An Almost Tight RMR Lower Bound for Abortable Test-And-Set

Aryaz Eghbali
Department of Computer Science, University of Calgary, Canada
aryaz.eghbali@ucalgary.ca

Philipp Woelfel
Department of Computer Science, University of Calgary, Canada
woelfel@ucalgary.ca

Abstract

We prove a lower bound of $\Omega(\log n/\log\log n)$ for the remote memory reference (RMR) complexity of abortable test-and-set (leader election) in the cache-coherent (CC) and the distributed shared memory (DSM) model. This separates the complexities of abortable and non-abortable test-and-set, as the latter has constant RMR complexity [27].

Golab, Hendler, Hadzilacos and Woelfel [29] showed that compare-and-swap can be implemented from registers and test-and-set objects with constant RMR complexity. We observe that a small modification to that implementation is abortable, provided that the used test-and-set objects are atomic (or abortable). As a consequence, using existing efficient randomized wait-free implementations of test-and-set [23], we obtain randomized abortable compare-and-swap objects with almost constant ($O(\log^* n)$) RMR complexity.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms

Keywords and phrases Abortability, Test-And-Set, Leader Election, Compare-and-Swap, RMR Complexity, Lower Bound

Digital Object Identifier 10.4230/LIPIcs.DISC.2018.21

Related Version A full version of the paper is available at [21], `https://arxiv.org/abs/1805.04840`.

Funding This research was undertaken, in part, thanks to funding from the Canada Research Chairs program and from the Discovery Grants program of the Natural Sciences and Engineering Research Council of Canada (NSERC).

Acknowledgements We thank the anonymous reviewers for their careful reading and detailed feedback.

1 Introduction

In this paper, we study the remote memory references (RMR) complexity of abortable test-and-set. Test-and-set (TAS) is a fundamental shared memory primitive that has been widely used as a building block for classical problems such as mutual exclusion and renaming, and for the construction of stronger synchronization primitives [37, 41, 20, 15, 8, 7, 6, 29].

We consider a standard asynchronous shared memory system in which n processes with unique IDs communicate by reading and writing shared registers. A TAS object stores a bit that is initially 0, and provides two methods, `TAS()`, which sets the bit and returns its previous value, and `read()`, which returns the current value of the bit. TAS is closely related to mutual exclusion [18]: a TAS object can be viewed as a one-time mutual exclusion algorithm, where only one process (the one whose `TAS()` returned 0) can enter the critical section [19].

TAS objects have consensus-number two, and therefore they have no wait-free implementations from atomic registers. In particular, in deterministic TAS implementations, processes may have to wait indefinitely, by spinning (repeatedly reading) variables. It is common to predict the performance of such blocking algorithms by bounding remote memory references (RMRs). These are memory accesses that traverse the processor-to-memory interconnect. Local-spin algorithms achieve low RMR complexity by spinning on locally accessible variables. Two models are common: In *distributed shared memory (DSM)* systems, each shared variable is permanently locally accessible to a single processor and remote to all other processors. In *cache-coherent (CC)* systems, each processor keeps local copies of shared variables in its cache; the consistency of copies in different caches is maintained by a *coherence protocol*. Memory accesses that cannot be resolved locally and have to traverse the processor-to-memory interconnect are called *remote memory references* (RMRs).

Golab, Hendler, and Woelfel [27] implemented deadlock-free TAS objects from registers with $O(1)$ RMR complexity for the DSM and the CC model, which in turn have been used to construct equally efficient comparison-primitives, such as compare-and-swap (CAS) objects [29]. These constructions are particularly useful in the study of the complexity of the mutual exclusion problem, for which the RMR complexity is the standard performance metric [10, 9, 36, 13, 16, 33, 34, 35, 14, 31, 32, 42, 24, 11, 38, 17, 39, 25].

In the context of mutual exclusion, it has been observed that systems often require locks to support a “timeout” capability that allows a process waiting too long for the lock, to abort its attempt [43]. In database systems, such as Oracle’s Parallel Server and IBM’s DB2, the ability of a thread to abort lock attempts serves the dual purpose of recovering from a transaction deadlock and tolerating preemption of the thread that holds the lock [43]. In real time systems, the abort capability can be used to avoid overshooting a deadline. Solutions to this problem have been proposed in the form of *abortable* mutual exclusion algorithms [43, 33, 42, 39, 17, 26]. In such an algorithm, at any point in the entry section a process may receive an *abort signal* upon which, within a finite number of its own steps, it must either enter the critical section or abort its current attempt to do so, by returning to the remainder section.

The complexity of the mutual exclusion problem in systems providing only registers is not affected by abortability: The abortable algorithms by Danek and Lee [17] and Lee [40] use only atomic registers and achieve $O(\log n)$ RMR complexity, which asymptotically matches the known lower bound for non-abortable mutual exclusion [13]. But abortable mutual exclusion algorithms seem to be much more difficult to obtain than non-abortable ones, and it is not surprising that all such algorithms preceding [17, 40] used stronger synchronization primitives (e.g., LL/SC objects in [33]). Moreover, no RMR efficient randomized abortable mutual exclusion algorithms are known, unless stronger primitives are used [42, 26]; on the other hand, several non-abortable randomized implementations use only registers [30, 31, 25, 14].

As mentioned earlier, CAS objects with $O(1)$ RMR complexity can be obtained from registers [29], but they cannot be used in an abortable mutual exclusion algorithm without sacrificing its abortability: if a process receives the abort signal while being blocked in an operation on a CAS object, it has no option to finish that operation in a wait-free manner, and thus may not be able to abort its attempt to enter the critical section. In general, implemented blocking strong objects, cannot be used to obtain abortable mutual exclusion objects.

