Computing (2022) 104:2225–2252
https://doi.org/10.1007/s00607-022-01105-1

SPECIAL ISSUE ARTICLE

Recoverable mutual exclusion with abortability

Prasad Jayanti[1] · **Anup Joshi**[1]

Received: 7 December 2020 / Accepted: 5 July 2022 / Published online: 18 August 2022

Abstract

Recent advances in non-volatile main memory (NVM) technology have spurred research on algorithms that are resilient to intermittent failures that cause processes to crash and subsequently restart. In this paper we present a Recoverable Mutual Exclusion (RME) algorithm that supports abortability. Our algorithm guarantees FCFS and a strong liveness property: processes do not starve even in runs consisting of infinitely many crashes, provided that a process crashes at most a finite number of times in each of its attempts. On DSM and Relaxed-CC multiprocessors, a process incurs $O(\min(k, \log n))$ RMRs in a passage and $O(f + \min(k, \log n))$ RMRs in an attempt, where n is the number of processes that the algorithm is designed for, k is the point contention of the passage or the attempt, and f is the number of times that p crashes during the attempt. On a Strict CC multiprocessor, the passage and attempt complexities are $O(n)$ and $O(f + n)$, respectively. Our algorithm uses only the read, write, and CAS operations, which are commonly supported by multiprocessors. Attiya, Hendler, and Woelfel proved that, with any mutual exclusion algorithm, a process incurs at least $\Omega(\log n)$ RMRs in a passage, if the algorithm uses only the read, write, and CAS operations (in: Proc. of the Fortieth ACM Symposium on Theory of Computing, New York, NY, USA, 2008). This lower bound implies that the worst-case RMR complexity of our algorithm is optimal for the DSM and Relaxed CC multiprocessors. This paper is an expanded version of our conference paper as reported by Jayanti and Joshi (in: Atig and Schwarzmann (eds) Networked Systems. Springer International Publishing, Cham, 2019), which presented the first Recoverable Mutual Exclusion (RME) algorithm that supports abortability. This algorithm from our conference paper (in: Atig and Schwarzmann (eds) Networked Systems. Springer International Publishing,

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.

✉ Anup Joshi
anup.s.joshi.gr@dartmouth.edu

Prasad Jayanti
prasad.jayanti@dartmouth.edu

[1] Dartmouth College, Hanover, NH 03755, USA

Cham, 2019) admits starvation when there are infinitely many aborts in a run. In this paper, we fix this shortcoming and prove the algorithm's properties by identifying an inductive invariant.

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

Mathematics Subject Classification 68W15

1 Introduction

Recent advances in non-volatile main memory (NVM) technology [11, 26, 30, 31] have spurred research on designing algorithms that are resilient to process crashes. NVM is byte-addressable, so it replaces main memory, directly interfacing with the processor. This development is exciting because, if a process crashes and subsequently restarts, there is now hope that the process can somehow recover from the crash by consulting the contents of the NVM and resume its computation.

To leverage this advantage given by the NVM, there has been keen interest in reexamining the important distributed computing problems for which algorithms were designed in the past for the traditional (crash-free) model of an asynchronous shared memory multiprocessor. The goal is to design new algorithms that guarantee good properties even if processes crash at arbitrary points in the execution of the algorithm and subsequently restart and attempt to resume the execution of the algorithm. The challenge in designing such "recoverable" algorithms stems from the fact that when a process crashes, even though the shared variables that are stored in the NVM are unaffected, the crash wipes out the contents of the process' cache and CPU registers, including its program counter. So, when the process subsequently restarts, it can't have a precise knowledge of exactly where it crashed. For instance, if the last instruction that a process executes before a crash is a compare&swap (CAS) on a shared variable X, when it subsequently restarts, it can't tell whether the crash occurred just before or just after executing the CAS instruction and, if it did crash after the CAS, it won't know the response of the CAS (because the crash wipes out the register the CAS's response went into). The "recover" method, which a process is expected to execute when it restarts, has the arduous task of ensuring that the process can still somehow resume the execution of the algorithm seamlessly.

The mutual exclusion problem, formulated to enable multiple processes to share a resource that supports only one process at a time [6], has been thoroughly studied for over half a century for the traditional (crash-free) model, but its exploration is fairly recent for the crash-restart model, where processes crash intermittently and restart subsequently. In the traditional version of the problem, each process p is initially in the "remainder" section. When p becomes interested in acquiring the resource, it executes the $\texttt{try}_p()$ method; and when this method completes, p is in the "critical section" (CS). To give up the CS, p invokes the $\texttt{exit}_p()$ method; and when this method completes, p is back in the remainder section. An algorithm to this problem

specifies the code for the try and exit methods so that at most one process is in the CS at any time and other desirable properties (such as starvation freedom, bounded exit, and First-Come-First-served, or FCFS) are also satisfied. Golab and Ramaraju were the first to reformulate this problem for the "crash-restart model", where a process can crash at any time and subsequently restart [10]. In the reformulated problem, which they called *Recoverable Mutual Exclusion* (RME), if p crashes while in try, CS, or exit, p's cache and registers (aka local variables) are wiped out and p returns to the remainder section (i.e., crash resets p's program counter to its remainder section). When p restarts after a crash, it is required to invoke a new method, named $\texttt{recover}_p()$, whose job is to "repair" the adverse effects of the crash and send p to where it belongs. In particular, if p crashed while in the CS, $\texttt{recover}_p()$ puts p back in the CS (by returning IN_CS). On the other hand, if p crashed while executing $\texttt{try}_p()$, $\texttt{recover}_p()$ has a choice—it can either roll p back to the Remainder (by returning IN_REM) or put it in the CS (by returning IN_CS), but of course without violating Mutual Exclusion. Similarly, if p crashed while executing $\texttt{exit}_p()$, $\texttt{recover}_p()$ has a choice of returning either IN_REM or IN_CS.

Golab and Ramaraju made a crucial observation that if p crashes while in the CS, then no other process should be allowed into the CS until p restarts and reenters the CS. This *Critical Section Reentry* (CSR) requirement was strengthed by Jayanti and Joshi's *Bounded CSR* requirement: if p crashes while in the CS, when p subsequently restarts and executes the recover method, the recover method should put p back into the CS in a bounded number of its own steps [17]. There has been a flurry of research on RME algorithms in the recent years [3, 5, 8–10, 14, 15, 17, 18, 20].

Orthogonal to this development of recoverable algorithms, motivated by the needs of real time systems and database systems, Scott and Scherer advocated the need for mutual exclusion algorithms to support the "abort" feature, whereby a process in the try section can quickly quit the algorithm, if it so desires [28]. More specifically, if p receives an abort signal from the environment while executing the try method, the try method should complete in a bounded number of p's steps and either launch p into the CS or send p back to the remainder section.[1]

In the past two decades, there has been a lot of research on abortable mutual exclusion algorithms for the traditional (crash-free) model.

The possibility of crashes, together with the CSR requirement, renders abortability even more important in the crash-restart model, yet there have been no *abortable* recoverable algorithms until the conference publication of the algorithm in this submission [18]. There has since been one more algorithm, by Katzan and Morrison [20], and we will soon compare the two algorithms.

1.1 RMR complexity

Remote Memory Reference (RMR) complexity is the standard complexity metric used for comparing mutual exclusion algorithms, so we explain it here. This metric is

[1] Of course, p might receive an abort signal even while executing the exit section. However, because of the standard "bounded exit" requirement that every process completes the exit method in a bounded number of its steps, no special intervention is necessary to handle an abort-signal during the exit method.

explained for the two prevalent models of multiprocessors—*Distributed Shared Memory* (DSM) and *Cache-Coherent* (CC) multiprocessors—as follows. In DSM, shared memory is partitioned into n parts, one per process, and each shared variable resides in exactly one of the n parts. A step in which a process p executes an instruction on a shared variable X is considered an RMR if and only if X is not in p's part of the partition.

In CC, the shared memory is remote to all processes, but every process has a local cache. A step in which a process p executes an instruction op on a shared variable X is considered an RMR if and only if op is *read* and X is not in p's cache, or op is any non-read operation (such as a *write* or *CAS*). If p reads X when X is not present in p's cache, X is brought into p's cache. If a process q performs a non-read operation op while X is in p's cache, X's copy in p's cache is deleted in the *Strict CC model*, but in the *Relaxed CC model* it is deleted only if op changes X's value. Thus, if X is in p's cache and q performs an unsuccessful CAS on X, then X continues to remain in p's cache in the relaxed CC model.[2]

A *passage* of a process p starts when p leaves the remainder section and completes at the earliest subsequent time when p returns to the remainder (note that p returns to the remainder either because of a crash or because of a normal return from try, exit or recover methods). An *attempt* of p starts when p leaves the remainder and completes at the earliest subsequent time when p returns to the remainder "normally," i.e., not because of a crash.[3] Note that each attempt includes one or more passages.

The *RMR complexity of a passage* (respectively, *attempt*) of a process p is the number of RMRs that p incurs in that passage (respectively, attempt).

1.2 Adaptive complexity

A process is *active* if it is in the CS, or executing the try, exit, or recover methods, or crashed while in try, CS, exit, or recover and has not subsequently invoked the recover method. The *point contention* at any time t is the number of active processes at t. The point contention of a passage (respectively, attempt) is the maximum point contention at any time in that passage (respectively, attempt). An algorithm is *adaptive* if the RMR complexity of each passage (or attempt) of a process p is bounded by a constant (independent of n) whenever the point-contention is bounded by a constant.

1.3 Our contribution

We design the first abortable RME algorithm, based on the ideas underlying two earlier algorithms—one that is recoverable but not abortable [17] and another that

[2] We are not aware of any real machines that satisfy the Relaxed CC model, but it might be possible to relate this model to the Strict CC model. In particular, we are currently investigating whether algorithms designed for the Relaxed CC model can be automatically transformed for the Strict CC model with only a constant factor blow up in the RMR complexity.

[3] An "attempt" [17] is similar to, but not the same as, "super-passage" [10]. The need for this distinction arises from the difference in the models: when a process p crashes in the try section and subsequently executes $\texttt{recover}_p()$, the recover method always puts p back in the try section in the model of [10], but it may put p in any of try, remainder, or critical sections in our model and the model of [17].

is abortable but not recoverable [13]. Our algorithm guarantees FCFS and a strong liveness property: processes do not starve even in runs consisting of infinitely many crashes, provided that a process crashes at most a finite number of times in each of its attempts. It also satisfies bounded exit, bounded CSR, and bounded abort.

The algorithm has adaptive, logarithmic worst-case RMR complexity. On DSM and Relaxed CC multiprocessors, a process p incurs $O(\min(k, \log n))$ RMRs in a passage and $O(f + \min(k, \log n))$ RMRs in an attempt, where n is the number of processes that the algorithm is designed for, k is the point contention of the passage or the attempt, and f is the number of times that p crashes during the attempt. On a Strict CC multiprocessor, the passage and attempt complexities are $O(n)$ and $O(f + n)$, respectively.

The algorithm's space complexity—the number of words of memory used—is $O(n)$. It is assumed that a memory word is wide enough to store a process name and an unbounded sequence number, which is incremented at most once by a process in each of its passages. Thus, on a standard 64-bit architecture, if we set aside 16 bits for the process name (to accommodate 64K processes), we would have 48 bits for the sequence number, which means that about 256 trillion passages have to occur before the sequence number wraps around.

