Recoverable Mutual Exclusion with Abortability

Prasad Jayanti and Anup Joshi(✉)

Dartmouth College, Hanover, NH 03755, USA
anupj@cs.dartmouth.edu

Abstract. In light of recent advances in non-volatile main memory technology, there has been a flurry of research in designing algorithms that are resilient to process crashes. As a result of main memory non-volatility, a process is allowed to crash any time during the execution, without affecting the state of the data stored in the main memory. With the assumption that a process eventually restarts after a crash, prior works have focused on designing mutual exclusion algorithms that use the non-volatile main memory to recover from such crashes. Such mutual exclusion algorithms that provide multiple processes with a mutually exclusive access to a shared resource in the presence of process crashes are called Recoverable Mutual Exclusion (RME) algorithms. We present the first RME algorithm where a process has the ability to abort executing the algorithm, if it decides to give up its request for a shared resource before being granted access to that resource. With n being the maximum number of processes for which the algorithm is designed, in the absence of a crash our algorithm guarantees a worst-case remote memory references (RMR) complexity of $O(\log n)$ per passage on the Distributed Shared Memory (DSM) machines, and a complexity of $O(\log n)$ or $O(n)$ on Cache Coherent (CC) machines, depending on how caches are managed.

Keywords: Concurrent algorithm · Synchronization · Mutual exclusion · Recoverable algorithm · Fault tolerance · Non-volatile main memory · Shared memory · Multi-core algorithms

1 Introduction

Recent advances in non-volatile main memory (NVMM) technology [1–3] have given rise to designing algorithms that are resilient to process crashes. These memory technologies allow interfacing the processor directly with the non-volatile main memory. Therefore, in the event of a process crash, the system restarts the crashed process and the process then recovers from the crash by consulting the contents of the NVMM.

The first author is grateful to the Frank family and Dartmouth College for their support through James Frank Family Professorship of Computer Science.
The second author is grateful for the support from Dartmouth College.

M. F. Atig and A. A. Schwarzmann (Eds.): NETYS 2019, LNCS 11704, pp. 217–232, 2019.
https://doi.org/10.1007/978-3-030-31277-0_14

To leverage this advantage given by the NVMM, there has been a keen interest recently in designing algorithms for such systems. A starting point is to design a variant of the classical mutual exclusion problem [4] in which the objective is to protect access to a shared resource in a manner that atmost one process has access to the resource at any point in time. Thus, it began with Golab and Ramaraju [5] reformulating the classical mutual exclusion problem into the novel *Recoverable Mutual Exclusion* (RME) problem in 2016. After which there has been a flurry of research in designing algorithms for the RME problem [6–10]. The main interest in these works has been in designing algorithms with various desirable properties while maintaining the remote memory reference (RMR) complexity for Cache-Coherent (CC) multiprocessors and Distributed Shared Memory (DSM) multiprocessors to a minimum.

A straightforward approach to recover from process crashes would be to shut down the entire system and restart it. However, a motive behind designing RME algorithms is to make the crashes less disruptive to other processes which do not crash. Therefore, repairing the damage due to a crash by using the NVMM is far less demanding to a process that did not crash since it does not suffer from such a full-system restart. However, prior works on RME fall short in a crucial aspect when compared to solutions to the classical mutual exclusion problem. Imagine a real-time system with multiple threads that uses one of these RME algorithms to secure access to a critical shared resource. In the event of a crash the waiting time of a non-crashing thread would still be increased by the time it takes for the crashing process to recover from its crash. This issue is further amplified when the crashes are frequent and the system is operating under tight deadlines. Hence, it makes sense for a waiting thread to be able to abort its attempt to acquire access to the shared resource, and not miss any of its other deadlines. Although classical mutual exclusion algorithms are amenable to support the ability to abort, unfortunately, none of the prior works on the RME problem support such an ability to abort.

In this paper, we present the first RME algorithm that provides the abort functionality. Our algorithm has a bounded RMR on the CC and DSM machines besides possessing some additional desirable properties.

Related Research. All of the prior work on RME has focused on designing algorithms that do not provide abortability as a capability. Golab and Ramaraju [5] formalized the RME problem and designed several algorithms by adapting traditional mutual exclusion algorithms. Ramaraju [11], Jayanti and Joshi [7], and Jayanti et al. [9] designed RME algorithms that support the First-Come-First-Served property [12]. Golab and Hendler [6] presented an algorithm that has sub-logarithmic RMR complexity on CC machines. In another work, Golab and Hendler [8] presented an algorithm that has the ideal $O(1)$ passage complexity, but this result assumes that *all* processes in the system crash *simultaneously*. Recently, Jayanti et al. [10] presented a unified algorithm that has a sub-logarithmic RMR complexity on both CC and DSM machines. For works not on RME but on the theme of crash-restart systems using non-volatile main-memory, Attiya et al. [13] present linearizable implementations of recoverable objects.

When it comes to abortability for classical mutual exclusion problem, Scott [14] and Scott and Scherer [15] designed abortable algorithms that build on the queue-based algorithms [16,17]. Jayanti [18] designed an algorithm based on read, write, and comparison primitives having $O(\log n)$ RMR complexity which is also optimal [19]. Lee [20] designed an algorithm for CC machines that uses the Fetch-and-Add and Fetch-and-Store primitives. Alon and Morrison [21] designed an algorithm for CC machines that has a sub-logarithmic RMR complexity and uses the read, write, Fetch-And-Store, and comparison primitives. Recently, Jayanti and Jayanti [22] designed an algorithm for the CC and DSM machines that has a constant amortized RMR complexity and uses the read, write, and Fetch-And-Store primitives. While the works mentioned so far have been deterministic algorithms, randomized versions of classical mutual exclusion with abortability exist. Pareek and Woelfel [23] give a sublogarithmic RMR complexity randomized algorithm and Giakkoupis and Woelfel [24] give an $O(1)$ expected amortized RMR complexity randomized algorithm.

