linearized or aborted before any subsequent linearization by the pending thread on that same object.

Izraelevitz et al. [10] considered a real-world failure model, in which processes are assumed to fail together, as part of a full-system crash. Under this model, persistent atomicity and recoverable linearizability are indistinguishable (and thus local). The term *durable linearizability* was used to refer to this merged consistency condition under the restricted failure model.

The goal of all correctness conditions defined above for the crash-recovery model is to maintain the state of concurrent objects consistent in the face of crash failures. However, to the best of our knowledge, none of them guarantees that the recovery code is always able to infer whether the failed operation completed and, if so, to obtain its response value. Inferring this may be challenging, since function responses are returned via volatile processor registers.

As a simple example, consider a base object $\mathcal{B}$ that supports the *read* and *write* operations. Suppose that a process p crashes while performing a $\mathcal{B}.write(v)$ operation, which, upon completion, should returns *ack*. Since the write is atomic, there is no problem in ensuring the consistency of $\mathcal{B}$ in spite of p's failure: either the write took effect before the failure, or it did not. However, in this case, once the process recovers from the failed *write* operation it has no way of knowing whether the write occurred or not. This is because p cannot tell whether the failure occurred before v was written to NVRAM or whether it occurred after v was written to NVRAM (and was possibly then overwritten) but the *ack* value, written to a volatile local variable, was lost because of the failure. Without this knowledge, if the failed *write* operation on $\mathcal{B}$ was applied by an operation of another recoverable object $\mathcal{O}$, $\mathcal{O}$ may not be able to recover correctly.

To address this problem, we consider the model proposed by Golab and Ramaraju for investigating mutual exclusion [6] and later used by [5, 11], which is the following. A set of asynchronous processes communicate by accessing atomic *non-volatile shared-memory variables*. In addition, each process has *volatile private variables* stored in processor registers. *Processes are unreliable* in the following sense: at any point in time, while executing a concurrent object function, a process p may incur a crash failure, causing all its local variables to be reset to arbitrary values.

A concurrent object is called *recoverable*, if it implements a `Recover` function that is responsible for fixing its internal state following a failure and allows it to either correctly complete its failed operation and obtain its response value, or re-try it, so it may proceed to perform its next operation. If the failure occurs while p executes an operation of a recoverable object, then, upon resurrecting p, the application or operating system guarantees that p's execution proceeds by executing the object's `Recover` function. It is assumed that the `Recover` function has access to the value of p's program counter at the time of the failure. *We note that knowledge of the last PC value leaves*

uncertainty regarding whether that last instruction was performed or not and, if it was, what its return value was.

We propose a new correctness property called *recoverable response linearizability* (RR-linearizability). Informally, RR-linearizability allows the recovery code to 'extend' the interval of the failed operation until the end of the recovery code. It requires a transactional effect guaranteeing that either the operation is linearized at some point between its invocation and the end of recovery code (which may attempt to complete it), or the operation has no effect. In addition, unlike previous definitions, RR-linearizability requires also that, following recovery, process p is able to determine whether the operation was linearized or not and, if it was, what its response value is.

2 Model and Definitions

2.1 Standard Shared-Memory Model

We use a standard model, based on Herlihy and Wing's model [9], of an asynchronous shared memory system. A set P of $n > 1$ *processes* $p_0, \ldots, p_{n-1}$ communicate by applying operations on shared *base objects* that support atomic operations, e.g., reads, writes and read-modify-write, to shared variables. No bound is assumed on the size of a shared variable (i.e., the number of distinct values it can take). Base objects are used in order to construct more complex implemented objects, such as queues and stacks, by defining access procedures that simulate each operation on the implemented object using operations on base objects.

The interaction of processes with implemented objects is modelled using steps and histories. There are four types of steps: (1) an invocation step, denoted (INV, p, X, op), represents the invocation by process p of an operation op on implemented object X; (2) a response step, denoted (RES, p, X, ret), represents the completion by process p of the last operation it invoked on object X, with response ret; (3) a crash step, denoted $(CRASH, p)$, represents the crash of a processes p;

A *history* H is a sequence of steps. Given a history H, we use $H|p$ to denote the subhistory of H containing all and only the events performed by process p. Similarly, $H|O$ denotes the subhistory of H containing all and only the events performed on object O, plus crash and recovery events. A response step is *matching* with respect to an invocation step s by a process p on object X in a history H if it is the first response step by p on X that follows s in H, and it occurs before p's next invocation (if any) in H.