Our algorithm uses only the read, write, and CAS operations, which are commonly supported by multiprocessors. Attiya, Hendler, and Woelfel proved that, with any mutual exclusion algorithm (even if the algorithm does not have to satisfy recoverability or abortability), a process incurs at least $\Omega(\log n)$ RMRs in a passage, if the algorithm uses only the read, write, and CAS operations [2]. This lower bound implies that the worst-case RMR complexity of our algorithm is optimal for the DSM and Relaxed CC multiprocessors.

1.4 Comparison to Katzan and Morrison's algorithm

To the best of our knowledge, there is only one other abortable RME algorithm, published recently by Katzan and Morrison [20]. By using the fetch&add instruction, in addition to CAS, they breach Attiya, Hendler, and Woelfel's lower bound and achieve sublogarithmic complexity: a process incurs at most $O(\min(k, \log n / \log\log n)$ RMRs in a passage and $O(f + \min(k, \log n / \log\log n))$ in an attempt. Furthermore, they achieve these bounds for even the Strict CC multiprocessor, and without the use of unbounded variables. The space complexity of their algorithm is $O(n \log^2 n / \log\log n)$.

To compare, their algorithm is better than ours in RMR complexity (by a factor of $\log\log n$) and worse than ours in space complexity (by a factor of $\log^2 n / \log\log n$). The more significant advantages of our algorithm are that it satisfies FCFS and a strong liveness property: processes do not starve even in runs consisting of infinitely many crashes, provided that each process crashes at most a finite number of times in each of its attempts. In contrast, Katzan and Morrison's algorithm guarantees starvation-freedom only in runs where the total number of crashes over all processes is finite.

On the downside, unlike Katzan and Morrison's algorithm, our algorithm employs variables that store a pair consisting of a process name and an unbounded counter

whose value is incremented at most once per passage. This requirement might not be a limitation in practice because, on 64-bit machines, if 48 bits of a word are reserved for the counter, the counter wraps around only after about 256 trillion passages.

Finally, Katzan and Morrison correctly point out a shortcoming in our conference paper: our algorithm there admits starvation if there are infinitely many aborts in a run. The algorithm has been revised to eliminate this shortcoming.

1.5 Related research

All of the works on RME prior to the conference version of our paper [18] has focused on designing algorithms that do not provide abortability as a capability. Golab and Ramaraju [10] formalized the RME problem and designed several algorithms by adapting traditional mutual exclusion algorithms. Ramaraju [25], Jayanti and Joshi [17], and Jayanti, Jayanti, and Joshi [14] designed RME algorithms that support the First-Come-First-Served property [21]. Golab and Hendler [8] presented an algorithm that has sub-logarithmic RMR complexity on CC machines. Jayanti, Jayanti, and Joshi [15] presented a unified algorithm that has a sub-logarithmic RMR complexity on both CC and DSM machines. In another work, Golab and Hendler [9] presented an algorithm that has the ideal $O(1)$ passage complexity, but this result assumes that *all* processes in the system crash *simultaneously*. Recently, Dhoked and Mittal [5] present an RME algorithm whose RMR complexity adapts to the number of crashes, and Chan and Woelfel [3] present an algorithm which has an O(1) amortized RMR complexity. Katzan and Morrison [20] gave an abortable RME algorithm that incurs sub-logarithmic RMR on CC and DSM machines.

When it comes to abortability for classical mutual exclusion problem, Scott [27] and Scott and Scherer [29] designed abortable algorithms that build on the queue-based algorithms [4, 23]. Jayanti [13] designed an algorithm based on read, write, and comparison primitives having $O(\log n)$ RMR complexity which is also optimal [2]. Lee [22] designed an algorithm for CC machines that uses the Fetch-and-Add and Fetch-and-Store primitives. Alon and Morrison [1] 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 [16] 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 [24] give a sublogarithmic RMR complexity randomized algorithm and Giakkoupis and Woelfel [7] give an $O(1)$ expected amortized RMR complexity randomized algorithm.

1.6 The differences with the conference publication

The previous version of this paper appeared in NETYS '19, but the algorithm there admits starvation if there are infinitely many aborts in a run. The algorithm here eliminates this shortcoming. Furthermore, the FCFS and strong starvation freedom

properties are hard to prove. Their proofs are presented here, but they were missing in the conference version.

2 Modeling an Abortable RME algorithm and its runs

An *Abortable RME algorithm* is described by the following elements.

- A set $\mathcal{P}$ of processes that may execute the algorithm. Each process $p \in \mathcal{P}$ has a set of registers, including a *program counter*, denoted PC_p, which points to an instruction in p's code.
- A set $\mathcal{X}$ of variables, which includes a Boolean variable ABORTSIGNAL$[p]$, for each $p \in \mathcal{P}$. No process except p can invoke any operation on ABORTSIGNAL$[p]$, and p can only invoke a read operation on ABORTSIGNAL$[p]$.
  Intuitively, the "environment" sets ABORTSIGNAL$[p]$ to *true* when it wishes to communicate to p that it should abort its attempt to acquire the CS and return to the remainder section.
- An assignment of initial values to variables in $\mathcal{X}$.
- A set OP of operations that each variable in $\mathcal{X} - \{\text{ABORTSIGNAL}[p] \mid p \in \mathcal{P}\}$ supports.
  For the algorithm in this paper, $OP = \{read, write, CAS\}$, where CAS$(X, r, s)$, when executed by a process p (and X is a variable and r, s are p's registers), compares the values of X and r; if they are equal, the operation writes in X the value in s and returns *true*; otherwise, the operation returns *false*, leaving X unchanged.
- Δ is a partition of $\mathcal{X}$ into $|\mathcal{P}|$ sets, named $\Delta(p)$, for each $p \in \mathcal{P}$. Intuitively, $\Delta(p)$ is the set of variables that reside locally at process p's part of the shared memory on a DSM machine. ($\Delta(p)$ has no relevance on a CC machine.)
- A set $\mathcal{M}$ of methods, which includes three methods per process $p \in \mathcal{P}$, named $\texttt{try}_p()$, $\texttt{exit}_p()$, and $\texttt{recover}_p()$, such that:
  - In any instruction of any method, at most one operation in OP is performed and it is performed on a single variable from $\mathcal{X}$.
  - The methods $\texttt{try}_p()$ and $\texttt{recover}_p()$ return a value from {IN_CS, IN_REM}, and $\texttt{exit}_p()$ has no return value.
  - None of $\texttt{try}_p()$, $\texttt{exit}_p()$, or $\texttt{recover}_p()$ calls itself or the other two. (This assumption simplifies the model, but is not limiting in any way because it does not preclude the use of helper methods each of which can call itself or the other helper methods.)

For each process $p \in \mathcal{P}$, we model p's code outside of the methods in $\mathcal{M}$ to consist of two disjoint sections, named $\texttt{remainder}_p()$ and $\texttt{cs}_p()$. Furthermore, we introduce the following *abstract* variables, which are not in $\mathcal{X}$ and not accessed by the methods in $\mathcal{M}$, but are helpful in defining the problem.

- $status_p \in \{good, recover\text{-}from\text{-}try, recover\text{-}from\text{-}cs, recover\text{-}from\text{-}exit, recover\text{-}from\text{-}rem\}$.

Informally, $status_p$ models p's "recovery status". If $status_p \neq good$, it means that either p has crashed and not yet restarted or p has restarted and invoked $\texttt{recover}_p()$ but has not yet completed $\texttt{recover}_p()$. The value of $status_p$ reveals the section of code where p most recently crashed.
- CACHE_p holds a set of pairs of the form (X, v), where $X \in \mathcal{X}$ and v is a value. Informally, if (X, v) is present in the cache, X is in p's cache and v is its current value. This abstract variable helps define what operations count as *remote memory references* (RMR) on CC machines.

Definition 1 *(State, Configuration, Initial Configuration)*

- A *state* of a process p is a function that assigns a value to each of p's registers, including PC_p, and a value to each of $status_p$, $\text{ABORTSIGNAL}[p]$, and CACHE_p.
- A *configuration* is a function that assigns a state to each process in $\mathcal{P}$ and a value to each variable in $\mathcal{X}$. (Intuitively, a configuration is a snapshot of the states of processes and values of variables at a point in time.)
- An *initial configuration* is a configuration where, for each $p \in \mathcal{P}$, $PC_p = \texttt{remainder}_p()$, $status_p = good$, $\text{ABORTSIGNAL}[p] = false$, and $\text{CACHE}_p = \emptyset$; and, for each $X \in \mathcal{X}$, X has its initial value.

Definition 2 *(Run)* A *run* is a finite sequence $C_0, \alpha_1, C_1, \alpha_2, C_2, \ldots \alpha_k, C_k$, or an infinite sequence $C_0, \alpha_1, C_1, \alpha_2, C_2, \ldots$ such that:

1. C_0 is an initial configuration and, for each i, C_i is a configuration and α_i is either $(p, normal)$ or $(p, crash)$, for some $p \in \mathcal{P}$.
   We call each triple (C_{i-1}, α_i, C_i) a *step*; it is a *normal step of p* if $\alpha_i = (p, normal)$, and a *crash step of p* if $\alpha_i = (p, crash)$.
2. For each normal step $(C_{i-1}, (p, normal), C_i)$, C_i is the configuration that results when p executes an enabled instruction of its code, explained as follows:
   - If $PC_p = \texttt{remainder}_p()$ and $status_p = good$ in C_{i-1}, then p invokes either $\texttt{try}_p()$ or $\texttt{recover}_p()$.
   - If $PC_p = \texttt{remainder}_p()$ and $status_p \neq good$ in C_{i-1}, then p invokes $\texttt{recover}_p()$.
   - If $PC_p = \texttt{cs}_p()$, then p invokes $\texttt{exit}_p()$.
   - Otherwise, p executes the instruction that PC_p points to in C_{i-1}.
     If this instruction returns IN_CS (resp., IN_REM), PC_p is set to $\texttt{cs}_p()$ (resp., $\texttt{remainder}_p()$).
     If the instruction causes p to return from $\texttt{recover}_p()$, $status_p$ is set to $good$ in C_i.
     If p performs a read on X and X is not present in CACHE_p in C_{i-1}, then (X, v) is inserted in CACHE_p, where v is X's value in C_{i-1}.
     In the Strict-CC model, if p performs a non-read operation on X, for all $q \in \mathcal{P}$, if CACHE_q contains a pair of the form (X, v), it is removed from CACHE_q. In the Relaxed-CC model, this removal happens only if p's non-read operation on X changes X's value. (In both models, since $(X, *)$ is removed from every process' cache anytime the value of X changes, at any time a process' cache contains at most one pair with X as its first component.)

3. For each crash step $(C_{i-1}, (p, \mathit{crash}), C_i)$, we have:
   - In C_i, PC_p is set to $\texttt{remainder}_p()$ and all other registers of p are set to arbitrary values, and CACHE_p is set to $\emptyset$.
   - If $\mathit{status}_p \neq \mathit{good}$ in C_{i-1}, then status_p remains unchanged in C_i. Otherwise, if (in C_{i-1}) p is in $\texttt{try}_p()$ (respectively, $\texttt{cs}_p()$, $\texttt{exit}_p()$, or $\texttt{recover}_p()$), then status_p is set in C_i to *recover-from-try* (respectively, *recover-from-cs*, *recover-from-exit*, or *recover-from-rem*).

Liveness of the algorithm, which guarantees that processes don't wait forever, can be realized only if the underlying model assures that every crashed process eventually restarts, no process stays in the CS forever, and no process permanently ceases to take steps when it is outside the Remainder section. Hence, "fair" runs of the algorithm where these assurances are kept are of interest, as captured by the next definition.

