RMR-Efficient Detectable Objects for Persistent Memory and Their Applications

Sahil Dhoked
Department of Computer Science, The University of Texas at Dallas, TX, USA

Ahmed Fahmy
Department of Electrical and Computer Engineering, University of Waterloo, Canada

Wojciech Golab
Department of Electrical and Computer Engineering, University of Waterloo, Canada

Neeraj Mittal
Department of Computer Science, The University of Texas at Dallas, TX, USA

Abstract

We describe a novel construction of arbitrary read-modify-write (RMW) primitives in a persistent shared memory model with process failures. Our construction uses blocking synchronization, in the form of recoverable mutual exclusion (RME), and is optimal in terms of the widely studied remote memory reference (RMR) complexity measure. The implemented objects tolerate either system-wide or independent process crashes, depending on the RME lock used, and also provide detectability for resolving the outcome of operations interrupted by failures. We prove that our construction is RMR-optimal using a reduction back to the RME problem. Our proof technique introduces a novel algorithmic style that enables solving challenging synchronization problems using a common execution path for both the system-wide and independent failure models, which previously required separate analyses, and relies only on a suitable implementation of the detectable base objects in each model to achieve RMR efficiency. Experiments demonstrate that our construction outperforms prior wait-free and lock-free algorithms on a multiprocessor with Intel Optane persistent memory.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms

Keywords and phrases persistent memory, synchronization, recoverability, fault tolerance, detectability, scalability, RMR complexity, theory, mutual exclusion

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2024.5

Funding *Wojciech Golab*: Supported by an Ontario Early Researcher Award, a Google Faculty Research Award, and the Natural Sciences and Engineering Research Council (NSERC) of Canada.

Acknowledgements We are grateful to the anonymous reviewers for their valuable feedback.

1 Introduction

Over the last decade, the emergence of persistent main memory has reinvigorated research on shared memory algorithms. The ability to retain data across power failures and system crashes opens the door to faster recovery of in-memory data structures than is possible using the traditional approach of rebuilding such structures using state saved in slower secondary storage, and has precipitated a thorough reexamination of established theoretical abstractions. The persistent memory revolution has so far witnessed several reformulations of classic correctness properties and synchronization problems (*e.g.*, [7, 31, 3, 24, 20]) into their recoverable counterparts, as well as the birth of more novel concepts such as Friedman, Herlihy, Marathe and Petrank's notion of *detectability* [18, 19] – the capability to forensically determine the outcome of an operation that was interrupted by a failure.

The blossoming landscape of problems and solutions hints at a bright future for algorithmic research on persistent main memory, though the full perspective remains difficult to comprehend given limited connections between emerging concepts. This research takes

important steps towards filling this gap by exploring the relationship between detectability, which we formalize using *sequential specifications* [6, 37, 39], and the *recoverable mutual exclusion* (RME) problem [24, 25], which is a modern take on the long-standing question of how to implement fault-tolerant mutex locks. In a nutshell, RME generalizes Dijkstra's mutual exclusion (ME) problem [16] by allowing processes to crash at arbitrary times and recover by starting over. While it makes sense intuitively that objects of arbitrary types can be constructed using RME locks, existing literature focuses on recovering the internal state of the lock and overlooks the fundamental problem of recovering actions performed in the critical section. At best, compositions of RME locks have been considered where such actions are largely idempotent, and can be recovered by repeating the critical section. The use of detectable objects to solve RME is similarly unexplored, and we view this as a missed opportunity since the treatment of such problems from first principles is notoriously challenging (*e.g.*, see [22, 34]). Advances along new research directions that simplify reasoning about failures and concurrency are urgently needed as the theory community takes aim at more complex synchronization problems in the realm of persistent memory.

Implementing detectable objects correctly and using them efficiently is itself technically challenging. Fundamentally, such object implementations face the problem that modern memory hierarchies combine persistent media, such as Intel Optane persistent memory, with volatile CPU registers that are used by primitive memory operations to return responses to the application program. In other words, it is not possible for a program to simply "look up" the outcome of its last memory operation before a failure, or even determine exactly which operation was the last one. Despite this, detectable objects must track the progress of operations across failures so that any process can resolve the status of its most recent operation, even if the corresponding state transition has already been overwritten by an operation applied by another process. Further perils lie ahead when detectable objects are used to solve more complex synchronization problems because such objects achieve only a partial separation of concerns between fault tolerance and concurrency control. To begin with, a recoverable algorithm must decide which of its possibly many detectable base objects need to be resolved, and in which order. Additional recovery state that is kept outside of the base objects (*e.g.,* via checkpoint variables) may also be needed to guide recovery towards a consistent state. On the other hand, once the detectable objects are properly implemented and techniques for their correct use are developed, a new algorithmic style emerges where application-specific imperatives, such as enforcing linearizability [29] or mutual exclusion [16], can be insulated from mundane low-level recovery actions such as determining the outcome of a read-modify-write (RMW) hardware instruction after a failure.

Our main contributions in this paper are two-fold. First (Section 4), we present a universal construction of detectable RMW primitives. Our RME-based construction is the first to implement arbitrary RMW types while using a sublinear number of *remote memory references* or RMRs – expensive memory operations that traditionally define the time complexity of ME and RME algorithms – while ensuring strict linearizability [1]. We show experimentally that it performs well in comparison to prior lock-free and wait-free techniques. Second (Section 5), we leverage the detectable RMW primitive to devise a novel RMR-optimal RME algorithm, and deduce from the properties of this algorithm a lower bound on the worst-case RMR complexity of any generic construction of detectable RMW primitives. The unique algorithmic style emerging from this reduction opens the door to solving complex problems using a common execution path for both the system-wide and independent failure models, which previously required separate analyses, and relies only on a suitable implementation of the detectable base objects to achieve RMR efficiency in each failure model.

2 Model

We combine Herlihy and Wing's classic model of linearizable data structures [27] with features borrowed from recent research on crash recovery for persistent memory [7, 31, 25, 23, 3, 37]. A fixed set of N asynchronous *processes*, denoted $p_1, p_2, \ldots, p_N$, communicate by applying operations on shared *base objects*. Processes may fail by crashing, either independently or simultaneously, and may recover from crashes by restarting their execution from the beginning under the same process ID. Each base object has a *sequential specification* that describes its possible states and state transitions. We will assume that base object operations are atomic with the understanding that they can be either supported directly in hardware, or emulated in software via shared object implementations that specify an access procedure for each emulated operation. Base objects are *persistent*, meaning that their states are unaffected by crash failures. In addition, processes may use private *volatile* variables whose values are reinitialized during recovery from a failure. The algorithms presented in this paper can be adapted to more complex models that consider a volatile cache through judicious insertion of persistence instructions, for example using the transformations described in [31, 13].

Executions of algorithms are modeled using *histories*, which are sequences of *system states* and *steps*. A system state is the combination of values assigned to all program variables, shared and private, including program counters. There are three types of steps: a *base object step* represents the execution of one shared memory operation by a process along with bounded local computation (*e.g.*, arithmetic and accessing private volatile variables); an *invocation* or *response* step represents the invocation of an operation on an implemented object or its response, respectively; and a crash step represents the crash failure of one or more processes. Crashes can be *independent* (affecting one process) or *system-wide* (affecting all processes simultaneously). We consider two types of algorithms in this paper: shared object implementations (Section 4) modeled as a collection of access procedures for different implemented operations, and recoverable mutual exclusion (RME) [24] (Section 5) modeled as a special infinite loop. Histories of RME-like algorithms comprise only base object and crash steps. Histories of implementations additionally contain invocation and response steps.

Histories of an algorithm must satisfy safety and liveness properties specific to that algorithm. For RME, where processes repeatedly execute a *non-critical section (NCS)*, *recovery section (RS)*, *entry section (ES)*, *critical section (CS)*, and *exit section (XS)*, the fundamental safety property is *mutual exclusion*: at most one process can be in the CS at a given time. A bounded prefix of the ES called the *doorway* controls the order of entry into the CS in *first come first serve* (FCFS) RME locks. Liveness means that executions of the RS, ES, and XS eventually terminate, and is predicated on two assumptions: (1) the CS is bounded, and (2) the history is *fair*, meaning that if a process leaves the NCS, it continues taking steps until it completes the XS and returns to the NCS without crashing. The latter implies that if a process crashes during a *passage*, which is a sequence of consecutive executions of the RS, ES, CS, and XS, then it eventually begins another passage. A maximal sequence of passages by one process where only the last one ends in a complete execution of the XS is called a *super-passage*. For shared object implementations, we choose *strict linearizability* [1] as the main correctness property: implemented operations appear to take effect instantaneously between their invocation and response steps, and any operation interrupted by a crash failure must take effect prior to the failure or not at all. Liveness means that every implemented operation eventually returns a response, and is once again predicated on the history being fair, meaning that if a process invokes an operation then it must continue taking steps until the operation produces a response or the process crashes.

Some shared object implementations are *detectable* [18, 19], meaning that a process can resolve precisely the outcome of an operation that was interrupted by a crash. We formalize detectability using sequential specifications, following the trend established in [6, 37, 39, 32], whereby detection is accomplished by invoking a special *resolution operation* (called **resolve**) on the implemented object. Specifically, we adopt the *unified sequential specification* (UDSS) introduced by Moridi et al. [39], whereby **resolve** returns a triple $\langle op, tag, res \rangle$ representing the most recent detectable operation op invoked by the calling process p_i that took effect, or else $\langle \perp, \perp, \perp \rangle$ if no such operation exists. Here res is the response of op, and tag is an auxiliary argument passed to op to disambiguate successive invocations of op that may return the same response (or no response at all).

We quantify complexity in terms of space, measured by counting the number of persistent base objects, and time, measured by counting *remote memory references* (RMRs) – expensive shared memory operations that necessitate communication on the interconnect that joins processors with memory. In the *cache-coherent* (CC) multiprocessor architecture, RMRs are closely tied to cache misses and invalidations. For analyzing upper bounds, we assume that every shared memory operation is an RMR unless a process reads an object that it has previously read, and which has not been overwritten since the most recent such read. In the *distributed shared memory* (DSM) model, each variable is statically defined as local to exactly one process and remote to all others, and an RMR is defined as any access to a remote variable. In special cases, algorithms or portions thereof may be *wait-free* [28], meaning that they terminate after a bounded number of steps in all executions.

3 Overview of Results

In this research, we derive both upper and lower bounds on the worst-case RMR complexity per operation of detectable RMW objects (more precisely on objects that simultaneously support **FAS** and **CAS**) by connecting this topic formally to RME. In terms of upper bounds, our detectable objects have the same RMR complexity asymptotically as their underlying RME locks, and require only $O(N)$ additional space. As we discuss in Section 4.2, experiments conducted using Intel Optane persistent memory show that our objects scale better than alternatives obtained using lock-free and wait-free techniques [6, 36] for Fetch-And-Store operations. Our lower bound on RMR complexity pertains to the independent failure model, and shows that implementing detectable objects is as hard as solving the RME problem. We obtain this result by constructing a reduction from the latter problem to the former, and applying existing lower bounds on the RMR complexity of RME [9, 10]. The lower bound is tight, which makes our detectable object construction RMR-optimal for independent failures. The same holds for system-wide failures, where we achieve $O(1)$ RMR complexity. As summarized in Tables 1 and 2, we present the first sub-linear (i.e., $o(N)$) RMR bounds for detectable unconditional RMW operations such as Fetch-and-Store.

4 From RME to Detectable Objects

This section presents a generic RMR-efficient construction of detectable read-modify-write (RMW) primitives using an RME lock as the main building block. This result complements prior work (see Section 6) on detectable constructions of comparison primitives that achieve RMR-efficiency indirectly by guaranteeing wait-freedom. As we show later on in Section 5, detectable **FAS** and **CAS** primitives instantiated using our construction can be used to solve the RME problem efficiently and relatively straightforwardly, which establishes equivalence in terms of worst-case RMR complexity (per passage and per operation, respectively) between RME and any generic detectable RMW construction.