Our Contribution. We show that, as with classical mutual exclusion, the recoverable mutual exclusion problem is amenable to abortability with a reasonable RMR complexity. We present the first abortable RME algorithm for the CC and DSM machines using only read, write, and comparison primitives. We design our algorithm by developing on ideas from a prior RME algorithm by Jayanti and Joshi [7]. Our algorithm has an RMR complexity of $O(f + \log n)$ when used on DSM machines and certain type of CC machines, but it has $O(f + n)$ RMR complexity on another type of CC machines (see Sect. 3.4 for full details), where n is the number of processes for which the algorithm is designed and f is the number of times a process crashes between the time it invokes and exits the algorithm. Attiya et al. [19] proved a lower bound that the RMR complexity is $\Omega(\log n)$ for even classical mutual exclusion algorithms that use read, write, and comparison primitives. Therefore, our algorithm adds only $O(1)$ RMR per crash on the DSM machines. In addition to the above, our algorithm satisfies the First-Come-First-Served [12] property. It would be interesting if it is possible to bring down the RMR complexity to $O(f + \log n)$ for all CC machines.

2 Model and Problem Specification

Our system consists of n asynchronous processes named $1, 2, \ldots, n$ and atomic persistent variables (which include shared variables and variables used by a single process). The persistent variables support the operations *read*, *write*, and *compare&swap* (CAS). The CAS operation has the signature: CAS(X, old, new), where X is a variable name, and *old* and *new* are some values. A CAS(X, old, new) operation atomically changes X's value to *new*, if X contained the value *old*, and returns *true*; otherwise, it returns $false$ and leaves X unchanged. The persistent variables are assumed to reside in the non-volatile

main memory (NVMM) [1–3], which allows them to retain their values in the event of a process crash. Note, our algorithm also uses process local variables, which are assumed to be stored in the process registers (we clarify in the description of the algorithm the nature of variables used). A *configuration* of the system is specified by the values of all shared variables and the states of the n processes, where the state of a process p is in turn specified by the value of PC_p, p's program counter, and the values of p's local variables. The configuration changes when a process takes a step. Any process can execute either a *normal step* or a *crash step* at any time. In a normal step of p, p executes the instruction pointed by its program counter PC_p. A crash step models the crash of a process and can occur regardless of what portion of code the process is executing.

The Abortable RME Problem. In the RME problem, each process repeatedly cycles through four sections of code—Remainder, Try, Critical, and Exit sections. An *algorithm* for RME specifies the code for the Try and Exit sections of each process. If a process p executes a normal step when in Remainder, p moves to Try; and if p executes a normal step when in CS, p moves to Exit (therefore, we encapsulate the CS of p with one normal step). A crash step of p sets PC_p to point to its Remainder section and sets all other registers of p to $\perp$. In addition, in the Abortable RME problem, p can receive an external signal to *abort* continuing to the CS while inside Try, in which case p may execute Exit without executing CS to go back to the Remainder[1]. A *run* of an algorithm is an infinite sequence of steps. We assume every run satisfies the following conditions: (i) if a process is in Try, Critical, or Exit sections, it later executes a (normal or crash) step, and (ii) if a process enters Remainder because of a crash step, it later executes a normal step.

RMR Complexity and Passages. In a CC machine each process has a cache. A read operation by a process p on a shared variable X fetches a copy of X from shared memory to p's cache, if a copy is not already present. Any non-read operation on X by any process invalidates copies of X at all caches. An operation on X by p counts as a *remote memory reference* (RMR) if either the operation is not a read or X's copy is not in p's cache. When a process crashes, we assume that its cache contents are lost. In a DSM machine, instead of caches, shared memory is partitioned, with one partition residing at each process, and each shared variable resides in exactly one partition. Any operation (read or non-read) by a process on a shared variable X is counted as an RMR if X is not in p's partition.

A *passage* of a process p in a run starts when p enters Try (from Remainder) and ends at the earliest later time when p returns to Remainder (either because p crashes or because p completes Exit and moves back to Remainder). A *super-passage* of a process p in a run starts when p either enters Try for the first

[1] p might have already set itself up to enter the CS, or could be executing the CS, in which case it executes Exit after completing the CS.

time in the run or when p enters Try for the first time after the previous super-passage has ended, and it ends when p returns to Remainder not by a crash step but by completing the Exit section. Note that p's super-passage can contain an unbounded number of p's passages because of its repeated crashes during the super-passage. The *passage RMR complexity* (respectively, *super-passage RMR complexity*) of an RME algorithm is the worst-case number of RMRs that a process incurs in a passage (respectively, in a super-passage). We express this RMR complexity in terms of n and f, where n is the number of processes for which the algorithm is designed and f is the number of times the process crashes during the super-passage.

Problem Statement. The goal is to design an algorithm (i.e., code for Try and Exit sections) for the Abortable RME problem, such that, all of the following conditions are met in every run of the algorithm. Conditions P1, P4, P5 are from Golab and Ramaraju [5], and P2, P6, P7, P8 are from Jayanti and Joshi [7]. The additional property P3 and the modifications needed for other properties to accomodate abortability are emphasized in italics.