Definition 3 *(Fair run)* A run $R = C_0, \alpha_1, C_1, \alpha_2, C_2, \ldots$ is *fair* if and only if either R is finite or, for all configurations C_i and for all processes $p \in \mathcal{P}$, the following condition is satisfied: unless $PC_p = \texttt{remainder}_p()$ and $\mathit{status}_p = \mathit{good}$ in C_i, p has a step in the suffix of R from C_i.

Definition 4 *(Passage and Attempt)*
- A *passage* of a process p is a contiguous sequence σ of steps in a run such that p leaves $\texttt{remainder}_p()$ in the first step of σ and the last step of σ is the earliest subsequent step in the run where p reenters $\texttt{remainder}_p()$ (either because p crashes or because p's method returns IN_REM).
- An *attempt* of a process p is a maximal contiguous sequence σ of steps in a run such that p leaves $\texttt{remainder}_p()$ in the first step of σ with $\mathit{status}_p = \mathit{good}$ and the last step of σ is the earliest subsequent normal step in the run that causes p to reenter $\texttt{remainder}_p()$ (which would be a return from $\texttt{exit}_p$, or a return of IN_REM from $\texttt{try}_p$ or $\texttt{recover}_p$).

Definition 5 *(RMR)*
- A step of p is an *RMR on a DSM machine* if and only if it is a normal step in which p performs an operation on some variable that is not in $\Delta(p)$.
- A step of p is an *RMR on a Strict or Relaxed CC machine* if and only if it is a normal step in which p performs a non-read operation, or p reads some variable that is not present in p's cache.

Definition 6 *(Active)* A process p is *active* in a configuration C if the condition $(PC_p \neq \texttt{remainder}_p()) \vee (\mathit{status}_p \neq \mathit{good})$ holds in C.

Definition 7 *(Point contention)* The *point contention* at a configuration C is the number of active processes in C.

3 Properties of an abortable RME algorithm

We state the properties required of an abortable RME algorithm which, for easy comprenhensibility, we have divided into four categories: basic safety, responsiveness, liveness, and fairness.

Basic safety properties

P1. Mutual Exclusion: At most one process is in the CS in any configuration of any run.[4]

P2. Critical Section Reentry (CSR) [10]: In any run, if a process p crashes while in the CS, no other process enters the CS until p subsequently reenters the CS.

P3. No Trivial Aborts: In any run, if ABORTSIGNAL[p] is *false* when a process p invokes $\texttt{try}_p()$ and it remains *false* throughout the execution of $\texttt{try}_p()$, then $\texttt{try}_p()$ does not return IN_REM.

Responsiveness properties

Once a process leaves the CS, it should be able to return to the remainder section without having to wait on other processes, as captured by the next property.

P3. Bounded Exit: There is an integer b, which may depend on $|\mathcal{P}|$, such that if in any run any process p invokes and executes $\texttt{exit}_p()$ without crashing, the method completes in at most b steps of p.

The next property formalizes the requirement that, if the environment signals a waiting process p to abort (and maintains that signal so that it is not missed), then the process should be able to quit $\texttt{try}_p()$ (i.e., either return to the remainder or capture the CS) without being obstructed by others.

P4. Bounded Abort [13]: There is an integer b, which may depend on $|\mathcal{P}|$, such that if at any point in any run a process p is in $\texttt{try}_p()$ or is in $\texttt{recover}_p()$ with $status_p = \textit{recover-from-try}$, and from that point on ABORTSIGNAL[p] stays *true* and p executes steps without crashing, then $\texttt{try}_p()$ or $\texttt{recover}_p()$ returns in at most b steps of p.

A process p finds itself in the remainder section either because it crashed while executing the algorithm or because it returned normally from the algorithm. In the former case, when p restarts, it is *required* to execute $\texttt{recover}_p()$, but in the latter case, p has a choice—it can execute either $\texttt{try}_p()$ or $\texttt{recover}_p()$. If p is unsure whether it is restarting from a crashed state, it can harmlessly "probe" by executing $\texttt{recover}_p()$. However, if p executes $\texttt{recover}_p()$ in the latter case, for efficiency we require $\texttt{recover}_p()$ to complete quickly (and return IN_REM).

P5. Fast Probing: There is an absolute constant c, i.e., a constant independent of $|\mathcal{P}|$, such that if in any run any process p executes $\texttt{recover}_p()$ without crashing and with $status_p \in \{\textit{good}, \textit{recover-from-rem}\}$, the method completes in at most c steps of p.

If p crashes while in the CS, the CSR property stated earlier prohibits others from entering the CS until p reenters the CS. Therefore, when p restarts and executes $\texttt{recover}_p()$, we would want p to be able to complete $\texttt{recover}_p()$ (and return IN_CS) without being obstructed by other processes [17]. Similarly, when p executes $\texttt{recover}_p()$ following a crash in the exit section, p should be able to complete $\texttt{recover}_p()$ (returning IN_CS or IN_REM) without having to wait on others. On

[4] We say a process p is in the CS if $PC_p = \texttt{cs}_p()$. Similarly, p is in the remainder, recover, try, or exit sections if PC_p equals or is in $\texttt{remainder}_p()$, $\texttt{recover}_p()$, $\texttt{try}_p()$, or $\texttt{exit}_p()$, respectively.

the other hand, if p crashes while executing $\texttt{try}_p()$, the execution of $\texttt{recover}_p()$ upon restart has two options: either it gives up the attempt to acquire the CS and returns IN_REM or it tries once again to acquire the CS. In the latter case, waiting is unavoidable, but in the former case we require that p completes $\texttt{recover}_p()$ without having to wait on others. The next property formalizes these requirements.

P6. Bounded Recovery: There is an integer b, which may depend on $|\mathcal{P}|$, such that if in any run any process p executes $\texttt{recover}_p()$ without crashing and either with $status_p \in \{$*recover-from-cs*, *recover-from-exit*$\}$ or with $status_p =$ *recover-from-try* and the method returns IN_REM, the method completes in at most b steps of p.

Liveness property

For the traditional mutual exclusion problem, the liveness condition is usually starvation-freedom, which states that if a process p is in $\texttt{try}_p()$ at any point in a fair infinite run, it is in the CS at a later point. We adapt this definition to allow for aborts and crashes. To accommodate aborting, we relax the phrase "it is in the CS at a later point" in the definition to "it returns from $\texttt{try}_p()$ at a later point." Furthermore, since a non-aborting waiting process cannot enter the CS if the process in the CS fails repeatedly (infinitely many times), we could require progress only when there are finitely many crashes:

> Starvation Freedom: In every fair infinite run in which there are only finitely many crash steps, if a process p is in $\texttt{try}_p()$ in a configuration, p subsequently returns from $\texttt{try}_p()$.

Our algorithm satisfies a stronger property that guarantees progress even when there are infinitely many crashes in the run, provided that each process crashes at most a finite number of times in each of its attempts.

P7. Strong Starvation Freedom: In every fair infinite run in which each process crashes at most a finite number of times in each of its attempts, if a process p is in $\texttt{try}_p()$ in a configuration, p subsequently returns from $\texttt{try}_p()$.

Fairness property

For the traditional mutual exclusion problem, a standard fairness property, known as First-Come-First-Served (FCFS), states that if a process p requests the CS before a process q, then q does not enter the CS before p. More precisely, if a process p completes the "doorway"—a bounded section of code at the start of $\texttt{try}_p()$—before q invokes its try method, then q does not enter the CS before p [21].

To extend this definition to the present setting where processes may crash or abort, we require that if p requests the CS before q *and* p *is well behaved* (i.e., p does not crash and p does not abort), then q does not enter the CS before p. More precisely:

P8. FCFS: If a process p completes the doorway in its attempt a before a process q begins its attempt b and p *neither crashes nor receives the abort signal in the attempt* a, then q does not enter the CS in the attempt b before p enters the CS in the attempt a.

4 A key building block: the min-array object [12]

The design of a mutual exclusion algorithm requires a facility by which processes can quickly identify a most deserving (i.e., a highest priority or a longest waiting process) among the waiting processes that should be launched into the CS next. When an algorithm is restricted to using only the read, write, and CAS operations, Jayanti's *min-array construction* [12] has proved useful for this purpose in some earlier algorithms [13, 17]. Our algorithm is also based on the min-array object.

A min-array object X of n locations supports two operations: $X[p].\texttt{write}(v)$, which can only be executed by process $p \in \{1, 2, \ldots, n\}$, writes v in $X[p]$; and $X.\texttt{findmin}()$ returns the minimum value among $X[1], X[2], \ldots, X[n]$. The construction in [12] presents a linearizable and wait-free implementation of this object using only the read, write, and CAS operations. The following properties of this implementation are what makes it useful for our algorithm:

- The implementation has adaptive and small worst-case step complexity. Specifically, a process p completes $X.\texttt{findmin}()$ in $O(1)$ steps and $X[p].\texttt{write}(v)$ in $O(\min(k, \log n))$ steps, where k is the maximum point contention during the execution of $X[p].\texttt{write}(v)$.
- Suppose that p invokes $X[p].\texttt{write}(v)$ and crashes before completing the method; when it restarts, suppose that it invokes $X[p].\texttt{write}(v)$ once more and yet again crashes before completing the method. Suppose this pattern repeats f times before p invokes $X[p].\texttt{write}(v)$ and executes it to completion. Despite the many partial executions before the full execution, the implementation ensures that the $X[p].\texttt{write}(v)$ operation appears to take effect exactly once. Furthermore, the total number of p's steps, over all of the partial executions and the final full execution, is $O(f + \min(k, \log n))$.
- Suppose that p invokes $X[p].\texttt{write}(v)$ and crashes before completing it. When p subsequently restarts, suppose that p chooses to abandon that write operation and instead executes $X[p].\texttt{write}(v')$ to completion, for some $v' \neq v$. Then, the implementation guarantees that either $X[p].\texttt{write}(v)$ does not take effect and only $X[p].\texttt{write}(v')$ takes effect, or $X[p].\texttt{write}(v)$ takes effect before $X[p].\texttt{write}(v')$ takes effect.
- The implementation has $O(n)$ space complexity (i.e., uses only $O(n)$ memory words).

5 The Algorithm and its intuitive description

We present in Fig. 1 our abortable RME algorithm for the set of processes $\mathcal{P} = \{1, 2, \ldots, n\}$. All the shared variables used by our algorithm are stored in NVM. Variables with a subscript of p to their name are local to process p, and are stored in p's registers or volatile memory. We begin by describing the role played by each of the shared variables used in the algorithm.

