Brief Announcement: Recoverable and Detectable Self-Implementations of Swap

Tomer Lev Lehman
Department of Computer Science, Ben Gurion University, Beer Sheva, Israel

Hagit Attiya
Department of Computer Science, Technion, Haifa, Israel

Danny Hendler
Department of Computer Science, Ben Gurion University, Beer Sheva, Israel

Abstract

Recoverable algorithms tolerate failures and recoveries of processes by using non-volatile memory. Of particular interest are *self-implementations* of key operations, in which a recoverable operation is implemented from its non-recoverable counterpart (in addition to reads and writes).

This paper presents two self-implementations of the SWAP operation. One works in the *system-wide failures* model, where all processes fail and recover together, and the other in the *independent failures* model, where each process crashes and recovers independently of the other processes.

Both algorithms are wait-free in crash-free executions, but their recovery code is blocking. We prove that this is inherent for the independent failures model. The impossibility result is proved for implementations of *distinguishable* operations using *interfering* functions, and in particular, it applies to a recoverable self-implementation of swap.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms

Keywords and phrases Persistent memory, non-volatile memory, recoverable objects, detectablitly

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.44

Related Version *Full Version*: `https://arxiv.org/abs/2308.03485`

Funding Partially supported by the Israel Science Foundation (grants 380/18 and 22/1425).
Tomer Lev Lehman: Partially supported by the Lynne and William Frankel Center for Computer Science.

1 Introduction

Recent years have seen a rising interest in the failure-recovery model for concurrent computing. This model captures an unstable system, where processes may crash and recover, and it has two variants: In the *system-wide failure* model (also called the *global-crash* model), all processes fail simultaneously and a single process is responsible for the recovery of the whole system. In the *independent failures* model (also called the *individual-crash* model), each process can incur a crash independently of other processes and recovers independently. *Recoverable* algorithms, tolerating failures and recoveries, have been presented for various concurrent data structures, for both the system-wide model [5, 7, 9, 14, 18, 20, 21] and the independent-failure model [1, 3, 5, 18, 20].

The correctness of a recoverable algorithm can be specified in several ways. *Durable Linearizability* [16] intuitively requires linearizability of all operations that survive the crashes. *Detectability* [9] ensures that upon recovery, it is possible to infer whether the failed operation took effect or not and, in the former case, obtain its response. *Nesting-safe Recoverable Linearizability* (NRL) [1], defined for the independent failures model, ensures detectability and linearizability. It also allows the nesting of recoverable objects. By providing implementations of NRL primitive objects, a programmer can combine several of these primitives to create recoverable implementations of complex higher-level objects and algorithms. This level of abstraction can be helpful in the adoption of recoverable algorithms.

To facilitate high-level implementations of complex NRL objects it is helpful to introduce implementations of low-level base NRL objects. An attractive approach to designing low-level based NRL objects is through *self-implementations* [20], in which a recoverable operation is implemented with instances of the same primitive operation, possibly with additional reads and writes on shared variables. This approach ensures that when using the recoverable version of an operation, the system must only support its hardware-implemented counterpart.

NRL self-implementations already exist for various primitives, including *read*, *write*, *test&set*, and *compare&swap* [1,3], as well as *fetch&add* [20]. A universal construction [3] using NRL read, write and *compare&swap* objects builds upon previously-introduced self-implementations of NRL objects to take any concurrent program with read, write and CAS, and make it recoverable while adding only constant computational overhead.

This paper presents the first NRL self-implementations of *swap*, for both the system-wide and the independent failures models. Swap is a widely-used primitive that is employed by concurrent algorithms. Our implementations borrow ideas from the *recoverable mutual exclusion (RME)* [12] algorithms of [11,17], which use a similar approach to overcome swap failures. Unlike these algorithms, however, our implementations are also challenged with the task of satisfying wait-freedom and linearizability.

Both our algorithms are wait-free in crash-free executions, while the recovery code in both is blocking.

We present an impossibility proof for implementing a class of *distinguishable operations* using a set of *interfering functions* [13] in a recoverable *lock-free* fashion in the independent failures model. In particular, this result applies to self-implementations of swap, but it also holds for, e.g., implementing swap using fetch-and-add and swap combined. Other distinguishable operations to which this proof applies are the *deque* of a queue object and the *pop* of a stack object. Our impossibility result unifies and extends specific results for self-implementations of *test&set* [1] and *fetch&add* [20]. Another related impossibility result addresses recoverable consensus in the independent failures model [10].

Several previous papers introduce general transformations to port existing algorithms and make them persistent, e.g., [3,4,6,8,15]. Most of these transformations use strong primitives such as *compare&swap* while their non-recoverable counterparts may use weaker primitives, in terms of their consensus number [13]. We believe future research may use our self-implementation of swap to extend general constructions such as [3] mentioned above to programs that also use swap as a primitive.