One way of dealing with this impasse can be to make implementations of strong primitives also abortable, and to devise mutual exclusion algorithms in such a way that they accommodate operation aborts. Similarly, other algorithms and data structures that may require timeout capabilities, can potentially be implemented from abortable objects, but not from non-abortable ones.

We define abortability in the following, natural way: In a concurrent execution, a process executing an operation on the object may receive an abort signal at any point in time. When that happens, the process must finish its method call within a finite number of its own steps, and as a result the method call may fail to take effect, or it may succeed. The resulting execution must satisfy the safety conditions of the object (e.g., linearizability), if all failed operations are removed. Moreover, a process must be able to find out, by looking at the return value, whether its aborted operation succeeded, and if it did, then the return value must be consistent with a successful operation.

It may be tempting to define a weaker forms of abortability, e.g., where a return value of an aborted operation does not indicate whether the operation succeeded or not. As discussed in Section 1.1, such a weaker notion of abortability has indeed been suggested [4], but in a different context, where a process can choose for itself to abort its own pending operation (e.g., if it detects contention). In our scenario, where aborts are determined in an adversarial manner, the usefulness of such a weaker notion is not clear. For example, abortable TAS objects (according to our definition) can easily be used to implement an abortable mutual exclusion algorithm (TAS-lock): One can store a pointer to a "current" TAS object in a single register R. To get the lock, a process calls `TAS()` on the TAS object that R points to, and if the return value is 0, then the process has the lock, and otherwise it keeps reading R until its value changes. To release the lock, the process simply swings the pointer R so that it points to a new, fresh TAS object (this technique was proposed in [5], and [1, 2] showed how to bound the number of involved TAS objects). This also works in the case of aborts, because a process knows whether its operation took effect, and thus whether it is allowed to swing the pointer (and in fact must, to avoid dead-locks).

For the weaker definition of abortability mentioned above, a process whose `TAS()` aborted may not be able to find out whether it has the lock or not, and then it can also not swing the pointer to a new TAS object, even though its `TAS()` may have set the bit from 0 to 1. In fact, suppose that two processes call `TAS()`, and both `TAS()` calls abort without receiving the information whether the aborted operation took effect. Then the TAS bit may be set, but none of the processes has received any information regarding who was successful, and reading the TAS object also provides no information.

Even though our notion of abortability may seem strong, any abortable mutual exclusion algorithm can be used to obtain any abortable object from its corresponding sequential implementation, by simply protecting the sequential code in the critical section. An interesting question is therefore, whether abortable objects can be obtained at a lower RMR cost than mutual exclusion.

We observe that this is true for implementations of abortable CAS objects from abortable TAS objects on the CC model: a straightforward modification of the constant RMR implementation of non-abortable CAS from TAS objects and registers [29], immediately yields an abortable CAS object, provided that the used TAS objects are atomic or also abortable.

▶ **Observation 1.** *There is a deadlock-free implementation of abortable CAS from atomic registers and deadlock-free abortable TAS objects, which has $T + O(1)$ RMR complexity on the CC model, provided that* `TAS()` *operations have RMR complexity T.*

This theorem immediately implies that we could use atomic TAS objects (which are trivially abortable) to obtain abortable CAS objects with constant RMR complexity. But obviously, it does not help constructing deterministic abortable CAS objects from registers. However, we can use the fact that there are known randomized constructions of TAS objects, which are not only RMR efficient, but even efficient with respect to step complexity. More precisely, Giakkoupis and Woelfel [23] presented a randomized TAS implementation from registers, where the maximum number of steps any process takes in a `TAS()` operation has expectation $O(\log^* n)$ against an oblivious adversary. (This means, the order in which processes take steps must be completely independent of their random decisions.) This construction is also randomized wait-free, meaning that, for any schedule all `TAS()` calls terminate with probability 1. Therefore, `TAS()` calls are abortable (in a randomized sense), as for any schedule each method call terminates with probability 1 (whether the process receives an abort signal or not). In the construction of CAS above, we can therefore use such randomized TAS implementation in place of abortable TAS.

► **Corollary 2.** *There is a deadlock-free randomized implementation of abortable CAS from atomic registers, such that on the CC model against an oblivious adversary each abort is randomized wait-free, and each operation on the object incurs at most $O(\log^* n)$ RMRs.*

Recall that there is also a deterministic constant RMR implementation of TAS from registers [27]. Hence, making this implementation abortable and applying Observation 1 would immediately yield deterministic constant RMR abortable implementations of CAS from registers. Unfortunately, it turns out that a deterministic constant RMR implementation of abortable TAS from registers does not exist. In particular, we define the abortable leader election (LE) problem, which is not harder than abortable TAS (with respect to RMR complexity). Our main technical result is an RMR lower bound of $\Omega(\log n/\log\log n)$ for that object.

In a (non-abortable) LE protocol, every process decides for itself whether it becomes the leader (it returns *win*) or whether it loses (it returns *lose*). At most one process can become the leader, and not all participating processes can lose. I.e., if all participating processes finish the protocol, then exactly one of them returns *win* and all others return *lose*. Note that then in an abortable LE protocol all participating processes are allowed to return *lose*, provided that all of them received the abort signal.

An abortable TAS object immediately yields an abortable LE protocol: Each process executes a single `TAS()` operation and returns *win* if the `TAS()` call returns 0, and otherwise *lose* (i.e., it returns *lose* also when the `TAS()` return value indicates a failed abort). Similarly, it is easy to implement abortable TAS from abortable LE and a single register, preserving the asymptotic RMR complexity (but care must be taken to obtain linearizability).