Table 1 A comparison of known algorithms for implementing detectable/recoverable RMW objects *under system-wide process failures*. Unless stated otherwise, the object is readable and the RMR complexity bound holds for both the CC and DSM models. M denotes the number of operations that have been invoked on the object.

Reference	Object	RMR Complexity		Linearizability (Safety) Guarantee	Progress Guarantee	
		Detectable Operation	Recover or Resolve		Detectable Operation	Recover or Resolve
Lehamn, Attiya, Hendler [36]	Fetch-And-Store (swap)	$O(N)$	$O(M)$	nesting-safe recoverable	wait-free	centralized
Herlihy [28, 7]	Fetch-And-ϕ	$O(N)$	$O(N)$	strict	wait-free	wait-free
Ben-David, Blelloch, Friedman, Wei [6]	Compare-And-Swap	$O(1)$	$O(1)$	strict	wait-free	wait-free
	Fetch-And-ϕ from Compare-And-Swap	unbounded	$O(1)$	strict	lock-free	wait-free
This work	Fetch-And-ϕ	$O(1)$	$O(1)$	strict	blocking	wait-free†

† Assumes an RME lock that supports a wait-free withdraw or abort feature.

Table 2 A comparison of known algorithms for implementing detectable/recoverable RMW objects *under independent process failures*. Unless stated otherwise, the object is readable and the RMR complexity bound holds for both the CC and DSM models. M denotes the number of operations that have been invoked on the object.

Reference	Object	RMR Complexity		Linearizability (Safety) Guarantee	Progress Guarantee	
		Detectable Operation	Recover or Resolve		Detectable Operation	Recover or Resolve
Attiya, Ben-Baruch, Hendler [3]	write	$O(1)$	$O(1)$	nesting-safe recoverable	wait-free	wait-free
	Compare-And-Swap	$O(1)$	$O(N)$	nesting-safe recoverable	wait-free	wait-free
	non-readable Test-And-Set	$O(1)$	$O(N)^*$	nesting-safe recoverable	wait-free	blocking
	counter	$O(1)$	$O(N)$	nesting-safe recoverable	wait-free	wait-free
Lehamn, Attiya, Hendler [36]	Fetch-And-Store (swap)	$O(N)$	$O(M)^*$	nesting-safe recoverable	wait-free	blocking
Jayanti, Jayanti, Jayanti [32]	writable Compare-And-Swap	$O(1)$	$O(1)$	recoverable	wait-free	wait-free
Herlihy [28, 7]	Fetch-And-ϕ	$O(N)$	$O(N)$	recoverable	wait-free	wait-free
Ben-David, Blelloch, Friedman, Wei [6]	Compare-And-Swap	$O(1)$	$O(1)$	strict	wait-free	wait-free
	Fetch-And-ϕ from Compare-And-Swap	unbounded	$O(1)$	strict	lock-free	wait-free
This work	Fetch-And-ϕ	$O\left(\frac{\log N}{\log \log N}\right)$	$O\left(\frac{\log N}{\log \log N}\right)$	strict	blocking	wait-free†

* For CC model only. The RMR complexity is unbounded for the DSM model.

† Assumes an RME lock that supports a wait-free withdraw or abort feature.

4.1 Detailed description and analysis

The sequential specification of the implemented object in our construction is a subset of Moridi et al.'s UDSS [39] applied to a generic Fetch-And-ϕ (**FA**ϕ) operation that returns the previous abstract state of the object and applies a state transition according to the user-specified function ϕ. In formal terms, we can regard ϕ as a parameter of the implemented operation, which enables the application of different state transitions at different points in the same execution. For example, Fetch-And-Increment (**FAI**) can be emulated as $\phi : s \to s + 1$, and a Fetch-And-Store (**FAS**) operation that swaps in value v can be emulated as $\phi_v : s \to v$. Compare-And-Swap (**CAS**) and blind **Write** operations are also easily emulated. The UDSS transformation maps **FA**ϕ to **detectable-FA**ϕ, which accepts an additional *tag* argument, and a **resolve** operation that returns a triple of the form $\langle op, tag, res \rangle$ as explained in Section 2. In a nutshell, the generic construction protects executions of **detectable-FA**ϕ using a recoverable mutex lock RM. Dealing with crash failures in this construction is nontrivial for two reasons. First, as we explain shortly, the critical section established by RM comprises multiple memory operations and requires cleanup after a crash. Second, a process that crashes during a passage through RM is required (by the specification of the RME problem) to restart and complete a failure-free passage. We address both concerns by adding recovery actions in the **resolve** operation, and by ensuring that this operation is executed after a failure via the following access restriction:

▶ **Assumption 1.** *If a process fails inside* **detectable-FA**ϕ *then it must eventually complete a call to* **resolve***, and moreover it must do so before invoking any other operation on the same object.*

The construction records the abstract state of the implemented object in a shared variable S with state transitions occurring at line 18. S is initialized to s_0, the initial abstract value of the implemented object. Additional state needed for detectability is maintained using a collection of arrays. Process p_i uses $NextOp[i]$, $NextTag[i]$ and $NextRet[i]$ at lines 6 to 11 to record the signature of its next detectable operation (*i.e.*, function ϕ), the corresponding tag argument, and the value fetched from S, respectively. In case the operation is interrupted by a crash and needs to be rolled back for strict linearizability, p_i first creates a secondary copy of these state variables in $PrevOp[i]$, $PrevTag[i]$, and $PrevRet[i]$, respectively, at lines 2 to 4. The decision to roll back is informed by the value of the checkpoint variable $C[i]$, inspired by [25], which p_i uses to track its progress through the critical section at lines 1, 5, 17, and 20. An additional array $LastS$ is used to record the last value read from S at line 12, which simplifies recovery. Finally, we use a global variable P and an array $Help[1..N]$ to implement a helping mechanism that synchronizes the critical section in **detectable-FA**ϕ with **resolve**. The variable P records the ID of the process executing the critical section at line 15, and is used at lines 13 to 14 to pass the previous value of S to the last process that completed the CS up to and including line 15.

The operation **resolve** starts with a passage through RM at lines 23 to 25, which is required only to repair the internal state of RM. The remainder of **resolve** is structured as a case analysis based primarily on the value of $C[i]$. Its goal is to ensure that $NextOp[i]$, $NextTag[i]$ and $NextRet[i]$ record the correct details of p_i's last detectable operation that took effect. No special action is required if $C[i] = 0$ or $C[i] = 1$. The case $C[i] = 2$ (line 28) indicates that a detectable operation failed before it took effect but after its details were already recorded in $NextOp[i]$, $NextTag[i]$ and $NextRet[i]$. These state changes are rolled back at lines 29 to 31 to maintain strict linearizability. The case $C[i] = 3$ (line 33) indicates that a detectable operation failed close to the point where the abstract state S is updated at

line 18, and is the most challenging to resolve. Process p_i must deduce correctly whether its last operation took effect, even if other processes subsequently applied their own **detectable-FA**ϕ operations. The key algorithmic idea is for p_i to obtain a copy of the abstract state S as of the time of its failure, and compare it against the value p_i fetched from S and saved in $LastS[i]$ at line 12. The copy is obtained by reading $Help[i]$ at line 36 if $Help[i] \neq \perp$ (line 35), which indicates that another process already completed the helping mechanism at lines 13 to 14, otherwise by reading S directly line 34. If the copy and saved value match (line 37) then either p_i's last **detectable-FA**ϕ operation did not update S at line 18, or it did update S but caused a trivial state transition via a fixed point of the function ϕ. We consider that p_i's last **detectable-FA**ϕ operation did not take effect in both cases, and roll back p_i's state changes at lines 38 to 40 similarly to the case $C[i] = 2$. Otherwise, the copy and saved value are unequal, which implies that p_i did update S. In that case p_i's last **detectable-FA**ϕ operation did take effect and the correct values are already stored in $NextOp[i]$, $NextTag[i]$ and $NextRet[i]$. All cases in the analysis finally reset $C[i]$ to 0, if needed, and then produce the response $\langle NextOp[i], NextTag[i], NextRet[i]\rangle$ at line 42.

Shared variables:

- S: read/write register, initialized to the designated initial state s_0
- P: read/write register, initialized to $\perp$
- $C[1..N]$: array read/write registers initialized to 0
- $PrevOp[1..N]$, $PrevTag[1..N]$, $PrevRet[1..N]$, $NextOp[1..N]$, $NextTag[1..N]$, $NextRet[1..N]$, $Help[1..N]$, $LastS[1..N]$: arrays of read/write registers initialized to $\perp$
- RM: starvation-free RMR-optimal recoverable mutex

Procedure detectable-FAϕ(tag: integer) for process p_i.

```
1  C[i] := 1
2  PrevOp[i] := NextOp[i]
3  PrevTag[i] := NextTag[i]
4  PrevRet[i] := NextRet[i]
5  C[i] := 2
6  NextOp[i] := φ
7  NextTag[i] := tag
8  RM.Recover()
9  RM.Enter()
10 s := S
11 NextRet[i] := s
12 LastS[i] := s
13 if P ∉ {⊥, i} then
14     Help[P] := s
15 P := i
16 Help[i] := ⊥
17 C[i] := 3
18 S := φ(s)
19 RM.Exit()
20 C[i] := 0
21 return s
```

Procedure resolve() for process p_i.

```
22 if C[i] ∈ {2, 3} then
23     RM.Recover()
24     RM.Enter()
25     RM.Exit()
26 if C[i] = 1 then
27     C[i] := 0
28 else if C[i] = 2 then
29     NextOp[i] := PrevOp[i]
30     NextTag[i] := PrevTag[i]
31     NextRet[i] := PrevRet[i]
32     C[i] := 0
33 else if C[i] = 3 then
34     cmp := S
35     if Help[i] ≠ ⊥ then
36         cmp := Help[i]
37     if LastS[i] = cmp then
38         NextOp[i] := PrevOp[i]
39         NextTag[i] := PrevTag[i]
40         NextRet[i] := PrevRet[i]
41     C[i] := 0
42 return
   ⟨NextOp[i], NextTag[i], NextRet[i]⟩
```

Figure 1 Strictly linearizable detectable readable fetch-and-ϕ implementation.

The main correctness properties of the construction are stated in Theorem 1 and Theorem 2. Detailed proofs are included in Appendix A.

Table 3 Properties of our detectable object implementation with different RME locks. In the table, F denotes the number of recent failures and $\dot{c}$ denotes the point contention of the passage.

Failure Model	RME Lock	RMR Complexity	Desirable Properties
independent	Dhoked and Mittal [15]	$O\left(\min(\dot{c}, \sqrt{F+1}, \frac{\log N}{\log \log N})\right)$	adaptive to failures adaptive to point contention variant of FCFS
independent	Katzan and Morrison [35]	$O\left(\min(\dot{c}, \frac{\log N}{\log \log N})\right)$	wait-free resolve method adaptive to point contention
system-wide	Dhoked, Golab and Mittal [13]	$O(1)$	wait-free resolve method FCFS

▶ **Theorem 1.** *The generic construction is strictly linearizable.*

▶ **Theorem 2.** *Every operation applied to the generic construction terminates eventually in any infinite fair execution history. Furthermore, each operation incurs $O(R)$ RMRs where R denotes the worst-case RMR complexity of the recoverable mutex RM.*

The detectable construction can be optimized in several ways. To begin with, a wait-free **read** operation can be implemented by simply reading S directly, without acquiring the RME lock. Furthermore, lines 23 to 25 of **resolve** can be replaced with a wait-free withdrawal operation, if RM supports it (*e.g.*, [14]), which cleans up the state of the RME lock without acquiring the CS and makes the entire resolution procedure wait-free. (Wait-free resolution is important for avoiding deadlocks during recovery from crash failures.) Finally, we can dispense with the RME lock altogether and reduce the RMR (and step) complexity to $O(1)$ if only one process at a time executes the **detectable-FA**ϕ operation, which is relevant in scenarios where the construction emulates a single-writer multi-reader register with wait-free **read**. Additional ideas related to application of detectable base objects in FCFS-fair RME locks are discussed in Appendix B.