P1. Mutual Exclusion: At most one process is in the CS at any point.
P2. Bounded Exit: There is a bound b such that, if a process p is in the Exit section and it executes steps without crashing, it enters the Remainder section in atmost b of its own steps.
P3. Bounded Abort: *There is a bound b such that, if a process p receives the abort signal and p executes steps without crashing, then p enters the CS or the Remainder section in b of its own steps.*
This property captures the intuition that a process frees itself from all waiting once it receives the abort signal.
P4. Starvation Freedom: If the total number of crashes in the run is finite and a process p has infinite number of steps, then p enters the CS in each super-passage *in which it does not receive an abort signal.*
P5. Critical Section Reentry (CSR) [5]: If a process p crashes while in the CS, then no other process enters the CS during the interval from p's crash to the point in the run when p next enters the CS.
P6. Wait-Free Critical Section Reentry (Wait-Free CSR) [7]: There is a bound b such that, if a process p crashes while in the CS, then p reenters the CS before completing b consecutive normal steps.
P7. First-Come-First-Served (FCFS) [7]: There is a bound b such that, if a process p performs b contiguous normal steps in its super-passage s before another process p' initiates its super-passage s' *and p does not receive an abort signal in s*, then p' does not enter the CS in super-passage s' before p first enters the CS in super-passage s'.
P8. Well-formedness: Let s be a normal step by p in which p completes the Try section, and s' be the latest step by p before s in which p starts a super-passage or p crashes outside of the Try section in CS, or Exit. *Well-formedness* stipulates where the control moves to after step s, as follows:

- If s' is a step when the super-passage starts, then s moves control to CS, *Exit section or Remainder section.*
- If s' is a crash step while p is in Try section, then s moves control to CS, *Exit section or Remainder section.*
- If s' is a crash step while p is in CS, then s moves control to CS.
- If s' is a crash step while p is in Exit, then s moves control to CS, Exit section, or Remainder section.

3 The Algorithm

We present our abortable RME algorithm in Fig. 1. The algorithm is designed for n processes, with each process getting a distinct name from the set $\{1, 2, \ldots, n\}$. All the persistent variables used by our algorithm are stored in the non-volatile main memory. Variables with names in small letters and a subscript of p to their name are variables local to the process p, and are stored in p's registers. We assume that the CS is an idempotent block of code, which allows a process to re-execute it from start even if the process crashes in the middle of the CS. We assume that the external signal to an arbitrary process p asking it to abort is made available at AbortSignal[p] as a boolean value, with a value of *true* indicating that the signal is active and a value of $false$ indicating otherwise. Our algorithm is obtained by expanding on ideas of Jayanti and Joshi's [7] algorithm. Therefore, like their algorithm, our algorithm relies on a special object called *min-array*. For more details about the min-array, please read the description of Registry in Sect. 3.1.

3.1 Shared Variables and Their Purpose

We describe below the role played by each shared variable used in the algorithm.

Go[p]: This is a flag that process p waits on before entering the CS and supports the read, write, and CAS operations. To achieve the local-spin property, Go[p] is allocated to p's memory module on DSM machines. This variable is set to a non-zero integer value by p inside the Try Section. A process q makes p the owner of the CS first, and then q releases p from its wait loop by assigning 0 to Go[p]. In our algorithm it is also possible that a process $r \neq q$ notices that p is the owner of the CS and hence may try to set Go[p] to 0 by attempting a CAS operation on this variable. Hence, in such cases a different process r, instead of q who captured CS for p, releases p from its wait.

Token: Token is an integer variable supporting read and CAS operations. Token is used to implement a counter so that its values can be used to assign token numbers to processes requesting the CS: in the Try section, a process reads Token to get its token number and then increments Token. The token thus obtained by a process is used for the entirety of its super-passage.

Persistent variables (stored in NVMM)
REGISTRY : A min-array, initially empty.
CSSTATUS $\in \{0,1\} \times \mathcal{P} \times \mathbb{N}$, initially $(0,1,0)$.
$\forall p$, GO$[p]$ is an integer initialized to 0.
$\forall p$, EXITING$[p]$ is a boolean initialized to $false$.
TOKEN is an integer initialized to 1.

`Try Section`
1. **if** EXITING$[p]$ **then go to** `Exit Section` (Line **12**)
2. $tok_p \leftarrow$ GO$[p]$
3. **if** CSSTATUS $== (1, p, *) \wedge tok_p == 0$ **then go to** `Critical Section` (Line **11**)
else if $tok_p \neq 0$ **then go to** Line **7**
4. $tok_p \leftarrow$ TOKEN
5. CAS(TOKEN, $tok_p, tok_p + 1$)
6. GO$[p] \leftarrow tok_p$
7. REGISTRY.`write`$(p, (p, tok_p))$
8. `promote`()
9. **while** GO$[p] \neq 0$
10. **if** ABORTSIGNAL$[p]$ **then go to** `Exit Section` (Line **12**)
`end Try Section`