Given a history H and a process p, an *operation* by p in H comprises an invocation step and its matching response, if it exists. An operation is *complete* if it has a matching response step, and *pending* otherwise. 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 has a matching response that 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.

A history H is *sequential* if no two operations in it are concurrent. Two histories H and H' are *equivalent* if for every process p, $H|p = H'|p$ holds. A history H is *well-formed* if for each process p, each invocation step in $H|p$ is immediately followed by a matching response, or by a crash step, and every response step in $H|p$ is a matching response for a preceded invocation. Informally, H is well-formed if $H|p$ is a sequential history of operations, except for the ones that may not have a response step due to crash steps.

An object O is defined using a *sequential specification* which defines its allowed behaviors and

is expressed as a set of possible sequential histories over O. A sequential history H is legal if for every implemented object O accessed in H, $H|O$ belongs to the sequential specification of O.

2.1.1 Correcntess Conditions

We now consider different variants of correctness conditions for NVRAM systems which takes into consideration failures. We follow Berryhill et al. [2] for formal definitions. For each variant, given a history H, we first define a way to extend H such that some pending operations are supplied with a matching response, and the rest pending operations are removed, followed by a definition containing requirements from the resulted extension.

All the following variants are in some sense a natural extension for the Herlihy and Wing's linearizability property, since it is widely used in conventional shared memory models. As such, we first give a formal definition for linearizability. However, linearizability does not support crash steps, and therefore the definition is valid for history H which is free of crash steps. Given such a history H, a *completion* of H is a history H' constructed from H by appending matching responses for a subset of pending operations, and then removing any remaining pending operations.

Definition 1 (Linearizability)**.** *A finite history H is linearizable if it has no crash events, and it has a completion H' and there exists 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 $op_1 <_S op_2$).

For strict linearizability, a *strict completion* of H is a history H' constructed from H by inserting matching responses for a subset of pending operations after the operations invocation and before the next crash step (if any), and finally removing any remaining pending operations and crash steps.

Definition 2 (Strict linearizability)**.** *A finite history H is strictly linearizable if it has a strict completion H' and there exists a legal sequential history S such that:*

SL1. H' is equivalent to S; and

SL2. $<_{H'} \subseteq <_S$ (i.e., if $op_1 <_{H'} op_2$ and both ops appear in S then $op_1 <_S op_2$).

For persistent linearizability, a *persistent completion* of H is a history H' constructed from H by inserting matching responses for a subset of pending operations after the operations invocation and before the next invocation step of the same process, and finally removing any remaining pending operations and crash steps.

Definition 3 (Persistent linearizability)**.** *A finite history H is persistently linearizable if it has a persistent completion H' and there exists a legal sequential history S such that both conditions SL1 and SL2 of definition 2 holds.*

For recoverable linearizability, a *recoverable completion* H' is obtained from H in exactly the same manner as a strict completion. In addition, the *invoked before* relation over a history H, denoted $\ll_H$ is an extension of the "happens before" relation, such that $op_1 \ll_H op_2$ if $op_1 <_H op_2$ or that both operations invoked by the same process p on the same object X, and the invocation step of op_1 precedes the invocation step of op_2 in H. Notice that the extension takes into account pending operations, while $<_H$ is not defined in such a case.

Definition 4 (Recoverable linearizability)**.** *A finite history H is recoverable linearizable if it has a recoverable completion H' and there exists a legal sequential history S such that:*

RL1. H' is equivalent to S; and

RL2. $\ll_H \subseteq <_S$ (i.e., if $op_1 \ll_H op_2$ and both ops appear in S then $op_1 <_S op_2$).

As shown by Berryhill et al. [2], the requirement for a strict completion H' does not prevent an operation from taking effect after a crash that interrupts it. This follows from the fact that unlike strict linearizability, we do not ask the sequential history S to respect the order $<_{H'}$, but rather the order $<_H$. For an operation *op* that was interrupt by a crash, in $<_H$ it is after any operation that was complete before the invocation of *op*, but there is no operation following *op* in $<_H$, since it has no response. Therefore, in S we allow to place *op* anywhere after its invocation without violating $<_H$. However, in order to prevent program order inversion of the same process on the same object, $\ll_H$ restrict S not place *op* in such a reverse order.