Note that, depending on which RME lock is used to protect the CS of **detectable-FA**ϕ, our implementation may satisfy additional desirable properties. We summarize these in Table 3 for some of the existing RME locks. Specifically, for the independent failure model, if Dhoked and Mittal's RME lock [15] is used, our implementation becomes adaptive to failures, and has only $O(1)$ RMR complexity as long as only a small number of failures occur.

4.2 Experimental Evaluation

We implemented our detectable construction in C++ using the Intel Persistent Memory Development Kit (PMDK) [44], and evaluated it on a 4-socket Intel Xeon Gold 6230 multiprocessor with Optane persistent memory. We compare against two alternatives: a lock-free emulation of **FAS** obtained using Ben-David, Blelloch, Friedman and Wei's wait-free detectable Compare-And-Swap (**CAS**) [6]; and a direct implementation of **FAS** using Lehman, Attiya and Hendler's partly wait-free detectable swap [36].[1] Our construction

[1] Lehman, Attiya and Hendler's construction uses a blocking recovery algorithm for independent failures, and can be considered fully wait-free in the system-wide failure model with centralized recovery.

internally uses Dhoked, Golab, and Mittal's withdrawable RME lock [14], which incurs $O(1)$ RMRs in the system-wide failure model. We consider two variations of the construction, one corresponding to the code in Figure 1 and the other leveraging critical section re-entry (CSR), which makes it possible to eliminate the helping mechanism in **detectable-FA**ϕ. The latter optimization sacrifices the possibility of wait-free resolution as it requires a critical section inside **resolve**.

Both our construction and the alternative algorithms were augmented with persistence barriers (*i.e.,* cache line write-backs and store fences) to control the durability and visibility of shared memory operations. The barriers were implemented by calling pmem_persist in the PMDK. The transformation of Izraelevitz, Mendes, and Scott [31] mandates placing such barriers after each memory operation, which is aggressive but appropriate for the lock-free and wait-free algorithms based on [6, 36]. However, this technique does not preserve the RMR complexity of busy-wait loops on our hardware platform, where persistence barriers always invalidate cache lines.[2] For this reason, we manually augmented the two implementations of our construction with barriers. Our manual technique relaxes Izraelevitz, Mendes, and Scott's recipe by using only one barrier per busy-wait loop. We also avoid barriers in the special case where a process reads a single-writer variable for which it is the exclusive writer. Additionally, we consider a more advanced persistence technique, implemented in the FliT library [47], that can effectively avoid redundant persistence operations while reading variables.

Each thread in the experiment repeatedly applies detectable **FAS** operations on a common shared object. Total throughput (operations/s) is measured at different levels of parallelism to evaluate scalability up to 160 threads: 4 processors with 20 hyperthreaded cores. Each point in our plots represents the average over a sample of three failure-free runs, each lasting 5s. Error bars (often too small to be seen) indicate one sample standard deviation. Runs with up to 20 threads collocate all the threads on a single processor for efficient synchronization, and larger experiments exhibit classic NUMA (Non-Uniform Memory Access) effects due to expensive cross-socket communication. Hyperthreading is used in runs with 81-160 threads.

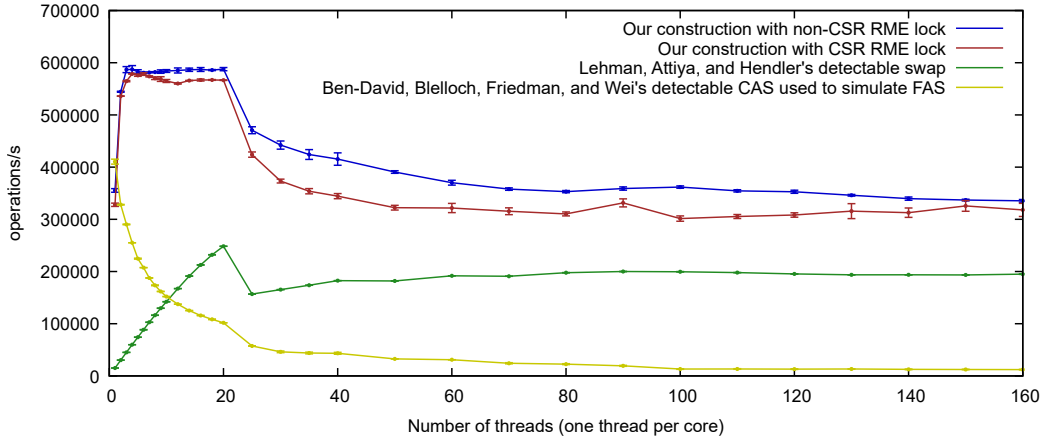

Figure 2 Scalability of non-optimized detectable Fetch-And-Store object implementations.

[2] Newer Intel Xeon processors offer a cache line writeback (CLWB) instruction that may either retain the cache line or invalidate it. Our processor model always invalidates the cache line [43].

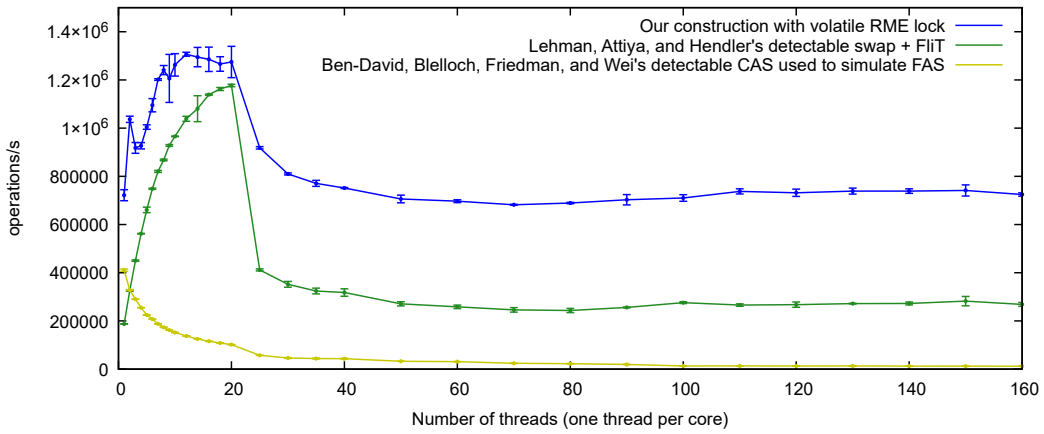

Figure 3 Scalability of detectable Fetch-And-Store object implementations with optimizations.

Figure 2 presents the scalability of implementations with aggressive persistence based on variations of [31] (i.e., without FliT). The results demonstrate that using a non-CSR RME lock in our construction with a helping mechanism is mostly superior to using a CSR RME lock without helping. Both alternatives substantially outperform the recoverable swap [36], which has bounded but linear RMR complexity in failure-free executions, and which we instantiated with $N = 160$. The lock-free implementation based on [6] exhibits the worst scalability, which is expected since FAS operations are contention-prone.

Next, we introduce centralized recovery (i.e., recovery in the main function of the benchmark program), which allows two optimizations. First, our construction can leverage an efficient failure detector-based RME lock [23] that mostly eliminates persistence barriers. Second, we can apply the hash table-based persistence mechanism provided by the FliT library [47] to minimize redundant persistence instructions for read-heavy shared variables. Figure 3 compares our construction with the optimized non-CSR lock against a FliT-optimized recoverable swap (FliT persistence applied to the *tail* variable and VTS array in [36]), and against the lock-free implementation based on [6].[3] The results, presented in Figure 2, show significant speed-ups for the optimized algorithms as compared to their counterparts in Figure 2. Our construction still leads in terms of throughput, especially at $\geq$25 threads where FliT's internal atomic counters suffer noticeably from NUMA effects.

5 From Detectable Objects to RME

This section presents a lower bound on the RMR complexity of detectable objects. We obtain the bound by devising a novel RME algorithm that uses detectable objects for synchronization, and then applying known lower bounds for the RMR complexity of the RME problem [9, 10]. Although the algorithm does not break new ground in terms of RMR and space complexity for the RME problem, its design embodies a novel algorithmic style that may be of independent interest: a common execution path is used for both the system-wide and independent failure models, which previously required separate designs and analyses. The generic algorithm is then instantiated for a particular failure model by substituting a suitable implementation of the detectable base objects, which determines the overall RMR complexity.

[3] The FliT library could not be applied to [6] due to missing support for double word atomic operations. Consequently, we use the same implementation of **CAS**-based **FAS** in Figure 3 as in Figure 2.

Our algorithm, presented in Figure 4 is an adaptation of Mellor-Crummey and Scott's (MCS) venerated queue lock [38], which represents attempts to enter the CS using a singly linked list of QNode structures. In this section we consider specifically the *context-free* version of MCS, in which nodes are allocated at line 58 of the entry section. Lines 61 to 71 follow the original MCS algorithm faithfully with the exception of the read-modify-write operations at line 62 and line 68, where we introduce detectability. The tail pointer variable T implements a subset[4] of Moridi et al.'s UDSS [39] applied to the combination of **FAS** and **CAS** operations: it supports detectable **FAS** and **CAS** operations that accept an auxiliary argument called a *tag* [39], in our case a QNode reference, and the resolution operation described earlier in Section 2. The latter is invoked at line 45 of the recovery section only if a process crashed between line 59 and line 72, where a reference to the current node is saved for recovery, and returns a triple $\langle op, tag, res \rangle$ as explained in Section 2. This triple encodes the name of the last operation as op, the value of the auxiliary argument as tag, and the response of the operation as res. The recovery section performs a case analysis on the values of these components, as we explain shortly, to decide the correct way to clean up the last passage.

A crash failure of process p_i occurring with $MyNode[i] = \perp$ (i.e., before line 59 or after line 72) is benign, and in this case the body of **Recover** is bypassed at line 44. More difficult cases occur if p_i crashes after threading its node behind the tail at line 62, and before it has passed control of the RME lock to the successor (i.e., completed line 71) or unthreaded its node from the queue (i.e., completed a successful swap at line 68). If p_i's most recent operation on the tail pointer T with respect to the current node (read at line 43) was **detectable-FAS** then p_i must complete its interrupted execution of the entry section. If p_i already linked with the predecessor, then it repeats the busy-wait loop in **Enter** and then completes **Exit** via line 50 and line 53. On the other hand, if there was a predecessor but the link was not established then p_i repeats **Enter** from line 64 and then completes **Exit** via line 52 and line 53. Finally, if there was no predecessor then p_i simply completes **Exit** via line 53. Next, there are two cases pertaining to a crash in **Exit**. If p_i successfully unthreaded its node from the queue then it proceeds to line 72 via line 55 and releases its node. Otherwise p_i has a predecessor p_j and repeats lines 70 to 72 of **Exit** via line 57 to link with p_j and pass control of the RME lock. Note that, in all of the above cases, p_i continues to execute the entry, critical and exit sections of the current passage, and no attempt is made to re-execute the CS from the recovery section. To guarantee critical section re-entry (CSR), the algorithm can be modified easily by having p_i bypass **Exit** at line 53 and then also bypass the body of **Enter** using an additional private variable to direct the flow of control.