11. `Critical Section`

`Exit Section`
12. EXITING$[p] \leftarrow true$
13. REGISTRY.`write`$(p, (p, \infty))$
14. $tok_p \leftarrow$ GO$[p]$
15. **if** $tok_p \neq 0$ **then**
16. $(bit_p, peer_p, peertok_p) \leftarrow$ CSSTATUS
17. **if** $bit_p == 0$ **then** CAS(CSSTATUS, $(bit_p, peer_p, peertok_p), (0, p, tok_p)$)
18. GO$[p] \leftarrow 0$
19. $(bit_p, peer_p, peertok_p) \leftarrow$ CSSTATUS
20. **if** $bit_p == 1 \wedge peer_p == p$ **then** CAS(CSSTATUS, $(1, p, peertok_p), (0, p, peertok_p)$)
21. `promote`()
22. EXITING$[p] \leftarrow false$
`end Exit Section`

procedure `promote`()
23. $(bit_p, peer_p, peertok_p) \leftarrow$ CSSTATUS
24. **if** $bit_p == 0$ **then**
25. $(succ_p, succtok_p) \leftarrow$ REGISTRY.`findmin`()
26. **if** $succtok_p \neq \infty$ **then** CAS(CSSTATUS, $(bit_p, peer_p, peertok_p), (1, succ_p, succtok_p)$)
27. $(bit_p, peer_p, peertok_p) \leftarrow$ CSSTATUS
28. **if** $bit_p == 1$ **then** CAS(GO$[peer_p], peertok_p, 0$)
end procedure

Fig. 1. Abortable RME Algorithm. Code for process p.

REGISTRY: REGISTRY is a min-array that has the same purpose as the one in Jayanti and Joshi's [7] work, which we reiterate here for clarity. The min-array, henceforth referred only as REGISTRY, is an array and has n locations, one per process. It supports two operations: `write`() and `findmin`(). REGISTRY.`write`(p, v), when executed by p, sets REGISTRY$[p]$ to v. The operation REGISTRY.`findmin`() returns the minimum value in the array. Like in [7],

REGISTRY acts like a queue by holding the names of processes waiting to enter the CS, and orders them according to their token numbers. p inserts in REGISTRY an element (p, tok_p) (Line **7**), where tok_p holds p's token number (we call this step by p as "registering" its super-passage). When exiting or aborting, p deletes this element (Line **13**) by writing (p, ∞) (we call this step by p as "unregistering" its super-passage). The elements in REGISTRY are ordered according to their token numbers: $(p, t) < (q, t')$ if $t < t'$ or $t = t' \wedge p < q$. Thus, the `findmin`() operation returns (p, t), where p is the process in REGISTRY with the smallest token. If REGISTRY is empty (i.e., every location in the array is empty), `findmin`() returns a value (q, ∞), with some process name q. We require that the two operations satisfy wait-freedom and idempotence, which allows the algorithm to repeatedly execute these operations in presence of a crash. As mentioned in Sect. 4 of [7], the implementation given in Appendix A of [7] does satisfy these properties. We therefore use that implementation in our algorithm. Their implementation of REGISTRY uses read, write, and CAS and is adapted from f-arrays [25].

CSSTATUS: This variable is a record with three fields: $(bit, peer, peertok)$. The first field, bit, is a single bit field denoting whether the CS is occupied or not. A value of 0 in the bit indicates the CS is free and in that case $peer$ denotes the process that last wrote to CSSTATUS while using $peertok$ as the token for its super-passage. If the value of the bit is 1, it indicates the CS is occupied and in that case, $peer$ denotes the name of the process that currently owns the CS and $peertok$ is the token used by the process with name $peer$ for its current super-passage. The operations supported by CSSTATUS are read and CAS.

EXITING$[p]$: This is a boolean variable that supports the read and write operations. p might crash while executing the Exit section, so we use the EXITING$[p]$ variable to remind p that it was executing the Exit section. Hence, EXITING$[p] = true$ indicates that p should be executing the Exit section after restarting from a crash; EXITING$[p] = false$ indicates p is yet to execute the Exit section in the current super-passage.

A Remark on Wrap-Around of Token Numbers. In our algorithm the bit size of the token numbers generated using TOKEN is constrained by the $peertok$ field of CSSTATUS. Assuming a word length of 64-bits, a reasonable assumption on modern multiprocessor systems, we argue as follows that wrap-around of token numbers is not a practical concern. Assume that the system consists of $16,384$ processes, it would therefore need 14 bits to represent each process. Accounting for the bit field from CSSTATUS, we are left with 49 bits to represent a token number. For the token number to wrap around, there must be 2^{49} passages. If there are 2^{20} (a million) passages per second, it would take 17 years for the token number to wrap around. Therefore, wrap-around is not a practical concern.

3.2 Informal Description

In this section we informally describe the working of our algorithm presented in Fig. 1. We first describe how a process p would execute the Try and Exit section in absence of a crash or an abort signal, and then proceed to explain the algorithm if a crash is encountered anywhere or an abort signal is activated.