Persistent variables (stored in NVM)
REGISTRY$[1 \ldots |\mathcal{P}|]$: A min-array; initially REGISTRY$[p] = (p, \infty)$, for all $p \in \mathcal{P}$.
CSSTATUS $\in \{0\} \times (\{0\} \cup \mathbb{N}^+) \cup \{1\} \times \mathcal{P}$; initially $(0, 1)$.
SEQ $\in \mathbb{N}$; initially 1.
$\forall p \in \mathcal{P$, GO$[p] \in \mathbb{N}^+ \cup \{-1, 0\}$, initially $\perp$.
TOKEN $\in \mathbb{N}$, initially 1.

```
1.  Remainder Section

    procedure try_p():
2.  tok_p ← TOKEN
3.  CAS(TOKEN, tok_p, tok_p + 1)
4.  GO[p] ← tok_p
5.  REGISTRY[p].write((p, tok_p))
6.  promote_p(false)
7.  wait till GO[p] = 0 ∨ ABORTSIGNAL[p]
8.  if GO[p] = 0: return IN_CS
9.  return abort_p()

10. Critical Section

    procedure exit_p():
11. REGISTRY[p].write((p, ∞))
12. s_p ← SEQ
13. SEQ ← s_p + 1
14. CSSTATUS ← (0, s_p + 1)
15. promote_p(false)
16. GO[p] ← −1

    procedure recover_p():
17. if GO[p] = −1: return IN_REM
18. return abort_p()

    procedure abort_p():
19. REGISTRY[p].write((p, ∞))
20. promote_p(true)
21. if CSSTATUS = (1, p): return IN_CS
22. GO[p] ← −1
23. return IN_REM

    procedure promote_p(boolean flag_p):
24. (b_p, s_p) ← CSSTATUS
    if b_p = 1:
        peer_p ← s_p
        go to Line 27
25. (peer_p, tok_p) ← REGISTRY.findmin()
    if tok_p = ∞ ∧ flag_p:
        peer_p ← p
    else if tok_p = ∞: return
26. if ¬CAS(CSSTATUS, (0, s_p), (1, peer_p)): return
27. g_p ← GO[peer_p]; if g_p ∈ {−1, 0}: return
28. if CSSTATUS ≠ (1, peer_p): return
29. CAS(GO[peer_p], g_p, 0)
```

Fig. 1 Abortable RME Algorithm for CC and DSM machines. Code for process p

- TOKEN is an unbounded positive integer. A process p reads this variable at the beginning of try$_p$() to obtain its token and then increments, thereby ensuring that processes that invoke the try method later will get a strictly bigger token.
- CSSTATUS and SEQ: These two shared variables are used in conjunction, with SEQ holding an unbounded integer and CSSTATUS holding a pair, which is either (*true*, p) (for some $p \in \mathcal{P}$) or (*false*, SEQ). If CSSTATUS $=$ (*true*, p), it means that p is in the CS and, if CSSTATUS $=$ (*false*, SEQ), it means that no process is in

the CS. If SEQ has a value s while p is the CS, when exiting the CS, p increments SEQ to $s+1$ and writes $(0, s+1)$ in CSSTATUS. As we explain later, this act is crucial to ensuring that no process will be made the owner of the CS after it has moved back to the remainder.

- GO[p] has one of three values — -1, 0, or p's token. The algorithm ensures that GO[p] $= -1$ whenever p is in the remainder "normally", i.e., not because of a crash but because the try, exit, or recover method returned normally. If GO[p] $= 0$, it means that p is made the owner of CS, hence p has the permission to enter the CS. After p obtains a token in $\texttt{try}_p()$, p writes its token in GO[p] and, subsequently when p must wait for its turn to enter the CS, it spins until either GO[p] turns 0 or it receives a signal to abort.
- REGISTRY is a min-array object [12] of n locations. After p obtains a token t in $\texttt{try}_p()$, it announces its interest in capturing the CS by writing the pair (p, t) in REGISTRY[p], and when no longer interested, it removes the token by writing (p, ∞) in REGISTRY[p]. The "less than" relation on pairs is defined as follows: $(p, t) < (p', t')$ if and only if $t < t'$ or $(t = t') \wedge (p < p')$.

Next we present an intuitive understanding of the algorithm, explaining the lines of code and, more importantly, drawing attention to potential race conditions and how the algorithm avoids them.

Understanding $\texttt{try}_p()$

After a process p invokes $\texttt{try}_p()$, it reads TOKEN into tok_p (Line **2**) and then attempts to increment it (Line **3**). The attempt to increment serves two purposes. First, if a different process q invokes $\texttt{try}_q()$ later, it gets a strictly larger token, which helps achieve FCFS. Second, if p were to abort its curent attempt A, it will obtain a strictly larger token in its next attempt A', which, as we will see, helps ensure that any process q that might attempt to release p from its busy-wait in the attempt A will not accidentally release p from its busy-wait in the attempt A'. Process p writes its token in GO[p] (Line **4**), where it will later busy-wait until some process changes GO[p] to 0, and then announces its interest in the CS by changing REGISTRY[p] from (p, ∞) to (p, tok_p) (Line **5**). It then executes $\texttt{promote}_p()$ Line **6**), which ensures that p will launch itself into the CS in the event that the CS is unoccupied and no other process has been waiting longer for the CS than p.

Understanding $\texttt{promote}_p()$

The $\texttt{promote}_p()$ procedure's purpose is to push a waiting process into the CS, if the CS is unoccupied. To this end, p reads CSSTATUS into (b_p, s_p) (Line **24**). If $b_p = 1$, it means that s_p owns the CS. In this case, p sets $peer_p$ to s_p. Recognizing that it is possible that $peer_p$ is still busywaiting (because it is unaware that it owns the CS), p jumps to Line **27**, where it releases $peer_p$ from its busywait. On the other hand, if the CS is unoccupied (i.e., $b_p = 0$), it obtains the minimum entry $(peer_p, tok_p)$ in the registry (Line **25**) to find the process $peer_p$ that has been waiting the longest. Since $\texttt{promote}_p()$ is called from p's Line **6**, when REGISTRY[p] has a finite token number, we have $tok_p \neq \infty$ when p executes Line **25**. So, p proceeds to Line **26**, where it attempts to launch $peer_p$ into the CS by performing a CAS on CSSTATUS. If p's CAS fails, it means that someone else must have succeeded in launching a process

into the CS between p's Line **24** and Line **26**; in this case p has no further role to play, so it returns from the procedure. On the other hand, if p's CAS succeeds, which means that $peer_p$ has been made the CS owner, p has a responsibility to release $peer_p$ from its busywait, i.e., p must write 0 in GO$[peer_p]$. However, there is potential for a race condition here, as explained by the following scenario: some process different from p releases $peer_p$ from its busywait; $peer_p$ enters the CS and then exits to the remainder; some other process q is now in the CS; $peer_p$ executes the try method once more and proceeds up to the point of busy-waiting. Recall that p is poised to write 0 in GO$[peer_p]$. If p executes that write, $peer_p$ will be released from its busywait, so $peer_p$ proceeds to the CS, where q is already present. So, mutual exclusion is violated! Our algorithm averts this disaster by exploiting the fact that, while $peer_p$ busywaits, GO$[peer_p]$'s value is never the same between different attempts of $peer_p$. Specifically, p reads GO$[peer_p]$ into g_p (Line **27**); if g_p is -1 or 0, it means that $peer_p$ is not busywaiting, so p has no role to play, hence it returns. If things have moved on and $peer_p$ no longer is in the CS, then too p has no role to play, hence it returns (Line **28**). Otherwise, there are two possibilities: either GO$[peer_p]$ is still g_p or it has changed. In the former case, $peer_p$ must be busywaiting, so it is imperative that p takes the responsibility to release $peer_p$ (by changing GO$[peer_p]$ to 0). In the latter case, $peer_p$ requires no help from p, so p must not change GO$[peer_p]$ (in order to avoid the race condition described above). This is precisely what the CAS at Line **29** accomplishes.

The rest of `try`$_p()$

Upon returning from `promote`$_p()$, p busywaits until it reads a 0 in GO$[p]$ or it receives a request to abort (Line **7**). If p reads a 0 in GO$[p]$, p infers that it owns the CS, so `try`$_p()$ returns IN_CS (Line **8**). If p receives a request to abort, it calls `abort`$_p()$ (Line **9**), which we describe next.

Understanding `abort`$_p()$

To abort, p writes (p, ∞) in the registry to make it known to all that it has no interest in capturing the CS (Line **19**). If any process will invoke the promote procedure after this point, it will not find p in REGISTRY, so it will not attempt to launch p into the CS. Does this mean that p can now return to the remainder section? The answer is a no because there are two race conditions that need to be overcome.

First, it is possible that, before p performed Line **19**, some process q performed its Line **25** to find p in REGISTRY, and then successfully launched p into the CS (by writing $(1, p)$ in CSSTATUS). Taking care of this scenario is easy: p can read CSSTATUS and if p finds that it owns the CS, it can abort by simply returning IN_CS.

The second potential race is more subtle and harder to overcome. As in the earlier scenario, suppose that, before p performed Line **19**, some process q performed its Line **25** to find p in REGISTRY (i.e., $peer_q = p$). Furthermore, suppose that q is now at Line **26** and CSSTATUS $= (0, s_q)$. So, after performing Line **19**, if p naively returns to the remainder and then q performs Line **26**, we would be in a situation where p has been made the CS owner after it was back in the remainder!

To overcome the above two race conditions, p calls `promote`$_p$(*true*) (Line **20**).

The parameter *true* conveys that the call is made by p while aborting, and has the following impact on how p executes `promote`$_p()$: if p finds the CS to be unoccupied

at Line **24** and finds REGISTRY to be empty at Line **25**, to preempt the second race condition discussed above (where some process q is poised to launch p into the CS), p will attempt to launch itself into the CS (by setting $peer_p$ to p at Line **25** and attempting to change CSSTATUS to $(1, peer_p)$). The key insight is that, after p performs the CAS at Line **26**, only two possibilities remain: either p is already launched into the CS (i.e., CSSTATUS $= (1, p)$) or it is guaranteed that no process will launch p into the CS. In the former case, $\texttt{abort}_p()$ returns IN_CS at Line **21**; and in the latter case, since it is safe for p to return to the remainder, $\texttt{abort}_p()$ returns IN_REM at Line **23** after setting GO$[p]$ to -1 at Line **22** (in order to respect the earlier mentioned invariant that GO$[p] = -1$ whenever p returns to the remainder normally).

Understanding $\texttt{exit}_p()$

There are two routes by which p might enter the CS. One is the "normal" route where p executes $\texttt{try}_p()$ without aborting or crashing, and $\texttt{try}_p()$ returns IN_CS, thereby sending p to the CS. The second route is where p receives an abort signal, calls at Line **9** $\texttt{abort}_p()$, which returns IN_CS at Line **21**, causing $\texttt{try}_p()$ also to return IN_CS at Line **9**. When p is in the CS, p's announcement in REGISTRY$[p]$ (made at Line **5**), would no longer be there if it entered the CS by the second route (because of Line **19**), but it would still be there if it entered the CS by the first route. So, when p exits the CS, it removes its announcement in REGISTRY$[p]$ (Line **11**). It then increments the number in SEQ and gives up its ownership of the CS by changing CSSTATUS from $(1, p)$ to $(0,$ SEQ$)$ (Lines **12**, **13**, **14**). To launch a waiting process, if any, into the just vacated CS, p then executes $\texttt{promote}_p()$ (Line **15**), and returns to the remainder after setting GO$[p]$ to -1 at Line **16** (in order to respect the earlier mentioned invariant that GO$[p] = -1$ whenever p returns to the remainder normally).

Understanding $\texttt{recover}_p()$

Process p executes $\texttt{recover}_p()$ when it restarts after a crash. If GO$[p]$ has the value -1, p infers that either $\texttt{recover}_p()$ was called when $status_p = good$ or the most recent crash had occurred early in $\texttt{try}_p()$, so $\texttt{recover}_p()$ simply sends p back to the remainder (Line **17**). Otherwise, $\texttt{recover}_p()$ simply calls $\texttt{abort}_p()$ (Line **17**), which does the needful. In particular, if p was in the CS at the most recent crash, then CSSTATUS would have $(1, p)$, which causes $\texttt{abort}_p()$ to send p back to the CS. Otherwise, $\texttt{abort}_p()$ extricates p from the algorithm, sending it either to the CS or to the remainder.