The space complexity of the algorithm can be bounded by recycling the queue nodes allocated at line 58. We defer the details to Appendix C due to lack of space. Our technique reduces space to $O(N)$ while preserving the dynamic joining property: the algorithm allows processes to join an execution "on the fly" and does not assume prior knowledge of N.

The correctness properties of the algorithm are stated below in Theorem 3 under the implicit assumption that detectable operations on the tail pointer T are atomic. The doorway of the algorithm, for FCFS, is a bounded prefix of the entry section and comprises lines 58 to 62. If T is simulated using a software implementation then the **detectable-FAS** operation must be wait-free, at least until its linearization point, and we address this technicality more formally in Appendix B.

[4] The full UDSS would also preserve non-detectable **FAS** and **CAS** operations, which we do not need.

Define QNode: struct { *next*: **ref to** QNode, *locked*: **Boolean** }

Shared variables:

- T: **ref to** QNode, **init** $\perp$
- q_i: **ref to** QNode structure **init** $\langle \perp, \textbf{false} \rangle$ and local to process p_i in DSM model
- $MyNode[1..N]$: array of **ref to** QNode structure, each element **init** $\perp$

Procedure Recover( ) for process p_i.

```
43 q_i := MyNode[i]
44 if q_i ≠ ⊥ then
45     ⟨op, tag, res⟩ := T.resolve()
46     if tag = q_i then
47         if op = FAS then
48             prev := res
49             if prev ≠ ⊥ ∧ prev.next = q_i then
50                 execute line 66 of Enter
51             else if prev ≠ ⊥ ∧ prev.next = ⊥ ∧ q_i.locked = false then
52                 execute lines 64–66 of Enter
53             execute Exit
54         else if op = CAS ∧ res = true then
55             execute line 72 of Exit
56         else if op = CAS ∧ res = false then
57             execute lines 70–72 of Exit
```

Procedure Enter( ) for process p_i.

```
58 q_i := new QNode
59 MyNode[i] := q_i
60 q_i.locked := false
61 q_i.next := ⊥
   // insert QNode into linked list, also use q_i as tag
62 prev := T.detectable-FAS(q_i, q_i)
   // check for predecessor
63 if prev ≠ ⊥ then
       // link with predecessor
64     q_i.locked := true
65     prev.next := q_i
       // wait for predecessor to release lock
66     await q_i.locked ≠ true
```

Procedure Exit( ) for process p_i.

```
   // check for successor
67 if q_i.next = ⊥ then
       // no known successor, try to unlink from T,
          also use q_i as tag
68     if T.detectable-CAS(q_i, ⊥, q_i) = true then
           // no successor, lock idle
69         continue at line 72
       // wait for successor to link with p_i
70     await q_i.next ≠ ⊥
   // transfer lock ownership to successor
71 q_i.next.locked := false
72 MyNode[i] := ⊥
```

Figure 4 Recoverable version of Mellor-Crummey and Scott's queue lock for process p_i.

▶ **Theorem 3.** *The RME algorithm presented in Figure 4 satisfies the mutual exclusion, starvation freedom, and first-come-first-served (FCFS) properties in both the independent and system-wide crash failure models. Furthermore, it incurs $O(R)$ RMRs per passage in both the CC and DSM models where R denotes the worst-case RMR complexity of operations on the tail pointer variable T.*

The upper bound on RMR complexity stated in Theorem 3 can be used to deduce a lower bound on the RMR complexity of implementing detectable base objects in the independent crash failure model, as stated in Theorem 4.

▶ **Theorem 4.** *A strictly linearizable implementation of an object that supports detectable* **FAS** *and* **CAS** *operations from base objects that store $O(\log N)$ bits each and support arbitrary single-word read-modify-write operations has a worst case RMR complexity per operation of $\Omega(\log N / \log \log N)$ in both the CC and DSM models with independent crash failures.*

Proof. The algorithm presented in Figure 4 is, informally speaking, a reduction from the RME problem to the problem of implementing detectable base objects. Assuming that the memory management technique from Appendix C is applied, the algorithm requires $O(N)$ queue nodes and stores values of size $O(\log N)$ bits for node references. Supposing that the detectable base object T is replaced with a strictly linearizable implementation[5] that internally also uses base objects of size $O(\log N)$ bits, the algorithm solves the starvation-free RME problem and is subject to the lower bound of Chan, Giakkoupis and Woelfel [9] for deadlock-free RME under independent failures. The latter implies that the RME algorithm incurs $\Omega(\log N/\log\log N)$ per passage in the worst case, in both the CC and DSM models. Since portions of the algorithm outside of accesses to T incur only $O(1)$ RMRs per passage, this implies that in some execution a process incurs $\Omega(\log N/\log\log N)$ in total while accessing T in one passage. This implies that the implementation of T has a worst case RMR complexity of $\Omega(\log N/\log\log N)$ per operation since there are at most four such operations per passage: one in **Enter**, one in **Exit**, and two in **Recover**. ◀

The approach illustrated in Figure 4 can be contrasted against the algorithms of Golab and Hendler [22] as well as Jayanti, Jayanti and Joshi [34], which use much more elaborate recovery protocols to carefully mend the queue of nodes after a failure. A single detectable base object, namely the tail pointer, simplifies this task to the point where the correct recovery actions can be decided by a fairly simple case analysis, without first identifying and then analyzing a complex collection of disconnected queue fragments. Although the detectable base objects are themselves difficult to implement, which we prove formally in Theorem 4 for independent failures, a generic construction of such objects could be a powerful and reusable building block for the next generation of RME algorithms, including recoverable reader-writer and group mutual exclusion locks.

6 Related Work

The prospect of recovering in-memory data structures directly from persistent main memory after a crash failure opened the door to rethinking some of the fundamental assumptions ingrained in decades of literature on shared memory algorithms. We focus in this section on theoretical aspects of persistent memory, acknowledging that the subject caught the attention of practitioners (*e.g.*, in [46, 11, 41, 30, 42, 12]) before it captivated the theory community.

One of the earliest theoretical issues addressed was the formulation of a suitable correctness condition for concurrent objects in a crash-prone environment. Herlihy and Wing's celebrated linearizability property [29] has a fundamental drawback in that regard: it assumes that a process finishes one operation before it invokes another. This implies that a crashed process cannot in general restart execution using its old identifier; a new identifier must sometimes be chosen, which goes against the nearly universal modeling assumption that a system comprises a bounded set of N processes with fixed identifiers. Several alternatives have been proposed, the most widely-cited of which is the *durable linearizability* (DL) property of Izraelevitz, Mendes, and Scott [31]. DL simply disallows immediate reuse of process IDs, which has the side-effect of allowing an operation that is interrupted by a crash to take effect after the crash. Aguilera and Frølund's *strict linearizability* (SL) property [1] instead requires interrupted operations to take effect either before the crash or not at all. It allows for, but

[5] Strictly linearizable implementations can be used in place of atomic base objects while preserving all correctness properties relevant to this paper, aside from RMR complexity and space complexity.

does not require, the reuse of process IDs across failures. Thus, SL is the strongest and most portable of the relevant correctness properties, though not always attainable, as shown in [1] for certain register constructions. Weaker conditions proposed in [26, 7] are susceptible to undesirable anomalies (non-local behavior, program order inversion). Attiya, Ben-Baruch, and Hendler defined *nesting-safe recoverable linearizability* (NRL) [3] for recovery of nested compositions of objects. Their framework assumes that the system restarts a crashed process from the most deeply nested interrupted operation to complete it, which circumvents the anomalies present in [26, 7].

Traditional linearizability-like notions of correctness were challenged by Friedman, Herlihy, Marathe and Petrank [18, 19], who introduced the notion of *detectability* – the ability to resolve the outcome of an interrupted operation during recovery from a failure. NRL satisfies this goal indirectly by ensuring that every interrupted operation eventually takes effect, and variations on this theme have been adopted in several works [5, 2, 45, 36, 40]. Alternatively, detectability can be embedded into an object's sequential specification by introducing a special *resolution operation*, and then combined with an existing correctness condition. This approach was first realized by Ben-David, Blelloch, Friedman, and Wei [6], and later generalized by Li and Golab [37] as the *detectable sequential specification* (DSS), without relying on the strong system assumptions of NRL. Moridi, Wang, Cui, and Golab [39] examined producer-consumer synchronization as a concrete use case for detectable objects, and also proposed to merge the distinct *prepare* and *execute* phases of a detectable operation in the DSS by introducing the conceptually simpler *unified DSS* (UDSS). Jayanti, Jayanti, and Jayanti's [32] *method-based* approach to detectability is a blend of [3] and [7].

Recoverable (i.e., crash-tolerant) objects can be constructed by augmenting conventional techniques through the addition of specialized recovery procedures and, in some models, explicit persistence instructions that flush updates from the volatile cache to the persistent memory [7, 6, 17, 2, 32, 3, 45, 36, 40, 37, 24, 19, 8]. Of these, only purely lock-based or wait-free techniques [7, 24, 3, 6, 45, 34] ensure bounded RMR complexity, and only a handful yield RMR-efficient (particularly sub-linear RMR complexity in the number of processes) implementations. The latter generally fall into three categories: solutions based on Golab and Ramaraju's *recoverable mutual exclusion* (RME) [24, 25], for which RMR-optimal solutions were later presented in [33, 13, 22, 34], wait-free implementations of comparison primitives such as Compare-And-Swap (CAS) and Load-Linked/Store-Conditional (LL/SC) [3, 6, 32], and universal wait-free constructions [7]. RME does not prescribe a concrete technique for recovering actions performed in the critical section, and hence only partially addresses the implementation problem solved in this paper. On the other hand, detectable wait-free implementations of comparison primitives compete directly with our RME-based generic construction, but they have limited applicability in the context of proving RMR complexity bounds due to their weak symmetry breaking power. For example, the RMR-optimal algorithms in [33, 13, 22, 34] rely on unconditional synchronization primitives, such as Fetch-And-Store (FAS), to maintain queue structures. The wait-free CAS in [6] is strictly linearizable [1], but the constructions in [7, 3, 32] are not.

Tables 1 and 2 summarize the performance guarantees of the existing implementations of detectable object operations for system-wide failures (Table 1) and independent failures (Table 2). For strict linearizability, as mentioned earlier, the implementation provides a **resolve** procedure to detect if the operation interrupted by a failure has taken effect and perform any needed recovery. For recoverable linearizability, the implementation provides a **recover** procedure to enable an operation interrupted by a failure to be completed.

Tight bounds on worst-case RMR complexity in the context of persistent shared memory are known for the RME problem in both the CC and DSM models. For independent failures, Golab and Ramaraju's algorithm [24, 25] solves RME in $O(\log N)$ RMRs per passage using read/write registers only, and this matches Attiya, Hendler, and Woelfel's lower bound [4], which applies also to comparison primitives [21]. Using unconditional primitives such as FAS, in combination with CAS and read/write registers, the RME problem is solvable in $O(\log N/\log N \log N)$ RMRs per passage using Jayanti, Jayanti, and Joshi's algorithm [34], which uses the tree structure introduced by Golab and Hendler [22]. This matches the lower bound of Chan and Woelfel [10] for algorithms that use base objects of size $O(\log N)$ bits. Katzan and Morrison [35] showed that more efficient solutions are possible using wider base objects, and the trade-off achieved was proved to be tight by Chan, Giakkoupis, and Woelfel [9]. In the system-wide failure model, RMR bounds for algorithms that use reads, writes, and comparison primitives are the same as in the independent failure model. On the other hand, FAS in combination with CAS and read/write registers can be used to construct efficient queue locks with $O(1)$ RMRs complexity using Jayanti, Jayanti, and Joshi's algorithm [33] or Dhoked, Golab, and Mittal's [14]. Our transformation of ME algorithms to RME using detectable base objects proves for the first time that an RMR-optimal generic construction of detectable RMW objects has the same worst-case RMR complexity per operation as the RME problem per passage in the system-wide and independent failure models.