Similarly to NRL, *detectable sequence specifications* (DSS), introduced by Li and Golab [18], formalizes the notion of detectability. The DSS-based approach is more portable and less reliant on system assumptions in comparison to NRL, but delegates the responsibility for nesting to application code.

2 Model, In Brief

We use a simplified version of the NRL system model [1]. There are n asynchronous *processes* $p_1, \ldots, p_n$, which communicate by applying atomic *primitive* read, write and read-modify-write operations to *base objects*. The state of each process consists of non-volatile *shared variables*, which serve as base objects, as well as volatile *local variables*. A *crash-failure* (or simply a *crash*) can occur at any point during the execution. A crash resets all local variables to arbitrary values but preserves the values of all non-volatile variables.

A process p *invokes an operation Op* on an object by performing an *invocation step*. *Op completes* by executing a *response step*, in which *the response of OP is stored to a local volatile variable of* p. It follows that the response value is lost if p incurs a crash before *persisting* it, that is, before writing it to a non-volatile variable.

In the *independent failures* model a *recoverable operation Op* is associated with a *recovery procedure Op*.RECOVER that is responsible for completing *Op* upon recovery from a crash. Following a crash of process p that occurs when p has a pending recoverable operation op_p, the system eventually resurrects p by invoking the recovery procedure of the failed op_p.

As proven by [2], detectable algorithms for the NRL model must keep an auxiliary state that is provided from outside the operation, either via operation arguments or via a non-volatile variable accessible by it. We assume that *Op*.RECOVER has access to a designated per-process non-volatile variable storing the sequence number of *Op* which is incremented before each operation invocation.

For the *system-wide failures* model in which all processes crash simultaneously, the system recovers by executing a parameterless *global recovery procedure* called *Op*.GRECOVER. Once *Op*.GRECOVER completes, the system resurrects each of the failing processes for performing an *individual recovery procedure for Op*, called *Op*.RECOVER.

3 Detectable Swap Algorithms

A swap object supports the *SWAP(val)* operation, which atomically swaps the object's current value *cur* to *val* and returns *cur*. A key challenge to overcome when implementing a detectable swap object from read, write, and primitive swap operations is that the return values of one or more primitive swap operations may be lost upon a system-wide failure that occurs before the operations are persisted. These non-persisted operations may have already affected the state of the swap object and, moreover, operations by other processes may have already returned the values written by these primitive operations. To ensure linearizability, the implementation must identify such operations and handle them correctly.

The return value of each SWAP operation must be the input of another SWAP operation (or the initial value of the swap object). Furthermore, the operand swapped in by one SWAP operation can be returned by at most a single other SWAP operation.

To ensure linearizability, the real-time order between non-overlapping operations must be preserved, as illustrated in Figure 1. This scenario involves six processes, $p_1, \ldots p_6$, performing eight SWAP operations, $op_0, \ldots op_7$. A system-wide crash occurs when operations op_0, op_2, op_4, op_6 have already been completed (hence their return values are specified), while operations op_1, op_4, op_5, op_7 are pending. Note that operations op_1, op_4, op_5, although not completed, have surely affected the global state of the swap object as their inputs are the return values of other operations, while op_7 (pending as well) might or might not have affected the object's state.

There are several ways the system may recover in order to produce a correct linearizable result. In all of them, op_1 must return 0. The remaining operations might return different values in the following ways: (1) op_4 returns 2, op_5 returns 3, and op_7 returns 6. (2) op_7 returns 2, op_4 returns 7, and op_5 returns 3. (3) op_4 returns 2, op_7 returns 3, and op_5 returns 7. There are several possible linearizations in this example, because op_7 may be linearized in several ways since its effect on the global state is unknown.

We represent the order of SWAP operations as a linked list of Node structures, the end of which is pointed to by a *tail* variable manipulated with primitive swaps. The list starts with a sentinel node called *headNode*, which holds the object's initial value (denoted $\perp$).

Each Node structure represents a single SWAP operation and stores a pointer *prev* to the node of its predecessor operation and the SWAP's operand *val*. The order of SWAP operations is reflected by the order of the Node structures in the list. By doing so, each Node points to the previous Node structure that represents the previous SWAP operation, hence, the operation's return value will be *Node.prev.val*.

A problem occurs if a process successfully swaps its Node into the list but fails before pointing from its structure to the previous Node. This type of failure may create *fragments* in the list representing the SWAP operations. Thus, instead of a single complete list, crashes may result in several incomplete disconnected lists. In order to reconnect these fragments back to a complete list, our algorithm goes over all previously-announced operations upon recovery and recreate a correctly-ordered complete list of operations.