2.1.2 Recoverable Response Linearizability

As discussed in the introduction, none of the above definitions guarantee a failing process can complete its pending operation upon recovery, or at least have an access to the response value of the operation in case it is linearized. In some cases, a process might be able to know whether the operation took effect. Friedman et al. [4] used the term *detectable execution* for an implementation which satisfy this condition. Quoting from [4]: "Durable linearizability does not provide a mechanism to determine whether an operation that executed concurrently with a crash was eventually executed. Without the ability to distinguish completed operations from lost operations, it would be difficult to recover the entire program, because in practice it is often important to execute each operation exactly once."

A concurrent object is called *recoverable* if any of its operations is equipped with a `Recover` function, such that if a process crash while executing an operation on the recoverable object, upon recovering the appropriate `Recover` function is triggered (by the system), and we require the process to complete its pending operation before invoking the next one. As we explain later, the completion requirement, although seems too restrictive, does not rule out an option for the `Recover` function to abort the pending operation, as in such case the process can reissue it. Nevertheless, this restriction simplifies the definition and proofs.

Recoverable object by its own does not consider the response value of the operation. For example, a primitive CAS is a recoverable object (with an empty `Recover` function), although a process crashing after executing CAS have no access to the response value upon recovery, as it was lost. For this reason we extend the definition of recoverable object, such that in addition to a `Recover` function it also needs to satisfy the following: every operation returns (i.e., there is a response step in the history) only after the response value is persistent. In a more formal way, a process p have a designated variable Res_p in the non-volatile memory such that at the time of (RES, p, X, ret) step, the value ret is written in Res_p. Notice that the object's semantic does not change, that is, we do not require the response value to be persistent at the linearization point.

In order to formally capture this behaviour we introduce a recover step, denoted $(RECOVER, p)$, represents the recovery of a process p, and the invocation of the `Recovery` function matching the pending operation, if there is one. A history H is called *recoverable well-formed* if every crash step by a process p which is not the last step of p, is followed by a recover step of p (and no other

steps of p are allowed in between), and vice verse. In addition, every response step follows either the matching invocation step or a recover step. We care only about such histories, as we assume the system always triggers the proper `Recovery` function whenever a process is waking-up after a crash.

A recoverable object is in some sense "fail-resistant". If a process crash after completing an operation, then upon recovery the process have an access to the response value residing in the non-volatile memory. On the other hand, if a process crash before the response value is persistent, i.e., before the operation was completed, then upon recovery the `Recovery` function will complete the operation, together with making the response value persistent. To our knowledge, this is the first definition to consider the affect of a crash on the crashing process, and not only on the object.

One can think of different ways to persist the response value. For example, an operation to a recoverable object gets a location in the shared memory as an extra operand, and the response value is persistent in this location at the response step. This can be implemented easily by replacing Res_p with the supplied location. We do not role out such solutions. However, for ease of presentation the definition uses a simple version.

Notice that any object can be implemented in a recoverable manner, as long as there is no restriction on the correctness condition it requires to satisfy. The `Recover` function does not allow a process p to invoke a new operation before completing the pending operation, hence in every history H a process have at most a single pending operation which is his last invoked operation. Therefore a natural requirement for such an object is linearizability. Since every operation of a process needs to be complete (except for maybe the last one), and we use the original definition of linearizability, this implies that locality holds under this definition.

Definition 5 (Recoverable response linearizability)**.** *A finite history H is recoverable response linearizable if H is recoverable well-formed and after removing all crash and recover steps from H we get a well-formed linearizable history.*

The formal definition allows the use of recoverable objects only, as we require the history to be linearizable under the original Herlihy and Wing's definition. However, such objects can be used in a more general context. One may use recoverable objects only in critical parts of the program, as such an object guarantee the ability to recover and complete the operation in case of a crash. In the rest of the program, assuming data lost is less critical, a simpler objects can be used, e.g., implementations that only guarantee recoverable linearizability. In such a way, the programmer have the flexibility to protect certain parts of the program in the cost of time and space complexity by using the more powerful recoverable objects, while the rest of the program is more efficient but not completely robust to crashes.

3 Recoverable Base Objects

We consider a model in which the program counter (PC) is stored in the non-volatile memory. This can be done either explicitly in the program, or implicitly by the operating system. In this model, upon recovery the last PC is available, and the system knows during what operation the process crash, and thus the proper `Recover` function is invoked. As a result, there is uncertainty regarding whether the last instruction was performed or not. We also assume the `Recover` function has an access to the last PC before the crash, or in other words, it is aware of at what line the crash took place.