Our main result is the following:

► **Theorem 3.** *For both, the DSM and the CC model, any deadlock-free abortable leader election (and thus any abortable TAS) implemented from registers has an execution in which at least one process incurs $\Omega(\log n/\log\log n)$ RMRs.*

This lower bound is asymptotically tight up to a $\log\log n$ factor, because one can trivially obtain a TAS object with $O(\log n)$ RMR complexity by protecting a straight forward sequential `TAS()` implementation with the abortable mutual exclusions algorithms by Danek and Lee [17] and Lee [40].

Leader election is one of the seemingly simplest synchronization primitives that have no wait-free implementation. In particular, as argued above, the lower bound in Theorem 3 immediately also applies to abortable TAS. This is in stark contrast to the $O(1)$ RMRs upper bound for non-abortable TAS and even CAS implementations [27, 29]. It shows that adding abortability to synchronization primitives is almost as difficult as solving abortable mutual exclusion, which has an RMR complexity of $\Theta(\log n)$ [17, 40].

In our lower bound proof we identify the crucial reason for why abortable LE is harder than its non-abortable variant: According to standard bi-valency arguments, for any deadlock-free LE algorithm, there is an execution in which some process takes an infinite number of steps. But it is not hard to see that one can design an (asymmetric) 2-process LE protocol in which one fixed process is wait-free, because the other one waits for the first one to make a decision if it detects contention. It turns out that this is not the case for abortable LE: Here, for any process, there is an execution in which that process takes an infinite number of steps.

1.1 Other Related Work.

Attiya, Guerraoui, Hendler, and Kuznetsov [12] consider augmenting non-wait-free implementations with a mechanism that allows processes to abort an ongoing operation and returning a special "failed" return value. Contrary to our model, where aborts are chosen in an adversarial manner, in the work of Attiay et. al. processes can decide when to abort in order to achieve termination (e.g., when they detect contention). This makes implemented objects weaker, while our abortable objects are stronger than non-abortable ones. A similar notion of abortable objects was suggested by Aguilera, Frølund, Hadzilacos, Horn, and Toueg [4]. In their work, processes can also decide to abort an ongoing operation, but the caller of an aborted operation may not find out whether its operation took effect or not. Since this uncertainty may not be acceptable, they also introduce query-abortable objects, where a query operation allows a process to determine additional information about its last non-query operation.

Note that their notion of abortability is quite different from the one used commonly for mutual exclusion and adopted by us, where the system, and not the implementation, dictates when a process needs to abort.

2 Abortable Compare-And-Swap in the CC Model

In this section we consider the cache-coherent (CC) model. Each process obtains a cached copy with each read of a register, and the cached copy only gets invalidated if some process later writes to the same register. Writes as well as reads of non-cached registers incur RMRs, while reads of cached registers do not.

A CAS object provides two operations, `CAS(`cmp, new`)`, and `read()`. Operation `read()` returns the current value of the object. Operation `CAS(`cmp, new`)` writes *new* to the object, if the current value is *cmp*, and otherwise does not change the value of the object. In either case it returns the old value of the object.

Golab et. al. [28] gave an implementation of CAS from TAS objects and registers, which has constant RMR complexity in the CC model, i.e., each `CAS()` and each `read()` operation incurs only $O(1)$ RMRs. In this section we show how to make that implementation abortable, provided that the used TAS objects are either atomic or abortable and deadlock-free. The pseudocode is in Figure 1. The original (non-abortable) version of the code is shown in black and our additional code to make it abortable in red (lines 6 and 20). it is assumed that the abort-signal is sent to a process by means of setting the process' flag *abort*.

Method NameDecide()

```
1  $x := T$.TAS()
2  if $x = 0$ then
3      $leader := \text{PID}$
4  else
5      while $leader = \perp$ do
6          if $abort$ then return $\perp$
7  return $leader$
```

Method CAS(cmp, new)

```
8  $d := D$
9  $old := d \rightarrow value$
10 if $old = cmp \wedge cmp \neq new$ then
11     $winner := d \rightarrow N$.NameDecide()
12     if $winner = PID$ then
13         $d' :=$ getNewPage()
14         $d' \rightarrow value := new$
15         $D := d'$
16         $d \rightarrow value := new$
17         $d \rightarrow flag :=$ True
18     else
19         while $d \rightarrow flag \neq$ True do
20             if $abort$ then return $\perp$
21         $old := d \rightarrow value$
22 return $old$
```

Figure 1 Implementation of (abortable) `NameDecide()` and `CAS()`. Without lines 6 and 20 the algorithms are equivalent to the non-abortable implementations in [28].

2.1 From TAS to Name Consensus

The implementation in [28] first constructs a name consensus object from a single TAS object T. This objects supports a method `NameDecide()`, which each process is allowed to call at most once. All `NameDecide()` calls return the same value (agreement), which is the ID of a process calling `NameDecide()` (validity).

The non-abortable implementation in [28] uses a TAS object T and a register *leader* that is initially $\perp$. In a `NameDecide()` call, a process p first calls T.`TAS()`. If the `TAS()` returns 0, then p *wins*, and writes p to *leader*. Otherwise, p *loses*, and so it repeatedly reads *leader*, until $leader \neq \perp$, upon which p can return the value of *leader*. It is easy to see (and was formally proved in [28]) that this is a correct name consensus algorithm.

We now show how this implementation can be made abortable, assuming the TAS object T is atomic or abortable. We assume that when a process receives the abort signal, a static process-local variable *abort*, which is initially false, changes to `True`.

Recall that abortability requires that the return value of a `TAS()` operation indicates whether it failed or succeeded. We assume a failed `TAS()` simply returns $\perp$. In `NameDecide()`, processes are only waiting until *leader* changes. If a process is receiving the abort signal while waiting for *leader* to change, then it can also simply return $\perp$. The rest of the algorithm is the same as the original name consensus algorithm.