6 The invariant

Figure 2 presents the invariant satisfied by the Abortable RME algorithm given in Fig. 1. The invariant is the conjunction of the 13 parts displayed there. Of these, Conditions (1) through (7) of the invariant are used later to prove the safety, responsiveness, and FCFS properties; (12) and (13) are used to prove strong starvation freedom; and (8) through (11) are needed to make the invariant inductive.

The invariant is presented with the following conventions. All statements about process p are universally quantified, i.e., $\forall p \in \mathcal{P}$ is implicit (these are Statements 3 through 11, and Statement 13). The program counter for a process p, i.e., PC_p, can

Conditions:

1. $\textsc{Token} \geq 1$
2. The first field of CSStatus is either 0 or 1. If it is 0, the second field must be Seq. If it is 1, the second field holds a process name. More precisely:
   $(\textsc{CSStatus} = (0, \textsc{Seq})) \vee (\exists q \in \mathcal{P}, \textsc{CSStatus} = (1, q))$
3. Go[p] contains one of three values—-0, -1, or tok_p, the positive integer value that p reads from Token—and the following condition states when Go[p] takes on each of these values.
   $(-1 \leq \textsc{Go}[p] < \textsc{Token}) \wedge (PC_p = \mathbf{5} \Rightarrow \textsc{Go}[p] = tok_p) \wedge (PC_p \in [\mathbf{6}, \mathbf{8}] \Rightarrow \textsc{Go}[p] \in \{0, tok_p\})$
   $\wedge\ (PC_p \in \{\mathbf{9}\text{-}\mathbf{16}, \mathbf{18}\text{-}\mathbf{22}, \mathbf{24}\text{-}\mathbf{29}\} \Rightarrow \textsc{Go}[p] \neq -1)$
   $\wedge\ ((PC_p \in \{\mathbf{2}\text{-}\mathbf{4}, \mathbf{23}\} \vee (PC_p \in \{\mathbf{1}, \mathbf{17}\} \wedge status_p \in \{good, recover\text{-}from\text{-}rem\})) \Rightarrow \textsc{Go}[p] = -1)$
4. The first field of Registry[p] i always p, and its second field is either ∞ or the positive integer token that p reads from Token. The following condition states when Registry[p] takes on these values.
   $(\exists t \in [1, \textsc{Token} - 1] \cup \{\infty\}, \textsc{Registry}[p] = (p, t))$
   $\wedge\ (PC_p \in [\mathbf{6}, \mathbf{8}] \Rightarrow \textsc{Registry}[p] = (p, tok_p))$
   $\wedge\ ((PC_p \in \{\mathbf{5}, \mathbf{12}\text{-}\mathbf{16}, \mathbf{20}\text{-}\mathbf{22}\} \vee \textsc{Go}[p] = -1) \Rightarrow \textsc{Registry}[p] = (p, \infty))$
5. CSStatus contains $(1, p)$ when process p "owns" the CS. The following condition states the regions of the code when p is guaranteed to own the CS and the regions when p is guaranteed not to hold the CS.
   $(((PC_p \in [\mathbf{6}, \mathbf{8}] \wedge \textsc{Go}[p] = 0) \vee PC_p \in [\mathbf{10}, \mathbf{14}] \vee status_p = recover\text{-}from\text{-}cs) \Rightarrow \textsc{CSStatus} = (1, p))$
   $\wedge\ ((PC_p \in \{\mathbf{5}, \mathbf{22}\} \cup [\mathbf{15}, \mathbf{16}] \vee \textsc{Go}[p] = -1) \Rightarrow \textsc{CSStatus} \neq (1, p))$
6. This condition states what values local variables of process p take on.
   $(PC_p = \mathbf{3} \Rightarrow 1 \leq tok_p \leq \textsc{Token}) \wedge (PC_p \in [\mathbf{4}, \mathbf{8}] \Rightarrow 1 \leq tok_p < \textsc{Token})$
   $\wedge\ (PC_p = \mathbf{13} \Rightarrow s_p = \textsc{Seq}) \wedge (PC_p = \mathbf{14} \Rightarrow s_p = \textsc{Seq} - 1)$
   $\wedge\ (PC_p \in [\mathbf{6}\text{::}\mathbf{24}, \mathbf{6}\text{::}\mathbf{29}] \cup [\mathbf{15}\text{::}\mathbf{24}, \mathbf{15}\text{::}\mathbf{29}] \Rightarrow flag_p = false) \wedge (PC_p \in [\mathbf{20}\text{::}\mathbf{24}, \mathbf{20}\text{::}\mathbf{29}] \Rightarrow flag_p = true)$
   $\wedge\ (PC_p \in [\mathbf{26}, \mathbf{29}] \Rightarrow peer_p \in \mathcal{P})$
   $\wedge\ (PC_p \in [\mathbf{2}, \mathbf{16}] \Rightarrow status_p = good)$
7. The following condition states that, if process p goes past the busywait loop at Line 7, it must be because either p has the permission to enter the CS (i.e., Go[p] = 0) or p received the abort signal.
   $(PC_p = \mathbf{8} \Rightarrow (\textsc{Go}[p] = 0 \vee \text{abort was requested})) \wedge (PC_p = \mathbf{9} \Rightarrow \text{abort was requested})$
8. Conditions (12) and (13) below are needed to prove strong starvation-freedom, and Conditions (8) through (11) below are auxiliary conditions that are needed to inductively prove (12) and (13).
   $PC_p \in \{\mathbf{25}, \mathbf{26}\} \Rightarrow (s_p \leq \textsc{Seq} \wedge (\forall q, PC_q \in \{\mathbf{13}, \mathbf{14}\} \Rightarrow s_p \leq s_q))$
9. $((PC_p = \mathbf{25} \wedge \textsc{CSStatus} = (0, s_p)) \Rightarrow$
   $\forall q, (\textsc{Registry}[q] \neq (q, \infty) \Rightarrow (PC_q \in \{\mathbf{6}\text{-}\mathbf{9}, \mathbf{18}, \mathbf{19}\} \vee (PC_q \in \{\mathbf{1}, \mathbf{17}\} \wedge \textsc{Go}[q] \neq -1))))$
   $\wedge\ ((PC_p = \mathbf{26} \wedge \textsc{CSStatus} = (0, s_p)) \Rightarrow (PC_{peer_p} \in [\mathbf{6}, \mathbf{8}] \cup \{\mathbf{18}\text{-}\mathbf{20}, \mathbf{20}\text{::}\mathbf{24}\}$
   $\vee\ (PC_{peer_p} \in \{\mathbf{20}\text{::}\mathbf{25}, \mathbf{20}\text{::}\mathbf{26}\} \wedge s_{peer_p} = s_p)$
   $\vee\ (PC_{peer_p} \in \{\mathbf{1}, \mathbf{17}\} \wedge \textsc{Go}[peer_p] \neq -1)))$
10. $PC_p = \{\mathbf{28}, \mathbf{29}\} \Rightarrow 1 \leq g_p < \textsc{Token}$
11. $PC_p = \mathbf{29} \Rightarrow ((PC_{peer_p} \in \{\mathbf{3}, \mathbf{4}\} \Rightarrow 1 \leq g_p < tok_{peer_p})$
    $\wedge\ (PC_{peer_p} = \mathbf{5} \Rightarrow 1 \leq g_p < \textsc{Go}[peer_p])$
    $\wedge\ ((PC_{peer_p} \in \{\mathbf{6}, \mathbf{7}, \mathbf{8}\} \wedge g_p = \textsc{Go}[peer_p]) \Rightarrow \textsc{CSStatus} = (1, peer_p)))$
12. If a process is registered, some q is either in CS or can be counted on to launch a waiting process into CS.
    $\min(\textsc{Registry}) \neq (*, \infty) \Rightarrow \exists q, (\textsc{CSStatus} = (1, q)$
    $\vee\ (PC_q \in \{\mathbf{1}, \mathbf{17}\} \wedge \textsc{Go}[q] \neq -1) \vee PC_q \in \{\mathbf{6}, \mathbf{15}, \mathbf{18}\text{-}\mathbf{20}, \mathbf{24}\}$
    $\vee\ (PC_q \in \{\mathbf{25}, \mathbf{26}\} \wedge \textsc{CSStatus} = (0, s_q)))$
13. If p has the ownership of CS but Go[p] $\neq 0$, then there is some q that can be counted on to set Go[p] to 0.
    $(\textsc{CSStatus} = (1, p) \wedge \textsc{Go}[p] \neq 0) \Rightarrow \exists q, (PC_q \in \{\mathbf{18}\text{-}\mathbf{20}, \mathbf{24}\} \vee (PC_q = \mathbf{27} \wedge peer_q = p)$
    $\vee\ (PC_q \in \{\mathbf{28}, \mathbf{29}\} \wedge peer_q = p \wedge g_q = \textsc{Go}[p])$
    $\vee\ (PC_q \in \{\mathbf{1}, \mathbf{17}\} \wedge \textsc{Go}[q] \neq -1))$

Fig. 2 Invariant of the Abortable RME Algorithm from Fig. 1

take any of the values from the set $\{\mathbf{1}, \ldots, \mathbf{29}\}$. However, when a call to procedure $\texttt{promote}_p()$ is made by p and p is executing one of the steps from Lines **24**-**29**, for clearly conveying where the call was made from, we prefix the value of PC_p with the line number from where $\texttt{promote}_p()$ was called, along with the scope resolution operator from C++, namely, "::". Thus, $PC_p = \mathbf{6} :: \mathbf{27}$ means p called $\texttt{promote}_p()$ from Line **6** and is now executing Line **27** in that call. Sometimes, in the interest of brevity, we use the range operator, i.e., $[a, b]$, to convey something more than just saying the range of values from a to b (inclusive). That is, if $PC_p \in [\mathbf{6}, \mathbf{8}]$, we also mean that PC_p could take on values from [**6** :: **24**, **6** :: **29**] because there is a call to

$\texttt{promote}_p()$ at Line **6**. Similarly, the range [**5**, **6**] includes Line **5** as well as the lines in the range [**6** :: **24**, **6** :: **29**] because, again, there is a call to $\texttt{promote}_p()$ at Line **6**.

The lemma below asserts that the invariant is correct. Its proof is presented in the archived version of this paper [19].

Lemma 1 *The algorithm in Fig. 1 satisfies the invariant in Fig. 2 (i.e., the conjunction of all the conditions stated in Fig. 2 holds in every configuration of every run).*

7 Proof of the properties and the main theorem

Using the invariant, we now prove that the algorithm satisfies all of the properties stated in Sect. 3, and its RMR complexity, on DSM and Relaxed-CC machines, is adaptive and logarithmic in the worst case.