Crash-Free and Abort-Free Super-Passage. When p wants to enter the CS from the Remainder section, it starts executing the Try section. Lines **1**–**3** perform a check if the preceding passage by p ended in a crash. Our algorithm maintains the invariant that whenever p is starting a super-passage, the following holds about the shared variables: CSStatus $\neq (1, p, *)$ (i.e., CSStatus says that p is not the owner of CS), Exiting$[p] = false$, and Go$[p] = 0$. Therefore, after reading the above shared variables, none of the **if** conditions from Lines **1**–**3** are met, hence, p proceeds execution from Line **4**. At Line **4** p obtains a token for itself and then increments the global counter (Line **5**). It then saves the obtained token into Go$[p]$ (Line **6**) for its own use so that in the event of a crash it does not obtain a different token. Then, at Line **7**, it inserts its name, tagged with its token, into the Registry (i.e., p "registers" its super-passage). If p executes normal steps upto Line **7** in super-passage s before another process q initiates its super-passage s' and p does not receive an abort signal in s, then q does not enter the CS in s' before p first enters the CS in s (this is useful for the FCFS property). After executing Line **7**, p executes the `promote`() procedure (Line **8**) whose job is to capture the CS for the longest waiting process q registered in Registry and inform q that it no longer needs to wait (we describe the procedure in detail shortly). Following this, p waits until it is informed that it no longer needs to wait (Line **9**) all the while simultaneously checking if it received an abort signal (Line **10**). If p reads that AbortSignal$[p] = true$ at Line **10**, it has received the external signal to abort continuing to the CS, hence, it starts executing the Exit section at Line **12**. Upon being informed about its turn to enter the CS (i.e., Go$[p] = 0$), p enters the CS. Note, we assume that starting when p enters the CS and so long as it is executing the CS PC_p remains **11**, except after crashes where for a brief while p executes some code from Try to get back to CS and PC_p changes back to **11**. When p leaves the CS, it first sets a checkpoint at Line **12** signifying that it has started executing the Exit section by writing $true$ into Exiting$[p]$, so that in the event of a crash it comes back to Exit section. At Line **13**, it removes its own name from the Registry (i.e., it "unregisters" its own super-passage). Following that it executes Lines **14**–**15** whose job is to check if p entered Exit section upon receiving an abort signal. Since at present we are considering a super-passage in absence of a crash or an abort signal, p entered the CS on noticing Go$[p] = 0$. Hence, at Line **14** p takes note of the current value of Go$[p]$ and at Line **15** it checks if that value is 0. By the above, the **if** condition at Line **15** is not met, hence, p resumes execution from Line **19**. At Line **19** p reads the current content of CSStatus. Our algorithm maintains the invariant that so long as p has ownership of the CS, CSStatus has the value $(1, p, tok_p)$, where tok_p is the value of p's token for current super-passage.

Therefore, the **if** condition at Line **20** is met, hence p marks the CS as available by performing the CAS at Line **20**. Following this, p tries to capture the CS for the longest waiting process by executing `promote()` (Line **21**). Whether p lets another process into the CS or not, it completes its own super-passage by setting EXITING[p] to $false$ (Line **22**) to indicate that it has completed executing the Exit Section.

Executing `promote()`. We describe the `promote()` procedure as follows. This procedure identifies a process that has been waiting the longest to enter the CS, and lets that process into the CS, if the CS is free. To this purpose, at Line **23**, p reads the contents of CSSTATUS. If the first bit of CSSTATUS is 0, it means the CS is free, therefore, p performs this check at Line **24**. If p finds that the bit is 0, at Line **25** p retrieves the information of the longest waiting process q from REGISTRY (i.e., the name of that process and its token). It then checks if q has a valid token at Line **26** (if an invalid token number denoted by ∞ is received, it means the REGISTRY is empty). If so, then p tries to install q as the new owner of the CS. p does this by performing a CAS at Line **26** that attempts to write into CSSTATUS the information of q. A successful CAS will indicate that q is the one who is going to occupy the CS now. Note, while p is executing Lines **24**–**26** in the manner described above, another process might be executing the same lines and could execute Line **26** before p. This would result in p's CAS at Line **26** to fail. It is also possible that p succeeded in doing the CAS at Line **26**, but crashed immediately. Our algorithm ensures that if p crashes while performing the `promote()` procedure, it will come back to re-execute the procedure from start. And in that re-execution of `promote()`, p will notice that the **if** condition at Line **24** does not meet (although, it had captured the CS for q prior to the crash). In either of the two cases described above, in Lines **27**–**28** p takes the responsibility to "wake" any process that is currently occupying the CS. Hence, at Line **27**, p again reads CSSTATUS to identify the process r whose name was last written into CSSTATUS. If p finds that the first bit of CSSTATUS is 1, it does a CAS on GO[r] to write a 0 (Line **28**). If r was not woken up already, this CAS ensures that it is woken up now. Otherwise, p's CAS is bound to fail because either GO[r] $= 0$ already or r started a new super-passage with a different token in GO[r] (which was written at Line **6**).

Servicing an Abort Signal. Next we describe how p services an abort when it notices that an abort signal has been activated after reading ABORTSIGNAL[p]. p notices the abort signal when it reads ABORTSIGNAL[p] at Line **10** and as a result it starts executing the Exit section at Line **12**. At Line **12** p first sets a checkpoint to signify that it has started executing the Exit section by writing $true$ into EXITING[p], so that in the event of a crash it comes back to Exit section. At Line **13**, it unregisters its own super-passage by removing its name from the REGISTRY. Following that it executes Lines **14**–**15** whose job is to check if p entered Exit section upon receiving an abort signal from the Remainder. Suppose it finds that $tok_p = 0$ (i.e., GO[p] $= 0$), which is possible because some

other process captured the CS for p while p left the Try section for aborting. In that case, it executes the remaining Exit section as described above. This is because the case is as if p entered the CS and then is completing its super-passage by executing Exit section. Assume otherwise that it finds $tok_p \neq 0$ (i.e., GO$[p] \neq 0$). It then reads the contents of CSSTATUS at Line **16** and checks if the CS is free by checking the first bit of CSSTATUS (Line **17**). If it finds that the CS is free, p attempts to update the content of CSSTATUS by writing its own name and token into it by performing a CAS at Line **17**. This updating of the content of CSSTATUS in spite of CS being free might not be intuitive, but it is one of the subtle features of our algorithm which we will explain shortly. p then clears its own token from its GO$[p]$ variable to prepare itself for the next super-passage (Line **18**). Note, when aborting GO$[p]$ might hold a token value p obtained for its current super-passage. If GO$[p]$ is not explicitly wiped, on its next super-passage p might re-use its old token due to Lines **2**–**3**. This will lead to a violation of FCFS property, hence, clearing its own token from GO$[p]$ at Line **18** is important. From Line **19** onwards p executes the Exit section as described above. However, it is important to note that having to do Lines **19**–**20** is another subtle feature of our algorithm, whose discussion we defer for later.