Clearly, the new code (line 6) does not affect RMR complexity, and following an abort the code is wait-free. Moreover, correctness (validity and agreement) in case of no failed `NameDecide()` operations follow immediately from correctness of the original algorithm. If a `NameDecide()` operation fails (i.e., returns $\perp$), then it did not change any shared memory object (its `TAS()` must have either failed, or returned 1). Hence, removing an aborted and failed `NameDecide()` operation from the execution does not affect any other processes, and therefore the resulting execution must be correct.

2.2 From Name Consensus to Compare-And-Swap

We now show how the abortable name consensus algorithm can be used to obtain abortable CAS. Consider the implementation of `CAS`(cmp, new) on the right hand side in Figure 1. The black code is logically identical to the one in [28]. It uses a register D that points to a *page*, which stores two registers, *value* and *flag*, as well as a name consensus object N. Register *value* at the page pointed to by D stores the current value of the object. (Thus, a `read()` operation, for which we omit the pseudo code, simply returns $D \to value$.) The `CAS()` operation assumes a wait-free method `getNewPage()`, which returns an unused page from a pool of pages (for simplicity assume this pool has infinitely many pages, but there are methods for wait-free memory management that allow using a bounded pool [29, 3]).

For a description of how the algorithm `CAS(`cmp, new`)` works, we refer to [28]. We can prove that the abortable version presented here is correct, provided that the non-abortable version (with line 20 removed) is: First of all, obviously line 20 does not change the RMR complexity. Moreover, if a process receives the abort-signal, then its abortable `NameDecide()` call terminates within a finite number of steps, and the process also does not wait in the while-loop, so its `CAS()` call completes within a finite number of its steps. Finally, notice that a `CAS()` call returns $\perp$ only if an abort signal was received, and in that case no shared memory objects are affected (the process cannot have won the `NameDecide()` call). Hence, all aborted and failed operations can be removed from the execution without changing anything for the remaining operations. As a result we obtain Observation 1.

3 RMR Lower Bound for Abortable Leader Election

In this section, we give an overview of the RMR lower bound proof for abortable leader election (and thus TAS) as stated in Theorem 3. Due to space constraints, the full proof is omitted, but it is made available in the technical report [21].

First, we define some notation, the system model, RMR complexity, and the abortable leader election problem.

3.1 Lower Bound Preliminaries

System Model and Notation. For an set Q and any non-negative integer k, let Q^k denote the set of all sequences of length k that contain only the elements in Q. Furthermore, Q^* is the set of all sequences over Q.

For the lower bound we assume a set $\mathcal{P}$ of n processes, and an arbitrary large but finite set $\mathcal{R}$ of shared registers. In each shared memory step (corresponding to a state transition), a process either reads or writes a register in $\mathcal{R}$. At an arbitrary point, a process may also receive an *abort signal* which does not result in a shared memory access, but in a state change of that process, provided the process has not received the abort signal earlier. Once a process has reached a halting state, it remains in that state forever, and does not execute any further shared memory steps.

For each process $p \in \mathcal{P}$, we define a special *abort symbol* $p^\top$, which is used to indicate that a process receives an abort signal (as defined below). For a set $P \subseteq \mathcal{P}$ let $P^\top = \{p^\top \mid p \in P\}$, and $P^\Delta = P \cup P^\top$. A *schedule* is a sequence σ over $\mathcal{P}^\Delta$. Thus, any schedule σ is in $(\mathcal{P}^\Delta)^*$. The length of a schedule σ is denoted by $|\sigma|$. Let $Proc(\sigma)$ denote the set of processes $p \in \mathcal{P}$ that occur in σ at least once, not counting symbols in $\mathcal{P}^\top$.

A *configuration* is a sequence that describes the state of each process in $\mathcal{P}$ and each register in $\mathcal{R}$. A configuration C and a schedule $\sigma \in P^\Delta$ of length one result in a new configuration $Conf(C, \sigma)$, obtained from C by process p taking its next step, if $\sigma = p \in \mathcal{P}$, or by process p receiving the abort signal, if $\sigma = p^\top \in \mathcal{P}^\top$. If $\sigma = \sigma_1\sigma_2 \dots \sigma_k$ is a schedule of length $k > 1$, then the new configuration is determined inductively as $Conf\big(Conf(C, \sigma_1); \sigma_2 \dots \sigma_k\big)$. Configuration C and schedule $\sigma = \sigma_1 \dots \sigma_k$ also define an *execution* $Exec(C, \sigma)$, which is a sequence $s_1 s_2 \dots s_k$, where s_i is the step executed or the abort signal received in the transition from $C_{i-1} = Conf(C, \sigma_1 \dots \sigma_{i-1})$ to $C_i = Conf(C_{i-1}, \sigma_i)$. To specify that an execution starting in C and running by schedule σ is running algorithm A, we use $Exec_A(C, \sigma)$. The length of an execution E is denoted by $|E|$. We call s_i an *abort step by process p*, if in s_i process p receives the abort signal.

The initial configuration is denoted by Γ. A configuration C is *reachable*, if there exists a schedule σ such that $Conf(\Gamma, \sigma) = C$. Since only reachable configurations are important in our algorithms and proofs, we use *configuration* instead of *reachable configuration* from this point on. For a configuration C we let $\sigma_{\to C}$ denote an arbitrary but unique schedule such that $Conf(\Gamma, \sigma_{\to C}) = C$, and we define $E_{\to C} = Exec(\Gamma, \sigma_{\to C})$.