Lemma 2 (At most one process in CS or Exit) *In any configuration of any run, if p and q are distinct processes and $PC_p \in \{10, 11, 12, 13, 14\}$, then $PC_q \notin \{10, 11, 12, 13, 14\}$.*

Proof Assume to the contrary that there is a configuration C of a run and distinct processes p and q such that $PC_p \in \{10, 11, 12, 13, 14\}$ and $PC_q \in \{10, 11, 12, 13, 14\}$. By Condition 5, in C, CSSTATUS equals both $(1, p)$ and $(1, q)$, which is impossible. □

Lemma 3 (Mutual Exclusion) *At most one process is in the CS (i.e., has the value 10 for its program counter) in any configuration of any run.*

Proof Follows from Lemma 2. □

Lemma 4 (Critical Section Reentry) *In any run, if a process p crashes while in the CS, no other process enters the CS until p subsequently reenters the CS.*

Proof Suppose that p crashes while in the CS, thereby moving from the CS to the remainder section with $status_p$ set to *recover-from-cs*. The value of $status_p$ remains *recover-from-cs* until p subsequently restarts and executes $\texttt{recover}_p()$ to completion. It follows from Condition 5 of the invariant that CSSTATUS $= (1, p)$ and GO$[p] \neq -1$ throughout this interval. Therefore, when p executes $\texttt{recover}_p()$, the condition GO$[p] = -1$ on Line 17 evaluates to *false*, causing p to call and execute $\texttt{abort}_p()$ at Line 18. When executing $\texttt{abort}_p()$, at Line 21, p reads $(1, p)$ from CSSTATUS and returns IN_CS, thereby reentering the CS. Hence, we conclude that CSSTATUS $= (1, p)$ from the time of p's crash to the time of its reentering the CS. Therefore, if the lemma is false and some process q is in the CS (i.e., $PC_q = 10$) before this reentry of p to the CS, we have a contradiction at the configuration where $PC_q = 10$: on the one hand, $PC_q = 10$ implies that CSSTATUS $= (1, q)$ (by Condition 5 of the invariant); and, on the other hand, as we have already argued, CSSTATUS $= (1, p)$. □

Lemma 5 (No Trivial Aborts) *In any run, if* ABORTSIGNAL$[p]$ *is false when a process p invokes $\texttt{try}_p()$ and it remains false throughout the execution of $\texttt{try}_p()$, then $\texttt{try}_p()$ does not return IN_REM.*

Proof Suppose that p returns from an execution e of $\texttt{try}_p()$ and ABORTSIGNAL$[p]$ is false throughout the execution e. Since the loop at Line 7 terminates only when ABORTSIGNAL$[p]$ is true or GO$[p] = 0$, it follows that when p quits the the loop at Line 7 in the execution e, GO$[p]$ is 0. Since no process other than p ever changes GO$[p]$ from 0, the value of GO$[p]$ remains 0 when p evaluates the condition at Line 8, so p returns IN_CS in the execution e. Hence, we have the lemma. □

Lemma 6 (Responsiveness) *The algorithm satisfies Bounded Exit, Bounded Abort, Bounded Recovery, and Fast Probing.*

Proof Each of the methods—$\texttt{exit}_p()$, $\texttt{recover}_p()$, $\texttt{abort}_p()$, and $\texttt{promote}_p()$—executes a constant number of read, write, CAS, or min-array operations. Furthermore, the min-array operations REGISTRY$[p]$.write() and REGISTRY$[p]$.findmin() are wait-free, and complete in $O(\log n)$ and $O(1)$ steps, respectively. Hence, the algorithm satisfies Bounded Exit, Bounded Abort, and Bounded Recovery.

To argue Fast Probing, suppose that p executes $\texttt{recover}_p()$ with $status_p \in \{good, recover\text{-}from\text{-}rem\}$. Then, when p executes Line 17 of $\texttt{recover}_p()$, we have $PC_p = 17$ and $status_p \in \{good, recover\text{-}from\text{-}rem\}$; and it follows from Condition 3 of the invariant that GO$[p] = -1$. Therefore, p returns from $\texttt{recover}_p()$ at Line 17, having executed only $O(1)$ steps. Hence, we have Fast Probing. □

7.1 Proof of FCFS

For any attempt α, let $proc_\alpha$ denote the process that executes the attempt α and, if $proc_\alpha$ executes a line numbered i in the attempt α, let $\alpha[i]$ denote the time when $proc_\alpha$ first executes that line in α.

Lemma 7 *1. The value in* TOKEN *is non-decreasing.*
2. If α and β are any two attempts and $\alpha[3] < \beta[2]$, then the value tok_{proc_α} read from TOKEN *at $\alpha[2]$ is smaller than the value tok_{proc_β} read from* TOKEN *at $\beta[2]$.*
3. The value in SEQ *is non-decreasing.*
4. CSSTATUS $\neq (1, p)$ *at the start and at the completion of any attempt by p.*

Proof We note that the CAS at Line 3 either succeeds and increments the value in TOKEN, or fails and leaves the value unchanged. Moreover, no other line in the algorithm changes TOKEN. Hence, we have Part (1) of the lemma.

Part (2) follows from the observation that at the time $proc_\alpha$ executes the CAS at Line 3 in α, either TOKEN's value is greater than tok_{proc_α}, or the CAS succeeds and TOKEN's value increments to $tok_{proc_\alpha} + 1$.

By Lemma 2, at most one process is in the code segment consisting of Lines 12 and 13. Since SEQ is incremented at these lines and these are the only lines that modify SEQ, Part (3) of the lemma holds.

Just before the step in which p starts an attempt and just after the step when p completes an attempt, we have $PC_p = 1$ and $status_p = good$. Then, the invariant (3) implies that GO$[p] = -1$, which by the invariant (5), implies that CSSTATUS $\neq (1, p)$. Hence, we have Part (4) of the lemma. □

Lemma 8 *If* REGISTRY$[p] \neq (p, \infty)$ *at some time t during an attempt α by a process p, then* CSSTATUS *has* $(1, q)$*, for some q, at some time between t and α's completion.*

Proof Assume to the contrary that REGISTRY$[p] \neq (p, \infty)$ at some time t during an attempt α by process p, and CSSTATUS $= (0, *)$ in the entire interval from t to α's completion. We argue two claims below:

- Claim 1: For some integer s, CSSTATUS $= (0, s)$ in the interval from t to α's completion.
  Proof: By our assumption, the first component of the pair stored in CSSTATUS is 0 in the interval from t to α's completion. Suppose that the claim is false and the value of CSSTATUS changes from $(0, s)$ to a different value $(0, s')$ at some time t' during the interval. Since Line 14 is the only line in the algorithm where any value of the form $(0, *)$ is written in CSSTATUS, it follows that some process q executes Line 14 at t'. Then, by the invariant (5), CSSTATUS $= (1, q)$ at t', which contradicts that CSSTATUS $= (0, *)$ in the entire interval from t to α's completion. □
- Claim 2: In the attempt α, it is not the case that p calls $\texttt{promote}_p(\textit{true})$ after time t and executes it to completion.
  Proof: Suppose that p calls $\texttt{promote}_p(\textit{true})$ after t in the attempt α and executes the method to completion. Claim 1 implies that p finds some value $(0, s)$ in CSSTATUS at Line 24, and CSSTATUS has the same value $(0, s)$ when p executes Line 26. So, p's CAS at Line 26 succeeds and changes the value of CSSTATUS to $(1, peer_p)$, contradicting Claim 1 above. □

At time t, since REGISTRY$[p] \neq (p, \infty)$, it follows from the invariant (4) that $PC_p \notin \{5, 12-16, 20-22\}$ and GO$[p] \neq -1$, and it follows from the invariant (3) that $PC_p \notin \{2-4, 23\}$. Thus, at t, we have $PC_p \in \{1, 6, 7, 8, 9, 10, 11, 17, 18, 19\}$ and GO$[p] \neq -1$.

When p completes α, $PC_p = 1$ and $status_p = good$, so GO$[p] = -1$ (by Condition (3) of the invariant). Thus, GO$[p]$'s value is changed to -1 by p at some point between t and α's completion by executing either Line 16 or Line 22. In either case, we argue below that a contradiction arises.

If p executes Line 16 between t and α's completion, since $PC_p \notin \{12, 13, 14, 15, 16\}$ at time t, it must be the case that p executes Line 12 at some point t' that is between t and α's completion. Then, by Condition (5) of the invariant, CSSTATUS $= (1, p)$ at t', contradicting Claim 1 from above.

If p executes Line 22 between t and α's completion, since $PC_p \notin \{20, 21, 22\}$ at time t, it must be the case that p calls $\texttt{promote}_p(\textit{true})$ (at Line 20) after time t and executes the method to completion, contradicting Claim 2 from above. □

Lemma 9 *If t, t', t'' are points in time such that $t < t' < t''$,* CSSTATUS *has the values $(0, s)$, $(1, q)$, and $(0, s')$ (for some s, q, and s') at times t, t', and t'' respectively, then $s' > s$.*

Proof Condition (2) of the invariant implies that SEQ $= s$ at t. The earliest time after t' that CSSTATUS's value changes is when q executes Line 14 of the exit method. Therefore, t'' is greater than or equal to the time of this execution of Line 14 by q. At

the time of this execution of Line 14 as well as at the time of the prior two lines by q (Lines 12 and 13), Condition (5) of the invariant implies that CSStatus $= (1, q)$. Therefore, the time at which q executes these lines, in particular Line 12, is after t. Therefore, by the monotonicity of Seq, the value s_q that q reads at Line 12 is greater than or equal to s. At Line 14, q writes $(0, s_q + 1)$ in CSStatus. It follows from Condition (2) of the invariant that Seq $= s_q + 1$ at the point when q executes Line 14. Since t'' is greater than or equal to the time of q's execution of Line 14, it follows from the monotonicity of Seq that $s' \geq s_q + 1$. Since $s_q \geq s$, it follows that $s' > s$. □

We introduce some more notation to state and prove the next few lemmas. If α is an attempt in which $proc_\alpha$ enters the CS, let $\alpha[cs]$ denote the earliest time during the interval of α's execution when CSStatus takes on the value $(1, proc_\alpha)$ (Part (4) of Lemma 7 assures that $\alpha[cs]$ is well defined). In the algorithm, Line 26 of the promote method is the only place where CSStatus could be changed to take on the value $(1, proc_\alpha)$. Let π_α denote the execution of the promote method that performs a successful CAS at Line 26 to change the value of CSStatus from $(0, *)$ to $(1, proc_\alpha)$, and let $proc_{\pi_\alpha}$ denote the process that executes π_α. Let $\pi_\alpha[i]$ denote the time at which $proc_{\pi_\alpha}$ executes the line numbered i of the promote method during π_α.

Lemma 10 *If α is an attempt in which $proc_\alpha$ enters the CS, then:*

1. *$proc_\alpha$ does not crash before Line 4 in α.*
2. $\alpha[4] < \pi_\alpha[25]$

Proof Suppose that α is an attempt in which $proc_\alpha$ enters the CS. If $proc_\alpha$ crashes before executing Line 4, then Go$[proc_\alpha] = -1$ at the time of this crash (by Condition (3) of the invariant). So, when $proc_\alpha$ restarts and executes the recover method, the recover method returns at Line 17. Thus, $proc_\alpha$ completes α without entering the CS, a contradiction. Hence, we have the first part of the lemma.

To prove the second part, assume to the contrary that $\pi_\alpha[25] < \alpha[4]$. We consider two cases and derive a contradiction in each case.