Recovery from a Crash. When p begins a passage after the preceding passage ended in a crash, p starts by reading EXITING$[p]$ at Line **1**. If it finds that EXITING$[p] = true$, then p crashed while executing the Exit section in the previous passage. p could be executing the Exit section in the previous passage as a result of an abort or due to p coming out of CS prior to crash. In any case, p executes the Exit section from Line **12**. Our Exit section is designed to be idempotent, i.e., if p crashes in the middle of Exit section and re-executes it from the start multiple times, then it would appear to take effect once. Hence, it allows us to execute the Exit section from Line **12** after a crash in the Exit section. If EXITING$[p] = false$, then p reads the contents of GO$[p]$ and CSSTATUS (Lines **2**–**3**). If p finds that CSSTATUS $== (1, p, *)$ (i.e., p has ownership of the CS with a certain token) and GO$[p] = 0$, then p has exclusive access to the CS. Hence, p moves to the CS at Line **11**. Otherwise, p checks if $tok_p \neq 0$ at Line 3, which implies that it has obtained a token prior to the crash, stored it in GO$[p]$, but does not have the ownership of CS yet. In that case p goes on to continue with the super-passage from Line **7**, where it starts with registering the super-passage and continuing as described above. If p finds that $tok_p = 0$, it means p is yet to even get a token for itself. In that case p starts from Line **4** as if it started a new super-passage (see description above).

3.3 Subtle Features of the Algorithm

In the description of the algorithm given above, we deferred the discussion of a few subtle features of the algorithm. We discuss those subtle features in this section, namely, (A) Maintaining GO$[p]$ as an integer variable instead of a boolean, (B) why does a process p perform a blind CAS on GO$[peer_p]$ at Line **28**,

(C) updating the content of CSStatus at Line **17** in spite of CS being free, and (D) performing Lines **19**–**20** even when servicing an abort. We demonstrate below the reason behind performing these operations as follows.

The Need for Feature A. In local-spin mutual exclusion algorithms (e.g., [7,16]) it is generally the case that the spin variable is a boolean flag. However, in our algorithm a process spins on an integer for a specific reason which we describe as follows. Suppose a boolean flag was used instead of an integer, the following scenario shows that it would result in violating mutual exclusion property. Suppose process p is in the CS in a configuration where every other process is in the Remainder section. A process q from the Remainder section needs access to the CS and hence executes the Try section and eventually makes a call to the `promote`() procedure at Line **8** in the Try. q executes the `promote`() procedure upto but not including Line **27**, where it is supposed to wake up the current owner of the CS. At this point p comes out of the CS and starts executing the Exit section so that it eventually calls the `promote`() procedure. p executes Lines **23**–**27** to make q the new owner of CS, reads the content of CSStatus into bit_p $(= 1)$ and $peer_p$ $(= q)$ at Line **27**, and at Line **28** p stops (i.e., just before letting q into the CS). At this point q resumes execution from Line **27**, notes that it itself is now the owner of the CS and hence sets its own Go flag to *true* at Line **28**. Therefore, q enters the CS, completes executing it, and then eventually finishes its super-passage. Now assume that another process r executes the Try section, finds the CS to be free and hence puts itself into the CS. After this q again decides to enter the CS, hence it starts a new super-passage. It executes the Try section to find the CS to be occupied by r, hence it waits for its turn by looping at Lines **9**–**10**. At this moment, p which had stopped at Line **28** resumes its execution and since it read the first bit of CSStatus to be 1 with $peer_p = q$, p lets q into the CS by writing *true* into Go$[q]$. q reads the Go$[q]$ flag and enters the CS. Since in the next configuration q and r are in the CS, mutual exclusion is violated. To avoid this issue we use the Go flag as an integer variable so that Go$[q]$ either stores 0 or the token q uses for its current super-passage. This way when p resumes later as described in the scenario above, it tries to CAS into Go$[q]$ with a token that q used in its earlier super-passage. Such a CAS is bound to fail in the scenario above since Go$[q]$ would use a new token in the next super-passage.

The Need for Feature B. Our algorithm is based on a previous RME algorithm by Jayanti and Joshi [7] in which the delegation of ownership of the CS to a process and writing to the spin variable of that process is done by a single process. However, in our algorithm from this paper a process p performs a blind CAS on Go$[peer_p]$ at Line **28**, if it finds that a process $peer_p$ occupies the CS, although p might not have made $peer_p$ the owner of the CS. The reason behind designing the algorithm this way is as follows. Suppose a process p is executing the `promote`() procedure such that it executes Lines **23**–**27**, where it is makes a process q the owner of the CS. However, p crashes just before writing 0 to Go$[q]$ at Line **28**. When p restarts, it cannot tell by reading any of the shared

variables if it was the one who made q the owner of the CS (unlike in [7], where reading the CSOwner variable would give this information). Hence, regardless of whether p made q the owner of CS or not, p assumes the responsibility of waking q from its wait loop and performs the CAS at Line **28**.