The projection of a schedule σ to a set $Q \subseteq \mathcal{P}^{\Delta}$ is denoted by $\sigma|Q$. For an execution E and a set P of processes, $E|P$ denotes the sub-sequence of E that contains all (abort and shared memory) steps by processes in P. If Q or P contains only one symbol, s, then we write $\sigma|s$ instead of $\sigma|\{s\}$, or $E|s$ instead of $E|\{s\}$.

Recall that a configuration C determines the state of each process. I.e., for any two executions E and E' resulting in the same configuration C, each process is in the same state at the end of E as at the end of E', and in particular $E|p = E'|p$. Therefore, we associate the state of a process in configuration C with $E_{\to C}|p$. (But note that if two executions E and E' are indistinguishable to each process in $Q \subseteq \mathcal{P}$, then this does not in general imply that $E|Q = E'|Q$.) The value of register r in configuration C is denoted by $val_C(r)$. Configurations C and D are *indistinguishable* to some process p, if $E_{\to C}|p = E_{\to D}|p$ and $val_C(r) = val_D(r)$ for every register $r \in \mathcal{R}$. For a set $Q \subseteq \mathcal{P}$, we write $C \sim_Q D$ to denote that configurations C and D are indistinguishable to each process in Q; for a set consisting of a single process p we write $C \sim_p D$ instead of $C \sim_{\{p\}} D$.

Finally, for two sequences s_1 and s_2 let $s_1 \sigma s_2$ denotes their concatenation. (We use this for schedules and executions.)

RMR Complexity. Our lower bound applies to both, the standard asynchronous distributed shared memory (DSM) model and cache-coherent (CC) model. In fact, we use a model that combines both, caches as well as locally accessible registers for each process.

We assume that the set of registers, $\mathcal{R}$, is partitioned into disjoint memory segments $\mathcal{R}_p$, for $p \in \mathcal{P}$. The registers in $\mathcal{R}_p$ are *local* to process p and *remote* to each process $q \neq p$. We say that at the end of execution E a process p has a valid cached copy of register r, if in E process p reads or writes r at some point, and no other processes writes r after that. Note that the configuration obtained at the end of an execution starting in Γ uniquely determines whether p has a valid cached copy of a register r. The reason is that the state of p in configuration C determines the value that was written to or read from r when p accessed r last, and p has a valid cached copy of r if and only if $val_C(r)$ equals that value. Let $Cache_p(C)$ denote the union of $\mathcal{R}_p$ and the set of registers of which process p has a valid cached copy in configuration C if p has not terminated in C, and the empty set if p is terminated in C.

A step in an execution E is either *local* or *remote* (we say it *incurs an RMR* if it is remote). All abort steps are local. A non-abort step by process p is local, if and only if it is either a read or a write of a register in $\mathcal{R}_p$, or it is a read of a register of which p has a local cached copy. (Our definition corresponds to a write-through cache; in a write-back cache, certain writes may also be local. Even though we believe that our lower bound proof technique can accommodate write-back caches, this is left for future work.)

For an execution E and a process p, $RMR_p(E)$ is the number of RMR steps by process p in execution E. For $Q \subseteq \mathcal{P}$ we define $RMR_Q(E) = \sum_{q \in Q} RMR_q(E)$, which is equal to the total number of RMRs incurred by processes in Q in E. For the sake of conciseness, we use $RMR(E)$ instead of $RMR_{\mathcal{P}}(E)$.

Abortable Leader Election. An algorithm solves *abortable leader election*, if for any schedule σ, in $Exec(\Gamma, \sigma)$ each process that terminates returns *win* or *lose*, at most one process returns *win*, and if all processes in $Proc(\sigma)$ return *lose*, then all processes in $Proc(\sigma)$ receive the abort signal.

We usually assume without explicitly saying so that an abortable leader election satisfies *deadlock-freedom* and *wait-free abort*, defined as follows: Wait-free abort means that after a process received the abort signal it terminates within a finite number of its own steps. An infinite schedule σ is P*-fair* for $P \subseteq \mathcal{P}$, if each process in P appears infinitely many times in σ. An infinite execution E is P*-fair*, if there exists a configuration C and a P-fair schedule σ such that $E = Exec(C, \sigma)$. We use *fair* schedule and *fair* execution, instead of P-fair, when $P = \mathcal{P}$. A method is *deadlock-free* if for any schedule σ all process' method calls terminate in $Exec(\Gamma, \sigma)$, provided this execution is P-fair, where P is the set of processes calling the method.

3.2 Properties of Abortable Leader Election

In this section we derive the critical property that distinguishes non-abortable from abortable leader election for the purpose of the lower bound. We consider algorithms in which each process returns either *win* or *lose* upon termination. We call such algorithms *binary*. Note that any (abortable) leader election algorithm is a binary algorithm. (Recall that in abortable leader election aborted and failed operations return *lose*, and not $\perp$ as in TAS.)

Several results in this section will concern only two arbitrarily selected processes in the n-process system for $n \geq 2$. For ease of notation, we will call these processes a and b.

For an execution E of a binary algorithm in which a returns x and b returns y, let (x, y) denote the *outcome vector* of E. For a binary algorithm A and a configuration C, let $\mathcal{V}_A(C)$ denote the set of all outcome vectors of $\{a, b\}$-only executions starting in C, in which processes a and b terminate.

First we observe that the outcome vectors of two indistinguishable configurations are equal.