– Case 1: $proc_\alpha \neq proc_{\pi_\alpha}$
Since $proc_{\pi_\alpha}$ changes the value of CSStatus to $(1, proc_\alpha)$ at Line 26 of π_α, the response it receives at Line 25 from Registry.`findmin()` must be $(proc_\alpha, \tau)$, for some finite integer τ. Since $\pi_\alpha[25] < \alpha[4]$ (by assumption) and since Registry$[proc_\alpha] = (proc_\alpha, \infty)$ just before the start of α and when $PC_{proc_\alpha} \in \{2, 3, 4\}$ (by Conditions (3) and (4) of the invariant), it follows that $proc_{\pi_\alpha}$ executes Line 25 of π_α when an earlier attempt α' of $proc_\alpha$ was in progress. Then, by Lemma 8, CSStatus must have a value of $(1, q)$ (for some q) at some time t between $\pi_\alpha[25]$ and the start of α. When $proc_{\pi_\alpha}$ reads CSStatus at Line 24 of π_α, the value it obtains must be $(0, s)$, for some s (the value cannot be of the form $(1, q)$ since $proc_{\pi_\alpha}$ proceeds to execute Line 26). When $proc_{\pi_\alpha}$ performs the CAS on CSStatus at Line 26 of π_α, since the CAS is successful, it must be the case that CSStatus's value, just prior to the CAS, is $(0, s')$, for some s'. Furthermore, since the CAS is successful, s' must equal s. However, since $\pi_\alpha[24] < t < \pi_\alpha[26]$, Lemma 9 implies that $s' > s$, a contradiction. □

– Case 2: $proc_\alpha = proc_{\pi_\alpha}$
In this case, π_α is $proc_\alpha$'s execution of `promote` during α. The first part of the lemma guarantees that, during α, $proc_\alpha$ does not crash before Line 4. Therefore, any call to `promote` by $proc_\alpha$ during α does not happen until $proc_\alpha$ executes Line 4, contradicting the assumption that $\pi_\alpha[25] < \alpha[4]$. □

Lemma 11 (FCFS) *For any two attempts α and β, if $proc_\alpha$ completes the doorway in its attempt α before $proc_\beta$ begins its attempt β, and $proc_\alpha$ neither crashes nor receives the abort signal in the attempt α, then $proc_\beta$ does not enter the CS in the attempt β before $proc_\alpha$ enters the CS in the attempt α.*

Proof Assume to the contrary that $proc_\beta$ enters the CS in β before $proc_\alpha$ enters the CS in α. Then, throughout the interval from the start of β to the time t when $proc_\beta$ first enters the CS in β, the premise of the lemma implies that REGISTRY$[proc_\alpha]$ has a finite token and REGISTRY$[proc_\beta]$ has a bigger token, possibly ∞. By the definitions of $\pi_\beta[26]$ and $\beta[cs]$, we have $\pi_\beta[26] = \beta[cs] \leq t$. By Lemma 10, we have $\beta[4] < \pi_\beta[25]$. Putting the above inequalities together, we have $\beta[4] < \pi_\beta[25] < \pi_\beta[26] = \beta[cs] \leq t$. Thus, $\pi_\beta[25]$ falls in the interval from the start of β to the time t. Since REGISTRY$[proc_\alpha]$ has a finite and smaller token than REGISTRY$[proc_\beta]$ during this interval, when $proc_{\pi_\beta}$ executes Line 25 and receives the response of REGISTRY.`findmin`() into $(peer_{proc_{\pi_\beta}}, tok_{proc_{\pi_\beta}})$, we have $peer_{proc_{\pi_\beta}} \neq proc_\beta$ and $tok_{proc_{\pi_\beta}} \neq \infty$. Therefore, Line 26 of π_β cannot possibly change CSSTATUS to $(1, proc_\beta)$, a contradiction. □

7.2 Proof of strong starvation-freedom

Lemma 12 *Consider a fair, infinite run in which each process crashes at most a finite number of times in each of its attempts, and a process p is "stuck" at Line 7, i.e., there exists a time τ such that, for all times $t \geq \tau$, $PC_p = 7$. If* CSSTATUS $= (1, p)$ *at any time $t \geq \tau$, then there exists a later time $t' \geq t$ when* GO$[p] = 0$.

Proof This proof is principally based on Condition (13) of the invariant. If GO$[p] = 0$ at t, the lemma is satisfied; hence, suppose that GO$[p] \neq 0$ at t. We note that GO$[p]$'s value remains unchanged unless some process performs a successful CAS on GO$[p]$ at Line 29. To prove the lemma by contradiction, assume that GO$[p]$ remains unchanged at all times after t.

Given the premise of the lemma, by Condition (13) of the invariant, there is a process q such that Statement S below holds at t:

$$\text{Statement S: } PC_q \in \{\mathbf{18}\text{–}\mathbf{20}, \mathbf{24}\} \vee (PC_q = \mathbf{27} \wedge peer_q = p) \vee (PC_q \in \{\mathbf{28}, \mathbf{29}\} \wedge peer_q = p \wedge g_q = \text{GO}[p]) \vee (PC_q \in \{\mathbf{1}, \mathbf{17}\} \wedge \text{GO}[q] \neq -1)$$

We observe that, whenever Statement S holds, Condition (3) implies that GO$[q] \neq -1$. We use this observation wherever needed in the rest of this lemma, without an explicit reference to it.

Let $t^* \geq t$ be any time at which Statement S holds. This statement asserts that if $PC_q = 1$, then GO$[q] \neq -1$, which implies (by Condition (3) of the invariant) that

$status_q \neq good$. Thus, if q is in the remainder at t^*, it will eventually restart and execute the recover method. In other words, q is guaranteed to take the next step.

Let σ denote the first step of q after t^*. We make the following observations about the possibilities for this step.

(O1) If σ is a crash step of q, the step sets PC_q to 1. Since GO$[q] \neq -1$, it follows that Statement S continues to hold after the step σ.
(O2) If σ is the execution of Line 29, then q performs a successful CAS on GO$[p]$, setting its value to 0, thereby satisfying the lemma. (The CAS is guaranteed to succeed because Statement S guarantees that if $PC_q = 29$, then $peer_q = p$ and $g_q =$ GO$[p]$).)
(O3) If σ is the execution of any of the other lines of code that PC_q points to at time t^*, which can be any of Lines 18-20, 24, 27, 28, 1, or 17, Statement S remains satisfied after the step σ **and** the step moves PC_q closer to Line 29, whose execution (as just observed) causes the lemma to be satisfied.

The three observations above, together with the premise that q eventually stops crashing, imply that GO$[p]$ is eventually set to 0. □

Lemma 13 *Consider a fair, infinite run in which each process crashes at most a finite number of times in each of its attempts, and a process p is "stuck" at Line 7, i.e., there exists a time τ such that, for all times $t \geq \tau$, $PC_p = 7$. If* CSSTATUS $= (1, q)$ *at any time $t \geq \tau$, there exists a later time $t' > t$ when* CSSTATUS $= (0, *)$.

Proof Suppose that in the run, CSSTATUS $= (1, q)$ at time τ. We note that no process other than q can change this value in CSSTATUS, and q can possibly change it by executing Line 14.

We observe that the following Statement S holds at τ.

Statement S:

1. GO$[q] \neq -1$.
   (This follows from Condition (5) of the invariant.)
2. $PC_q \notin \{2-4, 23\} \cup \{5, 22\} \cup [15, 16]$, and if $PC_q \in \{1, 17\}$, then $status_q \notin \{good, recover\text{-}from\text{-}rem\}$. (Therefore, $PC_q \in \{1, 6-14, 17-21\}$.)
   (This follows from the premise that CSSTATUS $= (1, q)$, the first part of Statement S that GO$[q] \neq -1$, and Conditions (5) and (3) of the invariant.)

Let $t^* \geq t$ be any time at which the Statement S above holds. Statement S.2 asserts that if $PC_q = 1$, then $status_q \neq good$. Hence, if q is in the remainder at t^*, it will eventually restart and execute the recover method. In other words, q is guaranteed to take the next step.

Let σ denote the first step of q after t^*. There are many possibilities for what this step σ can be, and we make observations about these.

(O1) If σ is a crash step of q, the step sets PC_q to 1, and since GO$[q] \neq -1$, Condition (3) of the invariant implies that $status_q \neq good$. It follows that Statement S continues to hold after the step σ.
(O2) If σ is the execution of Line 14, then CSSTATUS changes to $(0, *)$, thereby satisfying the lemma.

(O3) If σ is the execution of Line 7, one possibility is that PC_q remains at 7 (because GO$[q] \neq 0$ and the abort signal is not present), and Statement S continues to hold. However, importantly, it follows from Lemma 12 that q eventually quits the busywait at Line 7 and moves on to Line 8.

(O4) If σ is the execution of any of the other lines of code that PC_q points to at time t^*, which can be any of Lines 6, 8-14, 17-21, Statement S remains satisfied after the step σ **and** the step moves PC_q closer to Line 14, whose execution (as just observed) causes the lemma to be satisfied.

The observations above, together with the premise that q stops crashing, imply that CSSTATUS is eventually set to $(0, *)$. □

Lemma 14 *Consider a fair, infinite run in which each process crashes at most a finite number of times in each of its attempts, and a process p is "stuck" at Line 7, i.e., there exists a time τ such that, for all times $t \geq \tau$, $PC_p = 7$. If* CSSTATUS $= (0, *)$ *at any time $t \geq \tau$, then there exists a later time $t' > t$ when* CSSTATUS *changes from* $(0, *)$ *to* $(1, q)$*, for some q.*

Proof The proof is principally based on Condition (12) of the invariant. Since p is stuck at Line 7 from time τ, REGISTRY$[p]$ holds (p, a), for some positive integer a, at all times after τ; so min(REGISTRY) $\neq (*, \infty)$ at all times after τ.

Suppose that in the run R, CSSTATUS $= (0, s)$ at some time $t \geq \tau$. We note that CSSTATUS can change from this value if and only if some process q later performs a successful CAS at Line 26. To prove the lemma by contradiction, assume that CSSTATUS remains unchanged at $(0, s)$ at all times after t.

At time t, since min(REGISTRY) $\neq (*, \infty)$ and CSSTATUS $= (0, s)$, Condition (12) of the invariant guarantees that there is a process q for which the following statement holds:

$$\underline{\text{Statement S}}\text{: } (PC_q \in \{\mathbf{1}, \mathbf{17}\} \wedge \text{GO}[q] \neq -1) \vee PC_q \in \{\mathbf{6}, \mathbf{15}, \mathbf{18}\text{–}\mathbf{20}, \mathbf{24}\} \vee (PC_q \in \{\mathbf{25}, \mathbf{26}\} \wedge \text{CSSTATUS} = (0, s_q)))$$

We observe that, whenever Statement S holds, Condition (3) implies that GO$[q] \neq -1$.

Let $t^* \geq t$ be any time at which Statement S holds. If $PC_q = 1$, since GO$[q] \neq -1$, Condition (3) of the invariant implies that $status_q \neq good$. Hence, if q is in the remainder at t^*, it will eventually restart and execute the recover method. In other words, q is guaranteed to take the next step.

Let σ denote the first step of q after t^*. There are many possibilities for what this step σ can be, and we make observations about these.

(O1) If σ is a crash step of q, the step sets PC_q to 1, and since GO$[q] \neq -1$, Statement S continues to hold after the step σ.

(O2) If σ is the execution of Line 26, since CSSTATUS $= (0, s_q)$, the CAS at Line 26 succeeds and changes CSSTATUS to $(1, peer_q)$, thereby satisfying the lemma.

(O3) If σ is the execution of Line 24, the step sets PC_q to 25, and ensures that CSSTATUS $= (0, s_q)$ is true after the step. So, Statement S continues to hold after the step σ.

(O4) If σ is the execution of any of the other lines of code that PC_q points to at time t^*, which can be any of Lines 1, 17, 6, 15, 18-20, or 24-25, Statement S remains satisfied after the step σ **and** the step moves PC_q closer to Line 26, whose execution (as just observed) causes the lemma to be satisfied.