Our detectable object construction is an important step towards a general technique to transform a non-recoverable concurrent program into one that leverages persistent memory for recoverability. Several such transformation already exist. Ben-David, Blelloch, Friedman, and Wei's technique is based on segmenting the program into *capsules* that can be recovered using detectable operations [6]. Their technique is presented in the context of lock-free synchronization using Compare-And-Swap, but can be extended to support other detectable RMW operations.[6] Its main limitation is the assumption of a persistent call stack and page table, which requires a non-standard programming environment and operating system. Attiya, Ben-Baruch, Fatourou, Hendler, and Kosmas describe a general construction of recoverable data structures using persistent memory [2]. Their approach is specific to lock-free algorithms that use only read, write and CAS instructions.

7 Conclusion

Our work draws the first comprehensive connection between detectability and recoverable mutual exclusion. We showed how to use RME to construct detectable read-modify-write objects, and then used such objects to solve the RME problem using an algorithm that provides a common execution path for both system-wide and independent process failure models. Our constructions establish the first RMR lower bound on detectable object implementations, and pave the way toward future RMR-efficient solutions for complex synchronization problems such as recoverable versions of reader-writer exclusion and group mutual exclusion.

References

1 Marcos K. Aguilera and S. Frølund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, Hewlett-Packard Labs, 2003.

[6] Note that, since our implementation of the **detectable-FA**ϕ operation is blocking, the concurrent program generated by Ben-David, Blelloch, Friedman, and Wei's transformation when extended to use our generic **detectable-FA**ϕ operation will be safe but no longer lock-free.

2 Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. Tracking in order to recover - detectable recovery of lock-free data structures. In *Proc. of the 32nd ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 503–505, 2020. doi:10.1145/3350755.3400257.

3 Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *Proc. of the 37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 7–16, 2018. doi:10.1145/3212734.3212753.

4 Hagit Attiya, Danny Hendler, and Philipp Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In *Proc. of the 40th ACM Symposium on Theory of Computing (STOC)*, pages 217–226, 2008. doi:10.1145/1374376.1374410.

5 Ohad Ben-Baruch, Danny Hendler, and Matan Rusanovsky. Upper and lower bounds on the space complexity of detectable objects. In *Proc. of the 39th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 11–20, 2020. doi:10.1145/3382734.3405725.

6 Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *Proc. of the 31st ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 253–264, 2019. doi:10.1145/3323165.3323187.

7 Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *Proc. of the 19th International Conference on Principles of Distributed Systems (OPODIS)*, pages 20:1–20:17, 2016.

8 Trevor Brown and Hillel Avni. Phytm: Persistent hybrid transactional memory. *Proc. VLDB Endow.*, 10(4):409–420, 2016. doi:10.14778/3025111.3025122.

9 David Yu Cheng Chan, George Giakkoupis, and Philipp Woelfel. Word-size RMR tradeoffs for recoverable mutual exclusion. In *Proc. of the 42th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 79–89, 2023. doi:10.1145/3583668.3594597.

10 David Yu Cheng Chan and Philipp Woelfel. Tight lower bound for the RMR complexity of recoverable mutual exclusion. In *Proc. of the 40th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 533–543, 2021. doi:10.1145/3465084.3467938.

11 Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 105–118, 2011. doi:10.1145/1950365.1950380.

12 Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin C. Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proc. of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 133–146, 2009. doi:10.1145/1629575.1629589.

13 Sahil Dhoked, Wojciech Golab, and Neeraj Mittal. Modular recoverable mutual exclusion under system-wide failures. In *Proc. of the 37th International Symposium on Distributed Computing (DISC)*, pages 17:1–17:24, 2023. doi:10.4230/LIPICS.DISC.2023.17.

14 Sahil Dhoked, Wojciech Golab, and Neeraj Mittal. Modular recoverable mutual exclusion under system-wide failures. In *Proc. of the 37th International Symposium on Distributed Computing (DISC)*, pages 17:1–17:24, 2023. doi:10.4230/LIPICS.DISC.2023.17.

15 Sahil Dhoked and Neeraj Mittal. An adaptive approach to recoverable mutual exclusion. In *Proc. of the 39th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–10, New York, NY, USA, 2020. doi:10.1145/3382734.3405739.

16 Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965. doi:10.1145/365559.365617.

17 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 *Proc. of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 377–392, 2020. doi:10.1145/3385412.3386031.

18 Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. Brief announcement: A persistent lock-free queue for non-volatile memory. In *Proc. of the 31st International Symposium on Distributed Computing (DISC)*, volume 91, pages 50:1–50:4, 2017. doi:10.4230/LIPICS.DISC.2017.50.

19 Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *Proc. of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 28–40, 2018. doi:10.1145/3178487.3178490.

20 Wojciech Golab. The recoverable consensus hierarchy. In *Proc. of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 281–291, 2020. doi:10.1145/3350755.3400212.

21 Wojciech Golab, Vassos Hadzilacos, Danny Hendler, and Philipp Woelfel. RMR-efficient implementations of comparison primitives using read and write operations. *Distributed Computing*, 25(2):109–162, 2012. doi:10.1007/S00446-011-0150-8.

22 Wojciech Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *Proc. of the 36th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 211–220, 2017. doi:10.1145/3087801.3087819.

23 Wojciech Golab and Danny Hendler. Recoverable mutual exclusion under system-wide failures. In *Proc. of the 37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 17–26, 2018. doi:10.1145/3212734.3212755.

24 Wojciech Golab and Aditya Ramaraju. Recoverable mutual exclusion. In *Proc. of the 35th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 65–74, 2016.

25 Wojciech Golab and Aditya Ramaraju. Recoverable mutual exclusion. *Distributed Computing*, 32(6):535–564, 2019. doi:10.1007/S00446-019-00364-0.

26 Rachid Guerraoui and Ron R. Levy. Robust emulations of shared memory in a crash-recovery model. In *Proc. of the 24th International Conference on Distributed Computing Systems (ICDCS)*, pages 400–407, 2004. doi:10.1109/ICDCS.2004.1281605.

27 M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990. doi:10.1145/78969.78972.

28 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

29 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

30 Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *Proc. of the 21s International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 427–442, 2016. doi:10.1145/2872362.2872410.

31 Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Proc. of the 30th International Symposium on Distributed Computing (DISC)*, pages 313–327, 2016. doi:10.1007/978-3-662-53426-7_23.

32 Prasad Jayanti, Siddhartha Jayanti, and Sucharita Jayanti. Durable algorithms for writable LL/SC and CAS with dynamic joining. In *Proc. of the 37th International Symposium on Distributed Computing (DISC)*, pages 25:1–25:20, 2023. doi:10.4230/LIPICS.DISC.2023.25.

33 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. Constant RMR system-wide failure resilient durable locks with dynamic joining. In *Proc. of the 35th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 227–237, 2023. doi:10.1145/3558481.3591100.

34 Prasad Jayanti, Siddhartha V. Jayanti, and Anup Joshi. A recoverable mutex algorithm with sub-logarithmic RMR on both CC and DSM. In *Proc. of the 38th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 177–186, 2019. doi:10.1145/3293611.3331634.

35 Daniel Katzan and Adam Morrison. Recoverable, abortable, and adaptive mutual exclusion with sublogarithmic RMR complexity. In *Proc. of the 24th International Conference on Principles of Distributed Systems (OPODIS)*, pages 15:1–15:16, 2021.

36 Tomer Lev Lehman, Hagit Attiya, and Danny Hendler. Recoverable and detectable self-implementations of swap. In *Proc. of the 27th International Conference on Principles of Distributed Systems (OPODIS)*, pages 24:1–24:22, 2023. doi:10.4230/LIPICS.OPODIS.2023.24.

37 Nan Li and Wojciech Golab. Detectable sequential specifications for recoverable shared objects. In *Proc. of the 35th International Symposium on Distributed Computing*, volume 209 of *DISC*, pages 29:1–29:19, 2021. doi:10.4230/LIPICS.DISC.2021.29.

38 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. doi:10.1145/103727.103729.

39 Mohammad Moridi, Erica Wang, Amelia Cui, and Wojciech M. Golab. A closer look at detectable objects for persistent memory. In *Proc. of the Workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems (ApPLIED)*, pages 56–64, 2022. doi:10.1145/3524053.3542749.

40 Liad Nahum, Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Recoverable and detectable fetch&add. In *Proc. of the 25th International Conference on Principles of Distributed Systems (OPODIS)*, pages 29:1–29:17, 2021. doi:10.4230/LIPICS.OPODIS.2021.29.

41 Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proc. of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 401–410, 2012. doi:10.1145/2150976.2151018.

42 Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency: Semantics for byte-addressable nonvolatile memory technologies. *IEEE Micro*, 35(3):125–131, 2015. doi:10.1109/MM.2015.46.

43 Andy Rudoff. cascade lake doesn't support clwb?, 2021. [last accessed 11/04/2024]. URL: https://groups.google.com/g/pmem/c/DRdYIc70RHc/m/rtoP681rAAAJ.

44 Andy Rudoff and the Intel PMDK Team. Persistent memory development kit, 2020. [last accessed 2/11/2021]. URL: https://pmem.io/pmdk/.

45 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 *Proc. of the 23rd International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 505–509, 2021. doi:10.1007/978-3-030-91081-5_38.

46 Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: lightweight persistent memory. In *Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 91–104, 2011. doi:10.1145/1950365.1950379.

47 Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch, and Erez Petrank. Flit: a library for simple and efficient persistent algorithms. In *Proc. of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 309–321, 2022. doi:10.1145/3503221.3508436.

A Proof of Correctness from Section 4

A.1 Proof of Theorem 1

Proof. The linearization points defined below in Table 4 establish a strict linearization for any history of the generic construction. Any pending operation that is included in this linearization must be completed with a matching response, and the only applicable case is a pending **detectable-FA**ϕ operation that reaches line 18. The response of such an operation is the value written earlier to $NextRet[i]$ at line 11, which would be returned at line 21 if the operation had run to completion.

Table 4 Linearization points of generic construction from Figure 1.

Operation	Special case	Linearization point
detectable-FAϕ	non-trivial state transition applied to S at line 18 (i.e., S overwritten with new value)	write to S at line 18
detectable-FAϕ	trivial state transition applied to S at line 18 (i.e., S overwritten with same value) and line 20 completed in the same operation	write to S at line 18
detectable-FAϕ	remaining cases	n/a (does not take effect)
resolve	operation runs to completion	arbitrary point in the operation's execution
resolve	remaining cases	n/a (does not take effect)

It remains to show that the chosen strict linearization is a legal history, meaning that each operation returns a correct response. Recall that correctness in this context is interpreted with respect to the implemented object type, which is a subset of Moridi et al.'s UDSS [39] applied to a generic **FA**ϕ operation that returns the previous abstract state of the object and applies a state transition according to the user-specified function ϕ.