A similar challenge occurs in the RME algorithms of Golab and Hendler [11] and Jayanti et al. [17]. These algorithms mend fragments of the linked-list based queue, used in the MCS lock [19], which are created when failures occur just before or after primitive swap operations.

Our algorithm needs to address two additional challenges, however. First, the SWAP operations of our algorithm should be wait-free, whereas RME implementations are allowed to block. Second, unlike the RME implementations, our algorithm should provide linearizability. Specifically, the order of list fragments, constructed during recovery, must respect the real-time order between SWAP operations.

We address these challenges by employing a *fragment ordering* scheme, which we view as the key algorithmic novelty of our algorithms. The scheme encapsulates the critical steps of each SWAP operation by two vector timestamp computations. Based on the resulting timestamps, the recovery code ensures the following invariant: if a fragment A contains a Node n_A that was created after an operation associated with a Node n_B on fragment B was completed, then fragment B will be ordered after fragment A in the connected list. (Note that *prev* pointers define the *reverse order* between operations.)

Figure 2 presents a set of fragments that may be generated immediately after the system-wide crash ending the execution depicted in Figure 1. As specified, when ordering fragments, the algorithm uses vector timestamps for maintaining linearizability. As an example, consider a linked list, reconnecting the fragments of Figure 2, in which $op_4.prev \leftarrow op_0$, $op_1.prev \leftarrow op_3$, $op_7.prev \leftarrow op_2$ and $op_5.prev \leftarrow op_7$. Although this list contains the Nodes of all operations from tail to head, it violates linearizability because op_3 is ordered after op_2 although it follows it in real-time order. By using the two vector timestamps, our algorithm is able to order the fragments so that linearizability is maintained.

The full version presents the details of the algorithm and its correctness proof, proving the next theorem:

▶ **Theorem 1.** *There is an algorithm that implements a recoverable NRL SWAP in the system-wide failures model using only read, write and primitive Swap operations. The SWAP operations are wait-free.*

For the independent failures model, a recoverable algorithm must allow one or more processes to execute its recovery code concurrently, while other processes may concurrently execute their SWAP operations. In order to handle this concurrency correctly, we introduce two key changes to the system-wide failures algorithm. First, the recovery procedure now

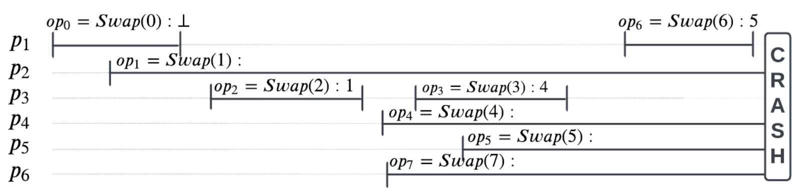

Figure 1 An example of the effect of a system-wide failure.

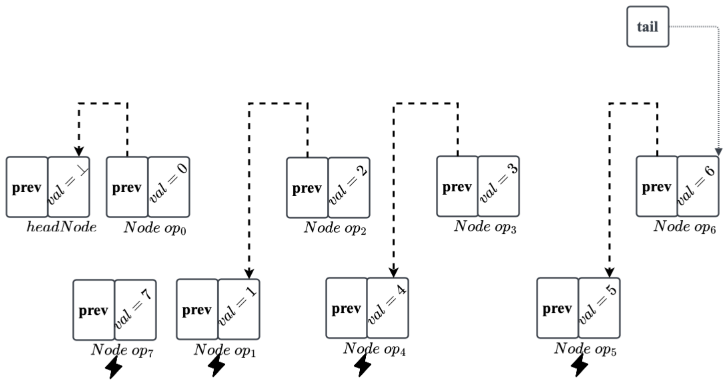

Figure 2 List fragments existing after the execution described in Figure 1, immediately after a system-wide crash. Operations 1,4,5 crashed after swapping their Nodes to *tail* but before persisting their pointer to their predecessor Node.

synchronizes concurrent invocations by using a starvation-free RME lock, effectively serializing the execution of the recovery code. We employ the RME lock of Golab and Ramaraju [12], which uses only reads and writes. The second change allows the recovery code to wait for a concurrent SWAP operation *Op* to either complete or crash. Only once this happens, can the recovery code continue. The full version presents the details of the algorithm and its correctness proof, proving the following theorem:

▶ **Theorem 2.** *There is an algorithm that implements a recoverable NRL SWAP in the independent failures model using only read, write and primitive Swap operations. The SWAP operations are wait-free.*

4 Impossibility of lock-freedom for the independent failures model

We prove a theorem establishing the impossibility of implementing lock-free algorithms for a wide variety of recoverable objects under the independent failures model. This result generalizes previous results [1, 20], to a wider family of operations and implementations.