The following example clarify why do we need the definition of recoverable object to holds even for the case of primitives. Consider an object supporting a compare-and-swap (CAS) atomic operation. Assume a process p is executing an operation $res \leftarrow C.CAS(old, new)$ followed by a crash. There are several options for at what exact time the crash took place, and each case raises a different problem.

In the standard crash model, a primitive operation is atomic and takes effect instantly in the history, that is, the response follows the invocation in the history, where no other step by any other process is allowed in between. Under the same definition, and assuming the process crash just after completing the CAS operation and before advancing the PC, there was a response step in the history, and thus the RR-linearizability does not require the process to recover the operation. However, the operation is still pending in some sense, as upon recovery the process does not know whether it took affect or not, since the PC still points the same line and res content was erased.

Considering the response of the CAS operation to be at the time where the PC is being advanced solves the former problem. Nevertheless, what if the process crash just after the PC was changed? Again, the operation is not pending, so there is no need to recover it. In this case, upon recovery the process knows the operation was completed, since the PC no longer points to it. However, the process have no access to the response value that was stored in res, residing in the cache, and hence it may not be able to proceed its execution.

A recoverable version of CAS avoids the above problems. The operation considered to be complete only after both the CAS was linearized, and the response value is persistent in Res_p. Therefore, a crash after this point can cause no problem, as the process knows the operation was completed as well as have an access to the response value.

In the following section we present algorithms for implementing recoverable versions for well known primitives. A system equipped with such primitives can be used to implement any object in a recoverable way in the following manner: in case of a crash, the process will simply recover the last primitive operation along which the process crashed. Once the operation completes, the process can continue and execute the remaining code safely. Due to this observation we focus our attention to implementation of recoverable primitives.

The strongest progress property is bounded wait-free, that is, the number of steps a process takes when executing the recovery code in the absence of a failure is finite and bounded by a known constant (may be a function of n, the number of processes in the system), regardless of the other processes steps and the failures the process experienced so far. In addition, we would like the recovery code to use a finite number of variables. Nonetheless, this progress property can not always be achieved, and therefore in some cases we weaken our demand to deadlock-freedom, that is, a process is allowed to wait in the recovery code, and if all processes takes enough steps then eventually the process will finish its recovery code.

The recoverable object definition requires the response value to be persistent for any operation. However, this requirement is inefficient in case of a trivial response as ack. In this case there is no point to save the response value in the non-volatile memory, as the only information relevant for the process is whether the operation was complete or not. For such an operation we define the response step to be at the point where to PC no longer points the code implementing the operation, and we do not persist the response value. This observation can be used for any operation with a trivial response, even though the recoverable object definition is general and does not handle these cases separately.

In addition, the question of having a crash along a `Recover` function needs to be address. One

option is to assume a crash can not occur along executing a `Recover` function. In practice this might be reasonable to assume a short time of stability in the system following a crash. However, we wold like to have a stronger model, capable dealing with crash even along a `Recover` function. Notice also that a `Recover` function which uses only such recoverable primitives is itself recoverable.

In the following, we use the convention of capital letters names for shred variables and small letters for local variables. Also, a restart response of the `Recover` function implies the operation was not linearized and the process restart it (by moving back to the operation's first line). During the analysis we ignore the ABA problem, as this can be solved easily by augmenting any value with the writing process's id, and a sequential number (each process will have its own sequential number). This way we can guarantee no two identical values are used along any execution.

Read Following the formal definition of RR-linearizability, a process needs to write the value it reads to Res_p in order to implement a recoverable read. However, such an implementation is redundant and not efficient. In case the process crash before completing its read operation it can simply reissue it upon recovery.

In general, operations that only read the status of an object and does not change it (more formally, operation that can be commute with any other operation by a different process), usually uses only read primitive (e.g., snapshot), or at least implemented in a way such that if a process crash in the middle of an operation then reissue it does not affect the rest of the processes. In such cases, one can implement a recoverable version of the operation by having the `Recover` function reissuing the operation.

Write For write, we "wrap" the atomic write with a mechanism which allows the process to conclude whether its write or a different write took place since the invocation. For that, process p have a designated variable R_p in the non-volatile memory. The same variable can be used for all write operations of process p.