We proceed by first proving a state invariant in Theorem 5 below:

▶ **Lemma 5.** *Consider a finite history H of the generic construction, let L be the corresponding strict linearization determined by the linearization points defined in Table 4, and let $(s, \mathcal{A}, \mathcal{T}, \mathcal{R})$ denote the abstract state of the implemented object at the end of L. Recall that s is the state of the underlying non-detectable type (in our case* **FA**ϕ*), $\mathcal{A}[i]$ identifies the signature (in our case the function ϕ) of the last detectable operation by p_i in L, $\mathcal{T}[i]$ is the tag argument of this operation, and $\mathcal{R}[i]$ is the operation's response. Recall also that $\mathcal{A}[i] = \mathcal{T}[i] = \mathcal{R}[i] = \perp$ if p_i does not have any detectable operation in L. Then the state $(s, \mathcal{A}, \mathcal{T}, \mathcal{R})$ is related to the variables of the generic construction at the end of H as follows:*

- *if $C[i] \in \{0, 1\}$ then $\mathcal{A}[i] = NextOp[i]$, $\mathcal{T}[i] = NextTag[i]$, and $\mathcal{R}[i] = NextRet[i]$*
- *if $C[i] = 2$ then $\mathcal{A}[i] = PrevOp[i]$, $\mathcal{T}[i] = PrevTag[i]$, and $\mathcal{R}[i] = PrevRet[i]$*
- *if $C[i] = 3$ and p_i applied a nontrivial state transition (i.e., changed the value of S) at line 18 in the* **detectable-FA**ϕ *operation where it most recently assigned $C[i] = 3$ at line 17, then $\mathcal{A}[i] = NextOp[i]$, $\mathcal{T}[i] = NextTag[i]$, and $\mathcal{R}[i] = NextRet[i]$*
- *if $C[i] = 3$ and p_i applied a trivial state transition (i.e., did not change the value of S) at line 18 in the* **detectable-FA**ϕ *operation where it most recently assigned $C[i] = 3$ at line 17, then $\mathcal{A}[i] = PrevOp[i]$, $\mathcal{T}[i] = PrevTag[i]$, and $\mathcal{R}[i] = PrevRet[i]$*
- *if $C[i] = 3$ and p_i has not yet reached line 18 in the* **detectable-FA**ϕ *operation where it most recently assigned $C[i] = 3$ at line 17, then $\mathcal{A}[i] = PrevOp[i]$, $\mathcal{T}[i] = PrevTag[i]$, and $\mathcal{R}[i] = PrevRet[i]$*

Moreover, $s = S$ (i.e., the abstract value of the implemented object matches the value stored in the variable S).

Correctness of the **detectable-FA**ϕ operation follows from the last clause of the invariant ($s = S$) and the use of the recoverable mutex RM to protect updates to S. The mutex ensures that the value of S does not change between when p_i reads S at line 11 and when p_i executes the linearization point of the **detectable-FA**ϕ operation at line 18 before returning the value read at line 21, which implies that the return value equals the value of S immediately before the linearization point. Next, consider the **resolve** operation. A completed call to

resolve by p_i enforces $C[i] = 0$ via line 27, line 32, or line 41, and so Theorem 5 ensures that $\mathcal{A}[i] = NextOp[i]$, $\mathcal{T}[i] = NextTag[i]$, and $\mathcal{R}[i] = NextRet[i]$ when p_i reaches line 42. Since p_i's **resolve** operation returns $\langle NextOp[i], NextTag[i], NextRet[i]\rangle$ at line 42, the lemma implies that the return value is equal to $\langle \mathcal{A}[i], \mathcal{T}[i], \mathcal{R}[i]\rangle$, as required by the UDSS.

Before proceeding with the proof of Theorem 5, we establish a secondary lemma that captures the effect of the helping mechanism.

▶ **Lemma 6.** *When a process p_i reaches line 37 of* **resolve***, the value of cmp is the value of the variable S as of when p_i last crashed during an execution of* **detectable-FA**ϕ.

Proof. First, note that p_i can only reach line 37 when $C[i] = 3$ due to p_i's earlier execution of line 33 and branch to line 34. This implies that p_i invoked **detectable-FA**ϕ at least once since $C[i] = 3$ can only be established by p_i at line 17, and that p_i crashed during its last invocation of **detectable-FA**ϕ as otherwise it would have reached line 20 and reset $C[i]$ back to 0 before returning. Furthermore, this most recent crash occurred after line 17 since Assumption 1 requires p_i to complete a call to **resolve** after each crash during **detectable-FA**ϕ, and since every complete execution of **resolve** by p_i resets $C[i]$ to 0 at line 27, line 32, or line 41. Since p_i's last crash in **detectable-FA**ϕ occurred after line 17, it follows that p_i already completed line 15 and assigned $P = i$, where the latter condition holds until the point of failure due to the critical section enforced by RM.

Next, consider how p_i arrived at line 37 of **resolve**. If $Help[i] = \perp$ at line 35 then *cmp* is the value read by p_i from S at line 34, and in this case no process has overwritten S between p_i's most recent crash in **detectable-FA**ϕ and p_i's execution of line 34. To see this, note that any process that overwrites S at line 18 after p_i's crash must first acquire the mutex RM at line 9 after p_i's crash, and then execute the helping mechanism at lines 13 to 14 inside the critical section. The first process that executes the helping mechanism after p_i's crash does so with $P = i$, which ensures that $Help[i]$ is overwritten with a value different from $\perp$ at line 14. Thus, the lemma holds when $Help[i] = \perp$ at line 35. Otherwise p_i observes $Help[i] \neq \perp$ at line 35, and *cmp* is the value read by p_i from $Help[i]$ at line 36. In this case some process p_j already executed the helping mechanism with $P = i$ before p_i read $Help[i]$ at line 37, and so $Help[i]$ at that point already holds the value read by p_j from S at line 14. Since p_j executes the helping mechanism after p_i's crash and before any process overwrites S, as explained earlier, the lemma holds in this case as well. ◀

Finally, we present the proof of Theorem 5 below.

Proof of Theorem 5. We proceed by induction on the number of steps, k, in the history H. We will use subscript notation in reference to the value of a program variable or abstract state component at the end of a history, for example denoting the value of $NextOp[i]$ at the end of H by $NextOp_H[i]$, or denoting the value of $\mathcal{A}[i]$ at the end of a linearization L of H by $\mathcal{A}_L[i]$.

In the base case $k = 0$, and $H = L = \langle\rangle$. Then $C_H[i] = 0$, $NextOp_H[i] = NextTag_H[i] = NextRet_H[i] = \perp$, and $S_H = s_0$ by initialization. Similarly, $\mathcal{A}_L[i] = \mathcal{T}_L[i] = \mathcal{R}_L[i] = \perp$ and $s_L = s_0$ by definition of the initial state of the UDSS. The lemma holds since $\mathcal{A}_L[i] = NextOp_H[i]$, $\mathcal{T}_L[i] = NextTag_H[i]$, $\mathcal{R}_L[i] = NextRet_H[i]$, and $s_L = S_H$.

Next, choose an arbitrary $k \geq 0$ and suppose for induction that the lemma holds for any history H of length k. Fix such a history H and let H' be an extension of H by another step σ by some process p_i. Let L and L' denote the strict linearizations of H and H', respectively. We proceed by a case analysis on σ. Steps where p_i accesses RM, or where p_i merely reads a program variable, are excluded from the analysis since the lemma follows directly from the induction hypothesis in those cases. The same principle applies to crash steps.

Case A. σ assigns $C[i] = 1$ at line 1 of **detectable-FA**ϕ. It follows from Assumption 1 that p_i can only invoke **detectable-FA**ϕ if this is p_i's first operation invocation, or if its last operation was a complete execution of **detectable-FA**ϕ or **resolve**. Then $C_H[i] = 0$ holds either by initialization or by the action of p_i's last operation, and so the induction hypothesis states that $\mathcal{A}_L[i] = NextOp_H[i]$, $\mathcal{T}_L[i] = NextTag_H[i]$, $\mathcal{R}_L[i] = NextRet_H[i]$, and $s_L = S_H$. The lemma requires $\mathcal{A}_{L'}[i] = NextOp_{H'}[i]$, $\mathcal{T}_{L'}[i] = NextTag_{H'}[i]$, $\mathcal{R}_{L'}[i] = NextRet_{H'}[i]$, and $s_{L'} = S_{H'}$ since $C_{H'}[i] = 1$. The lemma follows from the induction hypothesis since σ maintains $L' = L$, and does not affect any variable aside from $C[i]$.

Case B. σ updates $PrevOp[i]$, $PrevTag[i]$ or $PrevRet[i]$ at lines 2 to 4 of **detectable-FA**ϕ. Then $C_H[i] = 1$ holds by p_i's earlier execution of line 1. The induction hypothesis states that $\mathcal{A}_L[i] = NextOp_H[i]$, $\mathcal{T}_L[i] = NextTag_H[i]$, $\mathcal{R}_L[i] = NextRet_H[i]$, and $s_L = S_H$. The lemma requires $\mathcal{A}_{L'}[i] = NextOp_{H'}[i]$, $\mathcal{T}_{L'}[i] = NextTag_{H'}[i]$, $\mathcal{R}_{L'}[i] = NextRet_{H'}[i]$, and $s_{L'} = S_{H'}$ since $C_{H'}[i] = 1$. The lemma follows from the induction hypothesis since σ maintains $L' = L$, and does not affect any variable aside from $PrevOp[i]$, $PrevTag[i]$, and $PrevRet[i]$.

Case C. σ assigns $C[i] = 2$ at line 5 of **detectable-FA**ϕ. Then $C_H[i] = 1$ holds by p_i's earlier execution of line 1. The induction hypothesis states that $\mathcal{A}_L[i] = NextOp_H[i]$, $\mathcal{T}_L[i] = NextTag_H[i]$, $\mathcal{R}_L[i] = NextRet_H[i]$, and $s_L = S_H$. The lemma requires $\mathcal{A}_{L'}[i] = PrevOp_{H'}[i]$, $\mathcal{T}_{L'}[i] = PrevTag_{H'}[i]$, $\mathcal{R}_{L'}[i] = PrevRet_{H'}[i]$, and $s_{L'} = S_{H'}$ since $C_{H'}[i] = 2$. Step σ maintains $L' = L$ and does not affect any variable other than $C[i]$, and so the lemma holds by p_i's prior execution of lines 2 to 4, which copy $NextOp[i]$, $NextTag[i]$ and $NextRet[i]$ to $PrevOp[i]$, $PrevTag[i]$ and $PrevRet[i]$, respectively.

Case D. σ updates $NextOp[i]$, $NextTag[i]$ or $NextRet[i]$ at lines 6 to 11 of **detectable-FA**ϕ. Then $C_H[i] = 2$ holds by p_i's earlier execution of line 5. The induction hypothesis states that $\mathcal{A}_L[i] = PrevOp_H[i]$, $\mathcal{T}_L[i] = PrevTag_H[i]$, $\mathcal{R}_L[i] = PrevRet_H[i]$, and $s_L = S_H$. The lemma requires $\mathcal{A}_{L'}[i] = PrevOp_{H'}[i]$, $\mathcal{T}_{L'}[i] = PrevTag_{H'}[i]$, $\mathcal{R}_{L'}[i] = PrevRet_{H'}[i]$, and $s_{L'} = S_{H'}$ since $C_{H'}[i] = 2$. The lemma follows from the induction hypothesis since σ maintains $L' = L$, preserves $C_{H'}[i] = 2$, and does not affect any variable aside from $NextOp[i]$, $NextTag[i]$, and $NextRet[i]$.

Case E. σ assigns $C[i] = 3$ at line 17 of **detectable-FA**ϕ. Then $C_H[i] = 2$ holds by p_i's earlier execution of line 5. The induction hypothesis states that $\mathcal{A}_L[i] = PrevOp_H[i]$, $\mathcal{T}_L[i] = PrevTag_H[i]$, $\mathcal{R}_L[i] = PrevRet_H[i]$, and $s_L = S_H$. The lemma requires $\mathcal{A}_{L'}[i] = PrevOp_{H'}[i]$, $\mathcal{T}_{L'}[i] = PrevTag_{H'}[i]$, $\mathcal{R}_{L'}[i] = PrevRet_{H'}[i]$, and $s_{L'} = S_{H'}$ since $C_{H'}[i] = 3$ and since p_i has not yet applied a state transition to S at line 18. The lemma follows from the induction hypothesis since σ maintains $L' = L$ and does not affect any variable other than $C[i]$.