▶ **Observation 4.** *For any binary algorithm A, if configurations C and D are indistinguishable to processes a and b, then $\mathcal{V}_A(C) = \mathcal{V}_A(D)$.*

Proof. Since C and D are indistinguishable to processes a and b, $E_{\to C}|a = E_{\to D}|a$, $E_{\to C}|b = E_{\to D}|b$, and for any register r, $val_C(r) = val_D(r)$. Thus, for any x in $\{a, b\}^{\Delta}$, we have $\big(E_{\to C} \circ Exec(C, x)\big)|a = \big(E_{\to D} \circ Exec(D, x)\big)|a$, $\big(E_{\to C} \circ Exec(C, x)\big)|b = \big(E_{\to D} \circ Exec(D, x)\big)|b$, and for any register r, $val_{Conf(C,x)}(r) = val_{Conf(D,x)}(r)$. So by induction, for any finite $\{a, b\}$-only schedule σ, $Conf(C, \sigma) \sim_{\{a,b\}} Conf(D, \sigma)$. Therefore, if in $Exec(C, \sigma)$ process $p \in \{a, b\}$ terminates, it also terminates in $Exec(D, \sigma)$ and it returns the same value in both executions. Hence, the outcome vector $\mathcal{V}_A(C)$ is equal to $\mathcal{V}_A(D)$. ◀

For a binary algorithm A, configuration C is *bivalent* if $\big\{(win, lose), (lose, win)\big\} = \mathcal{V}_A(C)$. This definition of bivalency refers to two fixed but arbitrarily chosen processes, a and b. In a system with more than two processes, we may write $\{a, b\}$-bivalent to indicate the two processes a and b to which this definition applies. A configuration is *strongly bivalent* (or strongly $\{a, b\}$-bivalent) if it is bivalent and a solo-run by any process $p \in \{a, b\}$, starting in C, results in p winning. (Solo-run by p means an execution in which only process p takes steps, and none of them is an abort-step.)

A similar argument to the FLP Theorem [22] implies that for any deadlock-free binary algorithm and for any reachable bivalent configuration, there exists an infinite execution, where no process terminates.

▶ **Lemma 5.** *Let A be a deadlock-free binary algorithm and C an $\{a,b\}$-bivalent configuration. There exists an infinite schedule $\sigma \in \{a,b\}^*$, such that in $Exec_A(C,\sigma)$ none of a and b terminate.*

To prove this lemma we first prove Claim 6 and use the fact that none of a and b can be terminated in an $\{a,b\}$-bivalent configuration.

▶ **Claim 6.** *In any deadlock-free binary algorithm A, if configuration C is $\{a,b\}$-bivalent, then either one of $Conf(C,a)$ and $Conf(C,b)$ is $\{a,b\}$-bivalent, or there exists an infinite $\{a,b\}$-only execution, where none of a and b terminates.*

Proof. Since configuration C is $\{a,b\}$-bivalent, $\mathcal{V}_A(C) = \big\{(win, lose), (lose, win)\big\}$. Suppose neither $Conf(C,a)$ nor $Conf(C,b)$ is $\{a,b\}$-bivalent. Then there exist distinct $x, y \in \{win, lose\}$ such that

$$\begin{aligned}&\mathcal{V}_A\big(Conf(C,a)\big) = \{(x,y)\} \text{ , and}\\&\mathcal{V}_A\big(Conf(C,b)\big) = \{(y,x)\}\end{aligned} \tag{1}$$

We now distinguish two cases.

Case 1. In C, processes a and b are poised to access different registers or poised to read the same register. Thus,

$$Conf(C, a \circ b) = Conf(C, b \circ a). \tag{2}$$

By (1), $(y,x) \notin \mathcal{V}_A\big(Conf(C,a)\big)$. Since $\mathcal{V}_A\big(Conf(C,a\circ b)\big) \subseteq \mathcal{V}_A\big(Conf(C,a)\big)$, it holds $(y,x) \notin \mathcal{V}_A\big(Conf(C,a\circ b)\big)$. Thus, by (2), $(y,x) \notin \mathcal{V}_A\big(Conf(C,b\circ a)\big)$. Since $\mathcal{V}_A\big(Conf(C,b\circ a)\big) \subseteq \mathcal{V}_A\big(Conf(C,b)\big) = \{(y,x)\}$, this means that $\mathcal{V}_A\big(Conf(C,b\circ a)\big) = \emptyset$. But this contradicts deadlock-freedom, as in a fair schedule starting in $Conf(C, b\circ a)$ both processes must terminate and output something.

Case 2. In configuration C, both processes are poised to access the same register r, and at least one of them is poised to write r. Without loss of generality, assume that a is poised to write register r. If a takes its write step after b's step, then a's state and shared register values are no different than if only a takes its write step and b does not take its step. So $Conf(C,a) \sim_a Conf(C, b\circ a)$. If process a does not terminate in a solo-run starting in $Conf(C,a)$, then the claim is true, because there exists an infinite execution starting in C that neither a nor b terminates. However, if process a terminates in a solo-run starting in $Conf(C,a)$, by (1), we can conclude that $(x,y) \in \mathcal{V}_A\big(Conf(C,b\circ a)\big)$. Since $\mathcal{V}_A\big(Conf(C,b\circ a)\big) \subseteq \mathcal{V}_A\big(Conf(C,b)\big)$, it holds that $(x,y) \in \mathcal{V}_A\big(Conf(C,b)\big)$. This contradicts $\mathcal{V}_A\big(Conf(C,b)\big) = \{(y,x)\}$. ◀

Any deadlock-free (non-abortable) 2-process leader election algorithm has a bivalent initial configuration. But in any fair schedule, both processes terminate. Therefore, the infinite execution that is guaranteed by the above corollary cannot be fair; in particular, it requires one of the two processes to run solo at some point. However, one can construct a deadlock-free (non-abortable) leader election algorithm in which one process never takes an infinite number of steps, no matter what the schedule is. The lemma below shows that this is not true for abortable two-process leader election algorithm.