Algorithm 2 R.write(val) by process p

```
1: procedure WRITE
2:     temp ← R
3:     R_p ← temp
4:     R ← val
5: procedure RECOVER
6:     if pc < 4 then return restart
7:     if R_p == R then return restart
8:     return ack
```

For simplicity, we write the recovery code as a single instruction, although it needs to be written as several instructions, as it accesses two different locations in the shared memory. Since R_p is designated to p only, the point where p reads R determines the `Recover` function outcome.

The intuition for correctness is the following. If there was a write to R between the two reads of p (at line 2, and at the `Recover` function), then either this write is by p, and we can linearize it at the point where it took affect, or that there was a write by some other process, and we can linearize the write of p just before it. Hence, the real write "overwrite" p's write, and the rest of

the processes can not distinguish between the two scenarios. Thus, in case of a failure before line 4 the operation will be aborted.

Claim 1. *The recoverable write implementation given in 1 is RR-linearizable.*

Proof. First notice that the `Recover` function only reads, thus repeating it does not effect the other processes. That is, if a process crash along the `Recover` function then restarting the function is indistinguishable from executing it once. Hence, the proof can ignore crashes along the `Recover` function. Also, notice that there is no waiting or locks used in the code, hence if a process given enough time (with no crash) then eventually it will finish the write operation (including the `Recover` function if needed). That is, the implementation is bounded wait-free.

Consider a run α, and let p be a process executing operation $write(val)$ to variable R. If p does not crash along the operation, then obviously p write to R once in line 4, and this is also the linearization point. Assume there is a crash along the operation. If the crash occurs before line 4 then p did not wrote to R, and upon recovery it will restart the write operation. If the crash was in line 4 then p wrote to R_p the value it reads in line 2. There are two scenarios.

Assume $R_p == R$, i.e., there was no write to R between the time when p reads R in line 2, and in line 7. In particular, p did not wrote to R in line 4, since we assume all writes are distinct (and avoiding the ABA problem), and therefore it is safe to restart the write operation, as no process was affected by p's operation so far. Assume now $R_p = R$, i.e., there was a write to R between the time when p read R in line 2, and in line 7. If p wrote to R in line 4, then we can linearize the operation at this point. Otherwise, there is a write by a different process q. Linearizing p's operation just before q operation, causing q to overwrite the value of p, generates a history which is indistinguishable to all process from the history where p does not write at all. In any of these cases, there is a linearization point to the operation which respects the sequential specification of the object. □

Compare-and-Swap A Compare-and-Swap (CAS) object supports the $cas(old, new)$ operation, which atomically compares the value stored in the object to old, and if they are equal, sets the value to be new. The operation returns the compression's result, that is, if the operation succeeded or failed. CAS object also support a read operation which returns the value stored in the object.

The main idea is to have the CAS stored also the if of a process who last committed a successful CAS. Thus, a process can inform a different process if its CAS operation was successful. A process p first reads the CAS object. If it observes a value different then *old* it returns false. In such case, the operation is linearized at the linearization point of the read. Otherwise, p informs the last process who committed a successful CAS that his operation was completed by writing to a designated location in the non-volatile memory. This way, if p crash after a successful CAS operation, the next process which change the value of the CAS, first needs to inform p his CAS was successful. Therefore, upon recovery p can identify if its CAS took affect by seeing his value in C, or that some other process informed him about a success.

Following the code, a process crash before line 7 will restart its operation upon recovering, while a crash at line 7-8 (and before completing the operation) will depend on whether the CAS was successful. A key point in this implementation is that a fail CAS operation can be reissued anyway, since it does not effect other processes and thus reissuing it does not violate the sequential specification of the object.

Claim 2. *The recoverable CAS implementation given in 3 is RR-linearizable.*

Algorithm 4 C.cas(old,new) by process p

Shared variables:

- C: compare-and-swap object, init $< null, null >$
- R[N][N]: two dimensions array, init $null$

```
1:  procedure CAS
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:      Res_p ← ret
9:      return ret
10: procedure RECOVER
11:     Read C, R[p][∗]
12:     if < p, new > appears in C, or new appears in R[p][∗] then
13:         Res_p ← true
14:         return true
15:     else
16:         return restart
```

Proof. First notice that the implementation is bounded wait-free, as there is no use with locks or waiting. Also, the CAS object is composed from two fields, an id of the last process to successfully write to C and a value. A key point is that linearizing a failed CAS operation does not effect the rest of the processes, that is, removing such an operation is indistinguishable to other processes.