Case F. σ updates $Help[P]$ at line 14 or updates P at line 15 of **detectable-FA**ϕ. Then $C_H[i] = 3$ holds by p_i's earlier execution of line 17. The induction hypothesis states that $\mathcal{A}_L[i] = PrevOp_H[i]$, $\mathcal{T}_L[i] = PrevTag_H[i]$, $\mathcal{R}_L[i] = PrevRet_H[i]$, and $s_L = S_H$ since p_i has not yet applied a state transition to S at line 18. Similarly, the lemma requires $\mathcal{A}_{L'}[i] = PrevOp_{H'}[i]$, $\mathcal{T}_{L'}[i] = PrevTag_{H'}[i]$, $\mathcal{R}_{L'}[i] = PrevRet_{H'}[i]$, and $s_{L'} = S_{H'}$. The lemma follows from the induction hypothesis since σ maintains $L' = L$ and does not affect any variable other than $Help[P]$ or P.

Case G. σ applies a nontrivial state transition to S at line 18 of **detectable-FA**ϕ. In this case L' extends L by one **detectable-FA**ϕ operation, $S_{H'} \neq S_H$, and $C_H[i] = 3$ holds by p_i's earlier execution of line 17. The induction hypothesis states that $\mathcal{A}_L[i] = PrevOp_H[i]$, $\mathcal{T}_L[i] = PrevTag_H[i]$, $\mathcal{R}_L[i] = PrevRet_H[i]$, and $s_L = S_H$. The lemma requires $\mathcal{A}_{L'}[i] =$

$NextOp_{H'}[i]$, $\mathcal{T}_{L'}[i] = NextTag_{H'}[i]$, $\mathcal{R}_{L'}[i] = NextRet_{H'}[i]$, and $s_{L'} = S_{H'}$ where $\mathcal{A}_{L'}[i]$, $\mathcal{T}_{L'}[i]$, and $\mathcal{R}_{L'}[i]$ refer to p_i's current **detectable-FA**ϕ operation. It follows from p_i's earlier execution of line 6 that $\mathcal{A}_{L'}[i] = NextOp_{H'}[i]$, as required. It follows from p_i's earlier execution of line 7 that $\mathcal{T}_{L'}[i] = NextTag_{H'}[i]$, as required. It follows from p_i's earlier execution of line 11 and from the fact that updates to S are protected by RM that $NextRet_{H'}[i] = NextRet_H[i] = S_H$, where $\mathcal{R}_{L'}[i] = S_H$ since σ is the linearization point of p_i's current **detectable-FA**ϕ operation. Thus, $\mathcal{R}_{L'}[i] = NextRet_{H'}[i]$ holds, as required. Finally, $s_{L'} = S_{H'}$ follows by the action of step σ, which updates S to the new abstract value computed via the function ϕ, and where the input to ϕ is S_H since p_i holds the recoverable mutex RM while reading and writing S at line 18. Thus, the lemma holds.

Case H. σ applies a trivial state transition to S at line 18 of **detectable-FA**ϕ. In this case $L' = L$, $S_{H'} = S_H$, and $C_H[i] = 3$ holds by p_i's earlier execution of line 17. The induction hypothesis states that $\mathcal{A}_L[i] = PrevOp_H[i]$, $\mathcal{T}_L[i] = PrevTag_H[i]$, $\mathcal{R}_L[i] = PrevRet_H[i]$, and $s_L = S_H$. The lemma requires $\mathcal{A}_{L'}[i] = PrevOp_{H'}[i]$, $\mathcal{T}_{L'}[i] = PrevTag_{H'}[i]$, $\mathcal{R}_{L'}[i] = PrevRet_{H'}[i]$, and $s_{L'} = S_{H'}$. The lemma follows from the induction hypothesis since σ maintains $L' = L$ and does not affect any variable aside from S, where $S_{H'} = S_H$.

Case I. σ assigns $C[i] = 0$ at line 20 of **detectable-FA**ϕ. Then $C_H[i] = 3$ holds by p_i's earlier execution of line 17. The lemma requires $\mathcal{A}_{L'}[i] = NextOp_{H'}[i]$, $\mathcal{T}_{L'}[i] = NextTag_{H'}[i]$, $\mathcal{R}_{L'}[i] = NextRet_{H'}[i]$, and $s_{L'} = S_{H'}$. where $\mathcal{A}_{L'}[i]$, $\mathcal{T}_{L'}[i]$, and $\mathcal{R}_{L'}[i]$ refer to p_i's current **detectable-FA**ϕ operation.

Subcase I1. p_i applied a nontrivial state transition to S earlier at line 18. In this case p_i's current **detectable-FA**ϕ operation already reached its linearization point prior to step σ. The induction hypothesis states that $\mathcal{A}_L[i] = NextOp_H[i]$, $\mathcal{T}_L[i] = NextTag_H[i]$, $\mathcal{R}_L[i] = NextRet_H[i]$, and $s_L = S_H$. The lemma follows from the induction hypothesis since σ maintains $L' = L$ and does not affect any variable aside from $C[i]$.

Subcase I2. p_i applied a trivial state transition to S earlier at line 18. In this case p_i's current **detectable-FA**ϕ operation takes effect at step σ, and so L' extends L by one **detectable-FA**ϕ operation. The induction hypothesis states that $s_L = S_H$. The lemma requires $\mathcal{A}_{L'}[i] = NextOp_{H'}[i]$, $\mathcal{T}_{L'}[i] = NextTag_{H'}[i]$, $\mathcal{R}_{L'}[i] = NextRet_{H'}[i]$, and $s_{L'} = S_{H'}$, where $\mathcal{A}_{L'}[i]$, $\mathcal{T}_{L'}[i]$, and $\mathcal{R}_{L'}[i]$ refer to p_i's current **detectable-FA**ϕ operation. It follows from p_i's earlier execution of line 6 that $\mathcal{A}_{L'}[i] = NextOp_{H'}[i]$, as required. It follows from p_i's earlier execution of line 7 that $\mathcal{T}_{L'}[i] = NextTag_{H'}[i]$, as required. It follows from p_i's earlier execution of line 11 and from the fact that updates to S are protected by RM that $NextRet_{H'}[i] = NextRet_H[i] = S_H$, where $\mathcal{R}_{L'}[i] = S_H$ since σ is the linearization point of p_i's current **detectable-FA**ϕ. Thus, $\mathcal{R}_{L'}[i] = NextRet_{H'}[i]$ holds, as required. Finally, $s_{L'} = s_L$ follows despite $L' \neq L$ since p_i applied a trivial state transition in its current **detectable-FA**ϕ operation, $s_L = S_H$ follows from the induction hypothesis, and $S_H = S_{H'}$ follows since σ does not update S. Thus, $s_{L'} = S_{H'}$ holds as required, and the lemma holds.

Case J. σ assigns $C[i] = 0$ at line 27 of **resolve**. Then $C_H[i] = 1$ by p_i's earlier execution of line 26 and branch to line 27. The induction hypothesis states that $\mathcal{A}_L[i] = NextOp_H[i]$, $\mathcal{T}_L[i] = NextTag_H[i]$, $\mathcal{R}_L[i] = NextRet_H[i]$, and $s_L = S_H$. The lemma requires $\mathcal{A}_{L'}[i] = NextOp_{H'}[i]$, $\mathcal{T}_{L'}[i] = NextTag_{H'}[i]$, $\mathcal{R}_{L'}[i] = NextRet_{H'}[i]$, and $s_{L'} = S_{H'}$. The lemma follows from the induction hypothesis since σ maintains $L' = L$ and does not affect any variable aside from $C[i]$.

Case K. σ updates $NextOp[i]$, $NextTag[i]$, or $NextRet[i]$ at lines 29 to 31 of **resolve**. Then $C_H[i] = 2$ by p_i's earlier execution of line 26 and branch to line 27. The induction hypothesis states that $\mathcal{A}_L[i] = PrevOp_H[i]$, $\mathcal{T}_L[i] = PrevTag_H[i]$, $\mathcal{R}_L[i] = PrevRet_H[i]$, and $s_L = S_H$. The lemma requires $\mathcal{A}_{L'}[i] = PrevOp_{H'}[i]$, $\mathcal{T}_{L'}[i] = PrevTag_{H'}[i]$, $\mathcal{R}_{L'}[i] = PrevRet_{H'}[i]$, and $s_{L'} = S_{H'}$. The lemma follows from the induction hypothesis since σ maintains $L' = L$, and since step σ does not modify S, $PrevOp[i]$, $PrevTag[i]$, or $PrevRet[i]$.

Case L. σ assigns $C[i] = 0$ at line 32 of **resolve**. Then $C_H[i] = 2$ by p_i's earlier execution of line 28 and branch to line 29. The induction hypothesis states that $\mathcal{A}_L[i] = PrevOp_H[i]$, $\mathcal{T}_L[i] = PrevTag_H[i]$, $\mathcal{R}_L[i] = PrevRet_H[i]$, and $s_L = S_H$. The lemma requires $\mathcal{A}_{L'}[i] = NextOp_{H'}[i]$, $\mathcal{T}_{L'}[i] = NextTag_{H'}[i]$, $\mathcal{R}_{L'}[i] = NextRet_{H'}[i]$, and $s_{L'} = S_{H'}$. The lemma follows from the induction hypothesis, since σ maintains $L' = L$ and step σ does not modify S, and also from p_i's earlier execution of lines 29 to 31, where p_i restores $NextOp[i]$, $NextTag[i]$, and $NextRet[i]$ from $PrevOp[i]$, $PrevTag[i]$, and $PrevRet[i]$.

Case M. σ updates $NextOp[i]$, $NextTag[i]$, or $NextRet[i]$ at lines 38 to 40. Then $C_H[i] = 3$ by p_i's earlier execution of line 33 and branch to line 34. Furthermore, the value of p_i's private variable *cmp* equals the value of S as of when p_i failed during its last execution of **detectable-FA**ϕ by Theorem 6. Then p_i's execution of lines 38 to 40 implies that $LastS[i] = cmp$ held at line 37, where $LastS[i]$ is the value p_i fetched from S at line 12 during its last **detectable-FA**ϕ operation before assigning $C[i] = 3$ at line 17. This implies that p_i never reached line 18, or it did reach this line but applied a trivial state transition to S, given our earlier conclusion regarding the value of *cmp* and the fact that RM protects lines 12 to 18. In either case, p_i's last **detectable-FA**ϕ operation did not take effect, and so the induction hypothesis states that $\mathcal{A}_L[i] = PrevOp_H[i]$, $\mathcal{T}_L[i] = PrevTag_H[i]$, $\mathcal{R}_L[i] = PrevRet_H[i]$, and $s_L = S_H$. The lemma requires $\mathcal{A}_{L'}[i] = PrevOp_{H'}[i]$, $\mathcal{T}_{L'}[i] = PrevTag_{H'}[i]$, $\mathcal{R}_{L'}[i] = PrevRet_{H'}[i]$, and $s_{L'} = S_{H'}$. The lemma follows from the induction hypothesis since σ maintains $L' = L$ and does not affect any variable aside from $NextOp[i]$, $NextTag[i]$, and $NextRet[i]$.