The observations above, together with the premise that q eventually stops crashing, imply that CSSTATUS is eventually set to $(1, *)$, which contradicts our assumption that CSSTATUS $= (0, s)$ at all times after t. □

Lemma 15 *In every fair, infinite run in which each process crashes at most a finite number of times in each of its attempts, for any q, if* CSSTATUS *changes from $(0, *)$ to $(1, q)$ at any time t, then q enters the CS at some time after t.*

Proof Let σ denote the step performed at time t that changes CSSTATUS from $(0, *)$ to $(1, q)$, and let r denote the process that performs this step (r is possibly the same as q). Regardless of whether r is the same as or distinct from q, since CSSTATUS $\neq (1, q)$ just before the step σ, it follows from Condition (5) of the invariant that $PC_q \notin [10 - 14]$ immediately before and immediately after the step σ. Furthermore, Lemma 13 guarantees that CSSTATUS is changed to $(0, *)$ at some time $t' > t$, and the earliest such change is possible in the code only when q executes Line 14. Since $PC_q \notin [10 - 14]$ at time t, and $PC_q = 14$ at time $t' > t$, it must be the case that q enters the CS (i.e., q enters Line 10) sometime after t (and before t'). □

Lemma 16 (Strong Starvation Freedom) *In every fair, infinite run in which each process crashes at most a finite number of times in each of its attempts, no process is stuck at Line 7, i.e., for all processes p, if $PC_p = 7$ at any time, then $PC_p \neq 7$ at a later time.*

Proof Assume to the contrary that a process p is "stuck" at Line 7, i.e., there exists a time τ such that, for all times $t \geq \tau$, $PC_p = 7$. Then, by applying Lemmas 13, 14, and 15 repeatedly ad infinitum, we see that there are infinitely many attempts in which processes enter the CS. Since the number n of processes is finite, it follows that some process enters the CS infinitely many of its attempts (while p is stuck at Line 7, past its doorway), thereby violating the FCFS property, which contradicts Lemma 11. □

7.3 The main theorem

The theorem below summarizes the result of our paper.

Theorem 1 *The algorithm in Fig. 1 is an abortable RME algorithm for n processes using read, write, and CAS operations. It satisfies the following properties: Mutual Exclusion, Critical Section Reentry, No Trivial Aborts, Bounded Exit, Bounded Abort, Fast Probing, Bounded Recovery, FCFS, and Strong Starvation Freedom.*

A process incurs $O(\min(k, \log n))$ RMRs per passage on DSM and Relaxed-CC machines, and $O(n)$ RMRs per passage on Strict-CC machines, where k is the maximum point contention during the passage.

If a process p crashes f times during its attempt and k is the maximum point contention during the attempt, on DSM and Relaxed-CC machines, p incurs $O(f +$

$\min(k, \log n))$ *RMRs in that attempt, and on Strict-CC machines,* p *incurs* $O(f + n)$ *RMRs in that attempt.*

The algorithm's space complexity is $O(n)$.

Proof The properties listed are proved in Lemmas 3, 4, 5, 6, 11, and 16.

We now analyze the complexity. Consider the RMRs that a process p incurs due to its busy-wait at Line 7. On DSM machines, where GO[p] is assigned to p's part of NVM, p does not incur any RMRs at Line 7. On Relaxed-CC machines, one RMR is incurred when bringing GO[p] to p's cache at the start of executing Line 7, spinning on GO[p] incurs no RMRs, one RMR is incurred when some process changes GO[p] to 0, and possibly one more RMR is incurred to read that 0 in GO[p]. Thus, on Relaxed-CC machines, p incurs only $O(1)$ RMRs at Line 7. On Strict-CC machines, while p spins on GO[p] at Line 7, $O(n)$ processes could be at Line 29, each poised to perform a CAS on GO[p]. At most one of these succeeds in its CAS, but every one of them makes p incur an RMR with their CAS (albeit the CAS is unsuccessful). Thus, p can incur $O(n)$ RMRs at Line 7.

As explained in Sect. 4, each of Lines 5 and 19 incurs $O(\min(k, \log n))$ RMRs. Every other line in the code (except Line 7 that is already analyzed) incurs at most one RMR. Hence, we have the RMR complexity stated in the lemma.

The space complexity of $O(n)$ immediate from the observation that GO[p] array takes $O(n)$ space and REGISTRY takes $O(n)$ space, as explained in Sect. 4, other shared variables take $O(1)$ space, and $O(1)$ local variables per process. □

8 Discusion and conclusion

In this paper, we have introduced the notion of a mutual exclusion lock that is both recoverable and abortable. Our algorithm demonstrates a curious relation between recoverability and abortability: an algorithm designed only to be recoverable can easily incorporate abortability if only the recover method were carefully designed to be bounded even when recovering from a crash that occurs in the try method. This idea works because aborting can then be implemented by feigning a crash and executing the recover method. In fact, our algorithm showcased that this idea leads to an optimal RMR algorithm for DSM and Relaxed-CC machines, using only the commonly available read, write, and CAS operations.

It would be interesting to explore if the logarithmic RMR complexity, shown here for DSM and Relaxed-CC machines, is also attainable for Strict-CC machines.

Acknowledgements This paper's presentation and proof benefitted greatly from a plethora of critical comments by the reviewers of an earlier version of this journal submission. We also thank Siddhartha Jayanti for his critical reading and comments, and the NETYS '19 reviewers for their helpful feedback.

References

1. Alon A, Morrison A (2018) Deterministic abortable mutual exclusion with sublogarithmic adaptive rmr complexity. In: Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, New York, NY, USA, PODC '18, ACM, pp. 27–36
2. Attiya H, Hendler D, Woelfel P (2008) Tight RMR Lower Bounds for Mutual Exclusion and Other Problems. In: Proc. of the Fortieth ACM Symposium on Theory of Computing, New York, NY, USA, STOC '08, ACM, pp. 217–226
3. Chan DYC, Woelfel P (2020) Recoverable mutual exclusion with constant amortized rmr complexity from standard primitives. In: Proceedings of the 39th Symposium on Principles of Distributed Computing, New York, NY, USA, PODC '20, Association for Computing Machinery, p. 181–190
4. Craig TS (February 1993) Building FIFO and Priority-Queuing Spin Locks from Atomic Swap. Tech. Rep. TR-93-02-02, Department of Computer Science, University of Washington
5. Dhoked S, Mittal N (2020) An adaptive approach to recoverable mutual exclusion. In: Proceedings of the 39th Symposium on Principles of Distributed Computing, New York, NY, USA, PODC '20, Association for Computing Machinery, p. 1–10
6. Dijkstra EW (1965) Solution of a Problem in Concurrent Programming Control. Commun ACM 8(9):569–569
7. Giakkoupis G, Woelfel P (2017) Randomized abortable mutual exclusion with constant amortized rmr complexity on the cc model. In: Proceedings of the ACM Symposium on Principles of Distributed Computing, New York, NY, USA, PODC '17, ACM, pp. 221–229
8. Golab W, Hendler D (2017) Recoverable mutual exclusion in sub-logarithmic time. In: Proceedings of the ACM Symposium on Principles of Distributed Computing, New York, NY, USA, PODC '17, ACM, pp. 211–220
9. Golab W, Hendler D (2018) Recoverable Mutual Exclusion Under System-Wide Failures. In: Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, New York, NY, USA, PODC '18, ACM, pp. 17–26
10. Golab W, Ramaraju A (2016) Recoverable Mutual Exclusion: [Extended Abstract]. In: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, New York, NY, USA, PODC '16, ACM, pp. 65–74
11. Intel. Intel® Optane™ DC Persistent Memory Product Brief. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf, 2019 (accessed November 26, 2020)
12. Jayanti P (2002) f-arrays: Implementation and Applications. In: Proceedings of the Twenty-first Symposium on Principles of Distributed Computing, New York, NY, USA, PODC '02, ACM, pp. 270–279
13. Jayanti P (2003) Adaptive and efficient abortable mutual exclusion. In: Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing, New York, NY, USA, PODC '03, ACM, pp. 295–304
14. Jayanti P, Jayanti S, Joshi A (2018) Optimal Recoverable Mutual Exclusion using only FASAS. In: The 6th Edition of The International Conference on Networked Systems, NETYS 2018
15. Jayanti P, Jayanti S, Joshi A (2019) A recoverable mutex algorithm with sub-logarithmic rmr on both cc and dsm. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, New York, NY, USA, PODC '19, Association for Computing Machinery, p. 177–186
16. Jayanti P, Jayanti SV (2019) Constant Amortized RMR Complexity Deterministic Abortable Mutual Exclusion Algorithm for CC and DSM Models. In: Accepted for publication in PODC' 19
17. Jayanti P, Joshi A (2017) Recoverable FCFS mutual exclusion with wait-free recovery. In: 31st International Symposium on Distributed Computing, DISC 2017, pp. 30:1–30:15
18. Jayanti P, Joshi A (2019) Recoverable mutual exclusion with abortability. In: Atig MF, Schwarzmann AA (eds) Networked Systems. Springer International Publishing, Cham, pp 217–232
19. Jayanti P, Joshi A (2020) Recoverable mutual exclusion with abortability

20. Katzan D, Morrison A (2020) Recoverable, Abortable, and Adaptive Mutual Exclusion with Sublogarithmic RMR Complexity. In: Proceedings of The International Conference on Principles of Distributed Systems (OPODIS 2020), OPODIS 2020
21. Lamport L (1974) A New Solution of Dijkstra's Concurrent Programming Problem. Commun ACM 17(8):453–455
22. Lee H (2010) Fast local-spin abortable mutual exclusion with bounded space. In: Proceedings of the 14th International Conference on Principles of Distributed Systems, Berlin, Heidelberg, OPODIS'10, Springer-Verlag, pp. 364–379
23. Mellor-Crummey JM, Scott ML (1991) Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. ACM Trans Comput Syst 9(1):21–65
24. Pareek A, Woelfel P (2012) Rmr-efficient randomized abortable mutual exclusion. In: Aguilera MK (ed) Distributed Computing. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 267–281
25. Ramaraju A (2015) RGLock: Recoverable mutual exclusion for non-volatile main memory systems. Master's thesis, University of Waterloo
26. Raoux S, Burr GW, Breitwisch MJ, Rettner CT, Chen Y-C, Shelby RM, Salinga M, Krebs D, Chen S-H, Lung H-L et al (2008) Phase-change random access memory: A scalable technology. IBM J Res Dev 52(4/5):465
27. Scott ML (2002) Non-blocking Timeout in Scalable Queue-based Spin Locks. In: Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing, New York, NY, USA, PODC '02, ACM, pp. 31–40
28. Scott ML, Scherer WN (2001) Scalable queue-based spin locks with timeout. In: Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, New York, NY, USA, PPoPP '01, ACM, pp. 44–52
29. Scott ML, Scherer WN (2001) Scalable Queue-based Spin Locks with Timeout. In: Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, New York, NY, USA, PPoPP '01, ACM, pp. 44–52
30. Strukov DB, Snider GS, Stewart DR, Williams RS (2008) The missing memristor found. nature 453(7191):80
31. Tehrani S, Slaughter JM, Deherrera M, Engel BN, Rizzo ND, Salter J, Durlam M, Dave RW, Janesky J, Butcher B et al (2003) Magnetoresistive random access memory using magnetic tunnel junctions. Proc IEEE 91(5):703–714