A process p first reads the content of C. If the value in C is different then old, then the process return false, and we can linearize the operation at the point of the read. Otherwise, p will try to swap the content of C. Before that, it announce the last process which wrote to C that his write was successful, by writing in $R[id][p]$ the value it saw. We use a two dimensional array so that processes will not overwrite each other in this part. Now the process can try and replace the value of C with its own new value, while taking into consideration the id field of the CAS.

□

Test-and-Set A Test-and-Set (TAS) object initially stores the value 0, and supports the t&s operation which atomically change the value of the object to 1 and returns the previous value. A TAS object can be easily implemented using a single CAS object. However, we would like to support an implementation which only uses TAS and read/write primitives in case a process does not crash. It is possible to prove that given only TAS and read/write registers one can not implement a recoverable TAS which is also bounded wait-free, thus the `Recover` function will use waiting mechanism.

A process p tries to win the TAS object, and the winning process writes its id into a designated variable $Winner$. Once a process writes to $Winner$ any process can recover from a crash by simply

read $Winner$. Hence, the main difficulty is to make sure only one process writes to $Winner$, and this is the only process to return 0.

At the beginning of the `Recovery` function, process p first announce it is recovering by writing 2 into $R[p]$. Then, in order to make sure any process running from this point on will return 1 (in case the crash occurred before the t&s operation took effect) p executes another t&s operation on T. As discussed above, if $Winner$ was written to then p can know if it is the process to win T by comparing its id to the one in T. However, if $Winner$ was not written to, all crashed processes needs to agree on the winner. For that, we use a CAS object for simplicity, although any recoverable leader election algorithm can be used here (for example, one that only uses reads and writes). Notice that there is no need to use recoverable CAS here, since in a crash along the `Recovery` function, the process can simply restart the function.

Once p wins the CAS, it is the only crashed process to win, and therefore it is the candidate to win the TAS object. However, it might be that some other process already won the TAS, but yet to write to $Winner$. To overcome this scenario, p waits for every active process to either finish its t&s operation, or to start the `Recovery` function. We know that any process who will run starting from this point will lose the TAS object, and the CAS object (in case of a crash). Thus, if after the for loop no process wrote to $Winner$ then p can safely announce itself as the winner. Otherwise, p lost to some other active process, and this process wrote to $Winner$ since it won the TAS object in line 3.

4 Discussion

Algorithm 6 T.t&s() by process p

Shared variables:

- T: Test-and-Set object, init 0
- C: Compare-and-Swap object, init *null*
- R[N]: an array, init 0
- Winner: a register, init *null*

```
1:  procedure T&S
2:      R[p] ← 1
3:      ret ← T.t&s()
4:      if ret = 0 then
5:          Winner ← p
6:      Res_p ← ret
7:      R[p] ← 0
8:      return ret
9:  procedure RECOVER
10:     R[p] ← 2
11:     T.t&s()
12:     if Winner ≠ null then
13:         ret ← (Winner == p)
14:         Res_p = ret
15:         return ret
16:     C.cas(null, p)
17:     if C ≠ p then
18:         Res_p ← 1
19:         return 1
20:     for i from 0 to N do
21:         await(R[p] ≠ 1)
22:     if Winner = null then
23:         Winner ← p
24:     ret ← (Winner == p)
25:     Res_p = ret
26:     return ret
```

References

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

[2] 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.

[3] K. Censor-Hillel, E. Petrank, and S. Timnat. Help! In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 241–250, 2015.

[4] M. Friedman, M. Herlihy, V. J. Marathe, and E. Petrank. Brief announcement: A persistent lock-free queue for non-volatile memory. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, pages 50:1–50:4, 2017.

[5] 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.

[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] P. Jayanti and A. Joshi. Recoverable FCFS mutual exclusion with wait-free recovery. In *Distributed Computing - 31st International Symposium (DISC)*, 2017.

[12] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Computers*, 46(7):779–782, 1997.

A Formal Proofs

A *one-time* mutual exclusion algorithm is a ME algorithm in which each process completes at most a single passage. Since our proofs consider executions in which each process is allowed to complete a passage at most once, our results apply to one-time mutual exclusion algorithms.