Subcase N. σ assigns $C[i] = 0$ at line 41 of **resolve**. Then $C_H[i] = 3$ by p_i's earlier execution of line 33 and branch to line 34. The lemma requires $\mathcal{A}_{L'}[i] = NextOp_{H'}[i]$, $\mathcal{T}_{L'}[i] = NextTag_{H'}[i]$, $\mathcal{R}_{L'}[i] = NextRet_{H'}[i]$, and $s_{L'} = S_{H'}$. If p_i's last **detectable-FA**ϕ operation took effect by applying a nontrivial state transition to S then the induction hypothesis states that $\mathcal{A}_L[i] = NextOp_H[i]$, $\mathcal{T}_L[i] = NextTag_H[i]$, $\mathcal{R}_L[i] = NextRet_H[i]$, and $s_L = S_H$. In this case the lemma follows from the induction hypothesis since σ maintains $L' = L$ and does not affect any variable other than $C[i]$. Otherwise p_i's last **detectable-FA**ϕ operation either applied a trivial state transition to S, or did not reach line 18 at all, and hence did not take effect. In this case the induction hypothesis states that $\mathcal{A}_L[i] = PrevOp_H[i]$, $\mathcal{T}_L[i] = PrevTag_H[i]$, $\mathcal{R}_L[i] = PrevRet_H[i]$, and $s_L = S_H$. Now consider p_i's earlier comparison of *cmp* against $LastS[i]$ at line 37, and note that $LastS[i]$ is the value p_i fetched from S at line 12 during its last **detectable-FA**ϕ operation before assigning $C[i] = 3$ at line 17. Furthermore, the value of p_i's private variable *cmp* equals the value of S as of when p_i failed during its last execution of **detectable-FA**ϕ by Theorem 6. Since p_i never reached line 18, or it did reach this line but applied a trivial state transition to S, and since RM protects lines 12 to 18, it follows that $LastS[i] = cmp$ held at line 37. Thus, p_i branched to line 38 and executed lines 38 to 40 prior to reaching line 41. Given the induction hypothesis, lines 38 to 40 ensure that $\mathcal{A}_{L'}[i] = NextOp_{H'}[i]$, $\mathcal{T}_{L'}[i] = NextTag_{H'}[i]$, and $\mathcal{R}_{'L}[i] = NextRet_{H'}[i]$ by copying $PrevOp[i]$, $PrevTag[i]$ and $PrevRet[i]$ to $NextOp[i]$, $NextTag[i]$ and $NextRet[i]$, respectively. Finally, $s_{L'} = S_{H'}$ holds since σ maintains $L' = L$ and does not write S. Thus, the lemma holds. ◀

This concludes the proof of Theorem 1. ◀

A.2 Proof of Theorem 2

Proof. Aside from accesses to the recoverable mutex RM at lines 8 to 9, line 19, lines 23 to 25, the operations **detectable-FA**ϕ and **resolve** are wait-free and have $O(1)$ step complexity. Termination and RMR complexity therefore both depend on RM. This mutex is accessed correctly in the generic construction, meaning that its recovery, entry, and exit sections are called in the correct order, and that **Recover** is always called after a crash outside the non-critical section due to Assumption 1. Thus, each passage through RM terminates eventually in any infinite fair history by the starvation freedom of RM, and incurs $O(R)$ RMRs in any history. Since each **detectable-FA**ϕ and **resolve** operation makes at most one passage through RM, it follows that the worst-case RMR complexity per operation in the generic construction is $O(R)$. ◀

B Adapting the Detectable Construction for FCFS

While our generic construction of detectable objects does not preserve wait-freedom due to the use of an RME lock in the **detectable-FA**ϕ operation, there are several concrete ways in which it can guarantee partial wait freedom. Several techniques were discussed already in Section 4, and so we focus here on optimizations related to supporting first-come-first-serve (FCFS) fairness in RME and RME-like algorithms based on our detectable objects.

If the recoverable mutex RM supports first-come-first-serve (FCFS) (Definition 7) then it can be shown that the **detectable-FA**ϕ operation always *appears* to take effect inside the doorway of RM. Such operations are thus *wait-free up to their linearization point* provided that RM also provides bounded recovery. Note that steps taken up to the end of the doorway of RM at line 9 do not by themselves decide *whether or not* a particular **detectable-FA**ϕ operation will take effect since the latter depends on a process subsequently reaching line 18 or line 20 before crashing (see Table 4 in Appendix A). Thus, the exact order of linearization points is not necessarily determined without some busy-waiting, for example where p_1 completes the doorway of RM before p_2 starts the doorway, p_2 then waits for p_1 to complete the CS of RM, p_1 crashes before starting the CS, p_2 finally updates S at line 18, and only then it becomes irrevocable that p_2's **detectable-FA**ϕ operation takes effect while p_1's does not. However, this limitation is not restrictive if the implemented **detectable-FA**ϕ operation is by itself used to enforce FCFS fairness in some recoverable mutex RM' that uses detectable base objects, meaning that the recovery section and doorway of RM at lines 8 to 9 become the doorway of RM'. In this specific use case, the relative order of doorway executions only matters under Definition 7 for pairs of processes that actually enter the CS of RM' in their corresponding passages, which occurs only after completing the entry section of RM' failure-free, including the entire **detectable-FA**ϕ operation under consideration; this ensures that the linearization points are well-defined.

▶ **Definition 7** (FCFS). *An RME algorithm satisfies* first come first serve *(FCFS) if for any two passages by distinct processes p_1 and p_2, if p_1 completes the doorway before p_2 starts the doorway and both processes enter the CS in the corresponding passages then p_1 enters the CS before p_2.*

▶ **Theorem 8.** *Suppose that RM supports the FCFS property. Then* **detectable-FA**ϕ *operations appear to take effect in the doorway of RM, meaning that for every history H of the generic construction, there is a strict linearization S such that the total order of operations in S is consistent with the partial order of doorway executions corresponding to line 9. Specifically, if $op_1 <_S op_2$ holds then $op_2 <_D op_1$ is false, where $<_S$ is the total order*

over operations in S, and $<_D$ is the partial order over operations in S such that $op_1 <_D op_2$ if (and only if) op_1 and op_2 are both **detectable-FA**ϕ *operations and the doorway of RM is completed in op_1 before it is started in op_2.*

Proof. It follows from Theorem 1 that H has a strict linearization S consistent with the linearization points defined in Table 4 (see Appendix A). Take any such S, and let $<_S$ denote the corresponding total order of operations, including both **detectable-FA**ϕ and **resolve**. Now suppose for contradiction that $op_1 <_S op_2$ and $op_2 <_D op_1$ both hold for some pair of operations, and so the doorway in op_2 was completed before the doorway in op_1 was started. It follows from $op_1 <_D op_2$ and the FCFS property of RM that if the CS is entered in both op_1 and op_2 then it is entered in op_2 before op_1. Since each operation in S linearizes at a point inside the CS according to Table 4, the actual order of CS executions in H implies that $op_2 <_S op_1$, which contradicts our earlier supposition. ◀

A slightly weaker result can be shown if RM only supports the k-FCFS property [25] for some k, stated below in Theorem 9, which refers to the concepts of failure-concurrency and interfering super-passages. A super-passage by a process p_i is *0-failure-concurrent* (0-FC) if p_i crashes in this super-passage. Two super-passages *interfere* if neither super-passage ends (with a complete failure-free passage) before the other starts. A super-passage is *k-failure-concurrent* (k-FC) for some $k \geq 1$ if it interferes with some $(k-1)$-FC super-passage (possibly of the same process). A passage is k-FC for some $k \geq 1$ if it belongs to a k-FC super-passage.

▶ **Definition 9** (k-FCFS). *An RME algorithm satisfies k-FCFS for some $k \geq 0$ if for any two passages by distinct processes p_1 and p_2, if p_1 completes the doorway before p_2 starts the doorway and both processes enter the CS in the corresponding passages then p_1 enters the CS before p_2* unless at least one of these passages is k-FC.

To connect the k-FCFS property of RM to the behavior of our generic construction, we generalize the concepts of super-passages and k-FC to object implementations. A *super-passage* of a process p_i with respect to the generic construction is a maximal contiguous sequence of operations by p_i where at most the last operation in the sequence is complete. For example, a single complete execution of **detectable-FA**ϕ is a super-passage, and similarly for an execution of **detectable-FA**ϕ that is interrupted by a crash followed by one or more executions of **resolve** where only the last **resolve** operation is complete. We say that a super-passage of the generic construction by p_i is 0-FC if it contains at least one operation interrupted by p_i's crash, which in this context implies that the first operation in the super-passage is the interrupted one. Two super-passages *interfere* if neither super-passage ends (with a completed operation) before the other starts. A super-passage of the generic construction is k-FC for some $k \geq 1$ if it interferes with some $(k-1)$-FC super-passage. An operation applied to the generic construction is k-FC for some $k \geq 1$ if it belongs to a k-FC super-passage.

▶ **Theorem 10.** *Suppose that RM supports the k-FCFS property for some $k \geq 0$. Then* **detectable-FA**ϕ *operations appear to take effect in the doorway of RM in all super-passages of the generic construction that are not k-FC, meaning that for every history H of the generic construction, there is a strict linearization S such that the total order of operations in S is consistent with the partial order of doorway executions corresponding to line 9 in operations that are not k-FC. Specifically, if $op_1 <_S op_2$ holds then $op_2 <_D op_1$ is false for any pair of operations op_1 and op_2 that are not k-FC, where $<_S$ is the total order over operations in S, and $<_D$ is the partial order over operations in S such that $op_1 <_D op_2$ if (and only if) op_1 and op_2 are both non-k-FC* **detectable-FA**ϕ *operations and the doorway of RM is completed in op_1 before the doorway of RM is started in op_2.*

Proof. We proceed roughly as in the proof of Theorem 8. It follows from Theorem 1 that H has a strict linearization S consistent with the linearization points defined in Table 4 (see Appendix A). Take any such S, and let $<_S$ denote the corresponding total order of operations, including both **detectable-FA**ϕ and **resolve**. Now suppose for contradiction that $op_1 <_S op_2$ and $op_2 <_D op_1$ both hold for some pair of operations, and so the doorway in op_2 was completed before the doorway in op_1 was started, where neither op_1 nor op_2 is k-FC. It follows from $op_1 <_D op_2$ and the k-FCFS property of RM that if the CS is entered in both op_1 and op_2 then it is entered in op_2 before op_1. Since each operation in S linearizes at a point inside the CS according to Table 4, the actual order of CS executions in H implies that $op_2 <_S op_1$, which contradicts our earlier supposition. ◀

C Bounding Space Complexity in the Detectable Object Construction

The space complexity of the algorithm can be bounded by reclaiming and reusing the queue nodes allocated at line 58. In the absence of failures, each process can maintain a single node and reuse it safely across consecutive passages, but crash failures introduce the risk of an unsafe memory operation where a predecessor process signals its successor redundantly by executing line 71 of **Exit** from line 57 of **Recover**, after the successor has already repurposed its queue node for a new passage. This particular use-after-free scenario can be made safe by storing a pointer to the predecessor's queue node in the *locked* field instead of a boolean, similarly to Algorithm 3 in [13]. The domain of values for *locked* must also include a designated initial value $\perp$, as well as a special value $\top$ indicating that the queue node was already locked and then unlocked. The algorithm is revised slightly in several places: at line 51 of **Recover** the condition $q_i.locked =$ **false** is replaced with $q_i.locked = \perp$; at line 60 of **Enter** $q_i.locked$ is initialized to $\perp$ instead of **false**; at line 64 of **Enter** $q_i.locked$ is set to *prev* instead of **true**; at line 66 of **Enter** the condition $q_i.locked \neq$ **true** is replaced with $q_i.locked = \top$; and at line 71 of **Exit** the assignment $q_i.next.locked :=$ **false** is replaced with $q_i.next.locked.\mathbf{CAS}(q_i, \top)$. Another potentially unsafe memory access occurs when a successor links with a predecessor at line 65 of **Enter** from line 52 of **Recover**, however this is already addressed by the condition at line 51 (now modified to $prev \neq \perp \wedge prev.next = \perp \wedge q_i.locked = \perp$). The latter ensures that line 52 is reached only if lines 64 to 65 have not yet been executed, and accounts for the possibility that the predecessor has already moved on to its next passage, in which case $prev.next = \perp$ may hold but $q_i.locked = \perp$ is false since the predecessor has already completed the revised line 71 at least once and updated $q_i.locked$ to $\top$.