The result applies to *distinguishable operations*: Informally, an operation *Op* is distinguishable if it can be invoked with two operands, $x \neq y$, such that the return values of these invocations allows the system to determine which operation was applied first. The proof applies when implementations use only read, write, and a set of *interfering functions* [13], which are functions that either commute or overwrite. (See the full version.)

▶ **Theorem 3.** *There is no recoverable implementation of a distinguishable operation M from read, write, and interfering primitive operations in the independent failures model, such that both M and M.RECOVER are lock-free.*

Consider a swap object with initial value 0; when SWAP(1) and SWAP(2) are applied sequentially, only the first operation applied returns 0. This shows that SWAP is a distinguishable operation. Note also that a primitive swap is overwriting, since applying it twice overwrites the first application. Thus, the theorem implies that there is no recoverable self-implementation of SWAP, where both SWAP and SWAP.RECOVER are lock-free. This shows that the mutual exclusion lock in our algorithm for the individual failures model cannot be avoided.

It is also possible to show that the pop operation of a stack object, and the deque operation of a queue object, as well as *fetch&add* and *test&set*, are distinguishable.

5 Discussion

We present two NRL self-implementations of the swap object, one for the system-wide failures model and the other for the independent failures model. In both, SWAP operations are wait-free and the recovery code is blocking. In the system-wide failures model, this is a result of delegating the recovery to a single process, while in the independent failures model, it is due to coordination between the recovering process and the other processes. We also prove the impossibility of a lock-free implementation of distinguishable operations using read-write and a set of interfering functions, in the independent failures model. In particular, this shows that with independent failures, a self-implementation of swap cannot be lock-free.

Our algorithms use $O(m * n)$ space, where m is the number of SWAP invocations in the execution. Bounding memory consumption to $O(n)$ is relatively easy if a recoverable swap operation by one process can wait for operations by other processes to either make progress or fail. An interesting open question is to figure out whether the space complexity of detectable swap self-implementations with wait-free operations can be reduced to $o(m)$ or if $\Omega(m)$ is inherently required. We leave this question for future work.

References

1 Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *ACM Symposium on Principles of Distributed Computing*, pages 7–16, 2018.

2 Ohad Ben-Baruch, Danny Hendler, and Matan Rusanovsky. Upper and lower bounds on the space complexity of detectable objects. In *39th ACM Symposium on Principles of Distributed Computing*, pages 11–20, 2020.

3 Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–264, 2019.

4 Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *19th International Conference on Principles of Distributed Systems (OPODIS)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

5 Kyeongmin Cho, Seungmin Jeon, and Jeehoon Kang. Practical detectability for persistent lock-free data structures. *arXiv preprint arXiv:2203.07621*, 2022.

6 Andreia Correia, Pascal Felber, and Pedro Ramalhete. Persistent memory and the rise of universal constructions. In *15th European Conference on Computer Systems*, pages 1–15, 2020.

7 Panagiota Fatourou, Nikolaos D Kallimanis, and Eleftherios Kosmas. The performance power of software combining in persistence. In *27th ACM Symposium on Principles and Practice of Parallel Programming*, pages 337–352, 2022.

8 Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E Blelloch, and Erez Petrank. NVTraverse: In NVRAM data structures, the destination is more important than the journey. In *41st ACM Conference on Programming Language Design and Implementation*, pages 377–392, 2020.

9 Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. *ACM SIGPLAN Notices*, 53(1):28–40, 2018.

10 Wojciech Golab. The recoverable consensus hierarchy. In *32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 281–291, 2020.

11 Wojciech Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *ACM Symposium on Principles of Distributed Computing*, pages 211–220, 2017.

12 Wojciech Golab and Aditya Ramaraju. Recoverable mutual exclusion. In *2016 ACM Symposium on Principles of Distributed Computing*, pages 65–74, 2016.

13 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

14 Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 429–442, 2021.

15 Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. *ACM SIGARCH Computer Architecture News*, 44(2):427–442, 2016.

16 Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *30th International Symposium on Distributed Computing*, pages 313–327. Springer, 2016.

17 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. A recoverable mutex algorithm with sub-logarithmic RMR on both cc and dsm. In *ACM Symposium on Principles of Distributed Computing*, pages 177–186, 2019.

18 Nan Li and Wojciech Golab. Detectable sequential specifications for recoverable shared objects. In *35th International Symposium on Distributed Computing (DISC)*, 2021.

19 John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

20 Liad Nahum, Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Recoverable and detectable fetch&add. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

21 Matan Rusanovsky, Hagit Attiya, Ohad Ben-Baruch, Tom Gerby, Danny Hendler, and Pedro Ramalhete. Flat-combining-based persistent data structures for non-volatile memory. In *23rd International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS*, pages 505–509. Springer, 2021.