▶ **Lemma 7.** *Let A be a deadlock-free abortable 2-process leader election algorithm with wait-free aborts. For any process p, there exists an execution starting in the initial configuration, in which p takes a infinitely many steps.*

Proof. Let Γ be the initial configuration of A. For the purpose of contradiction, assume there is a fixed process, a, that terminates within a finite number of its own steps in all executions. Let b be the other process.

By the semantics of abortable leader election, there is no execution in which both processes win, i.e.,

$$(win, win) \notin \mathcal{V}_A(\Gamma). \tag{3}$$

Let algorithm A' be the same as A except that during any execution,

(1) if any of the two processes receive the abort signal, the abort signal is ignored; and

(2) if in step s process b reads (a, x), where $x \neq \perp$, then b continues its program, as if it had received the abort signal immediately after step s.[1]

In any execution of A, processes a and b can only both lose, if they both receive the abort signal. Since in A' both processes ignore the abort signals (and only b possibly simulates having received an abort signal), there is no execution of A' in which a and b both lose. Thus, for the initial configuration Γ' of A',

$$(lose, lose) \notin \mathcal{V}_{A'}(\Gamma'). \tag{4}$$

Consider any execution $E' = Exec(\Gamma', \sigma')$ of algorithm A' starting in Γ'. We now create an execution $E = Exec(\Gamma, \sigma)$ of A starting in Γ, by scheduling the processes in exactly the same order as in E', but removing all abort signals. Moreover, when for the first time b reads a value of (a, x) in E, where $x \neq \perp$ (if that happens), then we send process b the abort signal. By construction of A', processes a and b execute exactly the same shared memory steps in execution E of algorithm A as in execution E' of algorithm A'. Thus, for every schedule σ' there is a schedule σ such that processes a and b execute in $Exec_{A'}(\Gamma', \sigma')$ the same shared memory steps as in $Exec_A(\Gamma, \sigma)$. This implies

$$\mathcal{V}_{A'}(\Gamma') \subseteq \mathcal{V}_A(\Gamma). \tag{5}$$

Note that in the construction above, if σ' is fair, then so is σ. Hence, the fact that A is deadlock-free implies

$$A' \text{ is deadlock-free.} \tag{6}$$

In algorithm A, in a sufficiently long solo-run by a starting in Γ, in which a does not receive the abort-signal, process a terminates (by deadlock-freedom) and returns win (by the semantics of abortable leader election). Hence, in A' process a also terminates and returns win after a sufficiently long solo-run starting from Γ, because it takes exactly the same steps as in A. Since A' is deadlock-free by (6), process b terminates after a sufficiently long solo-run following a's solo-run, and by (3) process b returns $lose$. With a symmetric argument, for algorithm A', in a sufficiently long solo-run by b starting in Γ, followed by a sufficiently long solo-run of a, process b returns win and process a returns $lose$. Hence, $\{(win, lose), (lose, win)\} \subseteq \mathcal{V}_{A'}(\Gamma')$. Using (3) and (4) we conclude

$$\mathcal{V}_{A'}(\Gamma') = \big\{(win, lose), (lose, win)\big\}. \tag{7}$$

We will now show that A' is wait-free. This together with (7) contradicts Lemma 5, and thus proves the lemma.

[1] Recall that we assumed that each value that a process p writes is of the form (p, y), where $y \neq \perp$.

Recall that in every execution of algorithm A process a terminates within a finite number of its own steps. As a result, the same is true for A'.

Hence, it suffices to show that b terminates within a finite number of its own steps. Suppose there is an execution E^* of A' in which b executes an infinite number of steps. Then b never reads a value of (a, x), where $x \neq \bot$, as otherwise it would simulate having received the abort-signal in A, and then terminate after a finite number of steps. Since b never reads a value of (a, x), where $x \neq \bot$, it cannot distinguish E^* from a solo-run starting in Γ'. Hence, b does not terminate in such an infinite solo-run. This contradicts (6). ◀

One of the core properties of the abortable leader election problem that allows us to prove the lower bound is that there are no reachable strongly bivalent configurations in any execution.

▶ **Lemma 8.** *Let A be an abortable n-process leader election algorithm with wait-free aborts for $n \geq 2$. Further, let C be a reachable configuration and a, b two distinct processes that do not receive the abort-signal in $E_{\to C}$, and which both terminate in any $\{a, b\}$-fair execution starting in C. Then C is not strongly $\{a, b\}$-bivalent.*

Proof. Suppose C is strongly $\{a, b\}$-bivalent. Then it is $\{a, b\}$-bivalent, so

$$\mathcal{V}_A(C) = \{(lose, win), (win, lose)\}, \tag{8}$$

and if a or b runs solo starting in C, then that process wins. Because $\sigma \in \mathcal{P}^*$, neither a nor b receives the abort-signal in $Exec(\Gamma, \sigma)$. By the assumption that aborts are wait-free, processes a and b both terminate in sufficiently long solo runs starting in $Conf(C, a^\top)$ and $Conf(C, b^\top)$, respectively. Let x and y be the return values of a in $Exec(C, a^\top \circ a^{k_a})$ and of b in $Exec(C, b^\top \circ b^{k_b})$, respectively, for sufficiently large integers k_a and k_b.

Since $Conf(C, a^\top) \sim_a Conf(C, a^\top b^\top)$,

$$a \text{ returns } x \text{ in } Exec(C, a^\top b^\top \circ a^{k_a}). \tag{9}$$

Similarly, since $Conf(C, b^\top) \sim_b Conf(C, a^\top b^\top)$,

$$b \text{ returns } y \text{ in } Exec(C, a^\top b^\top \circ b^{k_b}). \tag{10}$$

We distinguish the following cases.