The Need for Feature C. When p is aborting from its super-passage by executing the Exit section, at Line **17** it performs a CAS on CSStatus to declare that the CS is free in spite of noticing that the CS is free and the first bit of CSStatus is 0 already. As we describe below, if this step is not performed, p could be made the owner of CS even though p has aborted its super-passage and is in the Remainder section. Assume that the CAS at Line **17** is not performed and the **if** block at Lines **15**–**18** contains only one step to write the value 0 to Go[p]. Assume there is a process q in the CS and all other processes including p are in the Remainder section. p decides to acquire access to CS, therefore, it executes the Try section and waits for its turn by looping at Lines **9**–**10**. q then comes out of the CS and starts executing the Exit section. q executes the Exit section all the way calling the `promote`() procedure and right upto Line **25** and stops at Line **26**. Therefore, the value of CSStatus $= (0, q, tok_q)$ and q has read p's entry from Registry such that q is enabled to perform the CAS at Line **26** and would succeed in doing so. At this moment p decides to abort its super-passage and hence it starts executing the Exit section. p first removes its entry from Registry at Line **13** (therefore, Registry becomes empty now). Since p was not woken up by q to go into the CS, Go[p] $= tok_p$, hence p writes 0 to Go[p] at Line **18**. The **if** condition at Line **20** is not met (since CSStatus $= (0, q, tok_q)$), therefore, p calls the `promote`() procedure at Line **21**. Inside the call to `promote`(), p finds that the Registry is empty at Line **25**, and the **if** condition at Line **28** is not met because CSStatus $= (0, q, tok_q)$. Hence, p completes `promote`() without modifying any shared variable, goes back to Exit where it writes $false$ to Exiting[p] and then goes back to Remainder. At this point q resumes execution and performs the CAS at Line **26**. Since CSStatus is unchanged in the meantime, q succeeds in doing the CAS thus making p the owner of the CS. This situation is undesirable because p is in the Remainder section and is made the owner of the CS. If instead the CAS at Line **18** is performed by p, then q's CAS at Line **26** would not succeed and hence the undesirable situation is avoided.

The Need for Feature D. When p aborts from its super-passage, it is possible that p is made the owner of the CS even though it is aborting. In such a scenario it is necessary that p relinquishes its ownership of the CS and continues with the abort. We demonstrate below (with an argument similar to the above) that not performing Lines **19**–**20** when p is aborting leads to an undesirable scenario. Like above, assume there is a process q in the CS and all other processes including p are in the Remainder section. p decides to acquire access to CS, therefore, it executes the Try section and waits for its turn by looping at Lines **9**–**10**. q then comes out of the CS and starts executing the Exit section. q executes the

Exit section all the way calling the `promote`() procedure and right upto Line **27** and stops at Line **28**. Therefore, the value of CSStatus $= (1, p, tok_p)$ and q is enabled to perform the CAS at Line **28**. At this moment p decides to abort its super-passage and hence it starts executing the Exit section. p first removes its entry from Registry at Line **13** (therefore, Registry becomes empty now). Since p was not woken up by q to go into the CS, Go$[p] = tok_p$, hence p writes 0 to Go$[p]$ (the **if** condition at Line **17** fails due to the value of CSStatus). By our assumption p does not perform Lines **19**–**20** but executes `promote`() procedure at Line **21** where it sets its own Go$[p]$ variable to 0 at Line **28** (because CSStatus $= (1, p, tok_p)$). It then goes back to the Remainder after updating Exiting$[p]$. At this moment q starts taking steps and is unsuccessful at the CAS at Line **28**. q then goes back to the Remainder after updating Exiting$[q]$. It follows that CSStatus $= (1, p, tok_p)$, where tok_p is the token p used in previous super-passage, although p is in Remainder. Had p performed Lines **19**–**20**, it would have updated CSStatus to $(0, p, tok_p)$ denoting that the CS should be kept free.

3.4 RMR Complexity

We discuss the RMR complexity a process incurs per passage as follows. As described in Lemma 2 of Jayanti and Joshi's work [7], the Registry.`write`() operation incurs $O(\log n)$ RMRs and the Registry.`findmin`() operation incurs $O(1)$ RMRs on both CC and DSM machines. On DSM machines, where we host the variables Go$[p]$ and Exiting$[p]$ in p's memory partition, p's operations on these variables incur zero RMRs. Therefore, on DSM machines our algorithm incurs $O(\log n)$ RMR per passage.

On CC machines, similarly, it would be tempting to believe that all these other operations incur constant RMRs, however, it depends on the way the cache is managed in the machine. Therefore, for this discussion we divide CC machines into two categories: (1) *strict* CC machines and (2) *relaxed* CC machines, which we describe below and discuss how the RMR is calculated in each category. On *strict* CC machines, a process will incur an RMR when a failed CAS is performed on a variable it is about to read even though the process had a cached copy of the variable prior to the CAS. On *relaxed* CC machines a process will not incur an RMR if a CAS operation fails on a variable it is about to read. Note, this behavior of incurring RMRs on CC machines is in addition to our discussion from Sect. 2. Therefore, the RMR complexity remains $O(\log n)$ on the relaxed CC machines (similar to DSM machines), but shoots up to $O(n)$ on strict CC machines for the following reason. Assume a process p is waiting to enter the CS at Line **9**, $n/2 - 1$ processes are in the Remainder section and there are $n/2$ processes that are about to execute Line **28** to perform a CAS on Go$[p]$. Out of these processes only one performs the CAS, goes back to the Remainder while letting p into the CS due to the CAS, and the rest $n/2 - 1$ process have still not executed Line **28**. Now there are $n/2$ processes in the Remainder section, p in the CS, and $n/2 - 1$ processes that are about to execute Line **28** to perform a CAS on Go$[p]$. Assume p completes the CS, executes the Exit section, and goes

back to Remainder section. Meanwhile the $n/2$ processes from Remainder come out of the Remainder for a new passage and queue up. p then queues up behind these processes with a new token and starts waiting at Line **9** to enter the CS. At this moment those $n/2 - 1$ processes that had stopped at Line **28** execute a step causing a failed CAS. This causes p to incur an RMR for every failed CAS incurring $O(n)$ RMRs. Therefore, on strict CC machines our algorithm incurs $O(n)$ RMRs per passage. To summarize, the algorithm incurs $O(\log n)$ RMRs per passage on DSM and relaxed CC machines, and $O(n)$ RMRs per passage on strict CC machines. Likewise, it incurs $O(f + \log n)$ RMRs per super-passage on DSM and relaxed CC machines, and $O(f + n)$ RMRs per super-passage on strict CC machines.

3.5 Main Theorem

The theorem below summarizes the result of our paper.

Theorem 1. *The algorithm in Fig. 1 is an abortable recoverable mutual exclusion algorithm for n processes and satisfies properties P1-P8 described in Sect. 2. The algorithm incurs $O(\log n)$ RMRs per passage on DSM and relaxed CC machines and $O(n)$ RMRs per passage on strict CC machines.*

References

1. Raoux, S., et al.: Phase-change random access memory: a scalable technology. IBM J. Res. Dev. **52**(4/5), 465 (2008)
2. Strukov, D.B., Snider, G.S., Stewart, D.R., Williams, R.S.: The missing memristor found. Nature **453**(7191), 80 (2008)
3. Tehrani, S., et al.: Magnetoresistive random access memory using magnetic tunnel junctions. Proce. IEEE **91**(5), 703–714 (2003)
4. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Commun. ACM **8**(9), 569 (1965)
5. Golab, W., Ramaraju, A.: Recoverable mutual exclusion: [extended abstract]. In: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, pp. 65–74. ACM, New York (2016)
6. Golab, W., Hendler, D.: Recoverable mutual exclusion in sub-logarithmic time. In: Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, pp. 211–220. ACM, New York (2017)
7. Jayanti, P., Joshi, A.: Recoverable FCFS mutual exclusion with wait-free recovery. In: 31st International Symposium on Distributed Computing, DISC 2017, pp. 30:1–30:15 (2017)
8. Golab, W., Hendler, D.: Recoverable mutual exclusion under system-wide failures. In: Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, pp. 17–26. ACM, New York (2018)
9. Jayanti, P., Jayanti, S., Joshi, A.: Optimal recoverable mutual exclusion using only FASAS. In: Podelski, A., Taïani, F. (eds.) NETYS 2018. LNCS, vol. 11028, pp. 191–206. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-05529-5_13

10. Jayanti, P., Jayanti, S., Joshi, A.: Recoverable Mutual Exclusion with Sublogarithmic RMR Complexity on CC and DSM machines. In: Accepted for publication in PODC 2019 (2019)
11. Ramaraju, A.: RGLock: recoverable mutual exclusion for non-volatile main memory systems. Master's thesis, University of Waterloo (2015)
12. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Commun. ACM **17**(8), 453–455 (1974)
13. Attiya, H., Ben-Baruch, O., Hendler, D.: Nesting-safe recoverable linearizability: modular constructions for non-volatile memory. In: Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, pp. 7–16. ACM (2018)
14. Scott, M.L.: Non-blocking timeout in scalable queue-based spin locks. In: Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing, PODC 2002, pp. 31–40. ACM, New York (2002)
15. Scott, M.L., Scherer, W.N.: Scalable queue-based spin locks with timeout. In: Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, PPoPP 2001, pp. 44–52. ACM, New York (2001)
16. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. **9**(1), 21–65 (1991)
17. Craig, T.S.: Building FIFO and priority-queuing spin locks from atomic swap. Technical report TR-93-02-02, Department of Computer Science, University of Washington, February 1993
18. Jayanti, P.: Adaptive and efficient abortable mutual exclusion. In: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, PODC 2003, pp. 295–304. ACM, New York (2003)
19. Attiya, H., Hendler, D., Woelfel, P.: Tight RMR lower bounds for mutual exclusion and other problems. In: Proceedings of the Fortieth ACM Symposium on Theory of Computing, STOC 2008, pp. 217–226. ACM, New York (2008)
20. Lee, H.: Fast local-spin abortable mutual exclusion with bounded space. In: Lu, C., Masuzawa, T., Mosbah, M. (eds.) OPODIS 2010. LNCS, vol. 6490, pp. 364–379. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17653-1_27
21. Alon, A., Morrison, A.: Deterministic abortable mutual exclusion with sublogarithmic adaptive RMR complexity. In: Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, pp. 27–36. ACM, New York (2018)
22. Jayanti, P., Jayanti, S.V.: Constant amortized RMR complexity deterministic abortable mutual exclusion algorithm for CC and DSM models. In: Accepted for publication in PODC 2019 (2019)
23. Pareek, A., Woelfel, P.: RMR-efficient randomized abortable mutual exclusion. In: Aguilera, M.K. (ed.) DISC 2012. LNCS, vol. 7611, pp. 267–281. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33651-5_19
24. Giakkoupis, G., Woelfel, P.: Randomized abortable mutual exclusion with constant amortized RMR complexity on the CC model. In: Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, pp. 221–229. ACM, New York (2017)
25. Jayanti, P.: f-arrays: implementation and applications. In: Proceedings of the Twenty-First Symposium on Principles of Distributed Computing, PODC 2002, pp. 270–279. ACM, New York (2002)