Case 1: $x = y = win$. In a sufficiently long solo-run by b following $Exec(C, a^\top b^\top \circ a^{k_a})$, process b must terminate (by deadlock-freedom). Since a wins in that execution, b must lose. Thus,

$$(win, lose) \in \mathcal{V}_A\big(Conf(C, a^\top b^\top)\big). \tag{11}$$

Applying a symmetric argument to a sufficiently long solo-run by a following $Exec(C, b^\top a^\top \circ b^{k_b})$, we obtain

$$(lose, win) \in \mathcal{V}_A\big(Conf(C, a^\top b^\top)\big). \tag{12}$$

Hence, using (8), we get $\{(win, lose), (lose, win)\} = \mathcal{V}_A\big(Conf(C, a^\top b^\top)\big)$. Then by Lemma 5, there exists an infinite execution starting in $Conf(C, a^\top b^\top)$, such that a and b do not terminate. This contradicts wait-free aborts.

Case 2: $x = y = lose$. In a sufficiently long solo-run by b following $Exec(C, a^\top b^\top \circ a^{k_a})$, process b must terminate (by deadlock-freedom). Since a loses in that execution, by (8), process b must win. Thus, $(lose, win) \in \mathcal{V}_A\big(Conf(C, a^\top b^\top)\big)$, and with a symmetric argument $(win, lose) \in \mathcal{V}_A\big(Conf(C, a^\top b^\top)\big)$. We get a contradiction for the same reasons as in Case 1.

Case 3: $\{x, y\} = \{win, lose\}$. Without loss of generality, assume $x = win$. Then in $Exec(C, a^{\top} a^{k_a})$ process a wins. On the other hand, since C is strongly bivalent, b wins in a sufficiently long solo-run starting in C. Since $C \sim_b Conf(C, a^{\top})$, process b also wins in a long enough solo-run starting in $Conf(C, a^{\top})$. Hence, we have shown that any of the two processes in $\{a, b\}$ wins in a solo-run starting in $Conf(C, a^{\top})$. By deadlock-freedom and (8) the other process loses, if it performs a long enough solo-run afterwards. This shows that $Conf(C, a^{\top})$ is strongly bivalent.

Now let A' be the 2-process algorithm in which a and b act exactly as in algorithm A, but the initial configuration is $\Gamma' = Conf(C, a^{\top})$. Then A' is a deadlock-free abortable 2-process leader election algorithm with wait-free aborts: The wait-free abort property is inherited from A. Deadlock-freedom follows from the assumption that a and b terminate in any fair execution starting in C. Correctness follows from (8) and the fact that each process wins in a long enough solo-run starting in the initial configuration $Conf(C, a^{\top})$ (because that configuration is strongly bivalent).

Moreover, in A' process a always terminates within a finite number of its own steps. This follows from the wait-free abort property of A and the fact that both processes simulate A starting in configuration $Conf(C, a^{\top})$, in which a has already received the abort-signal. This contradicts Lemma 7. ◀

3.3 Constructing an Expensive Execution

We now consider an abortable leader election algorithm. We will construct a schedule such that in an execution starting in the initial configuration at least one process takes $\Omega(\log n / \log \log n)$ RMR steps, where n is the number of processes.

3.3.1 Additional Assumptions

We make the following assumptions that do not restrict the generality of our results. Recall that processes are state machines, each using some infinite state space $\mathcal{Q}$. We assume that during an execution a process never enters the same state twice. Further, we assume that each register stores a pair in $\mathcal{P} \times (\mathcal{Q} \cup \{\bot\})$, where $\bot \notin \mathcal{Q}$. The initial value of each register in $\mathcal{R}_p$ is $(p, \bot)$, and when a process p writes to any register, it writes a pair (p, x), where x is p's state before its write operation, and in particular $x \neq \bot$. I.e., we are using a full information model, where processes write all information they have observed in the past. As a result, no two writes in an execution write the same value. Each process's first shared memory step is a read outside of its local shared memory segment, that we call *invocation read*, and thus incurs an RMR. Adding such a step to the beginning of each process's program does not affect the asymptotic RMR complexity of the algorithm. We will assume that at the end of its execution, each process p reads all registers in $\mathcal{R}_p$ once. Since those reads do not incur any RMRs, this assumption can be made without loss of generality. We call p's last read of register $r \in \mathcal{R}_p$ the *terminating read* of r, and we assume that after p's last terminating read, p will immediately enter a halting state.

3.3.2 Terminology and Notation

We define some additional terms and notation.

We say process p is *visible* on register r in configuration C if $val_C(r) = (p, x)$, for some $x \in \mathcal{Q}$. Let $L(C)$ be the set of processes that have lost in configuration C.

When we construct our high RMR execution, we need to make sure that whenever a process gains information about some other process that has not yet lost, someone pays for that with an RMR. To keep track of who knows who, we define a set $K(C)$ that contains pairs (p,q) of processes. Informally, (p,q) is in $K(C)$ if p has already gained information about process q in the execution leading to configuration C, or p can gain such information for "free" (i.e., without an RMR being paid for that). Gaining information does not only mean that p reads a register that q has written; it means anything that might affect p's execution, e.g., p's cache copies being invalidated. $K(C)$ is the union of three sets $K_1(C)$, $K_2(C)$, and $K_3(C)$, defined as follows:

- $K_1(C)$ is the set of all pairs (p,q), $p \neq q$, such that in $E_{\to C}$ process p reads a register while process q is visible on that register. I.e., p reads a value of (q,x), where $x \in \mathcal{Q}$. *Informally:* p has learned about q in $E_{\to C}$.
- $K_2(C)$ is the set of all pairs (p,q), $p \neq q$, such that in $E_{\to C}$ process q takes at least one shared memory step and process p reads a register in $\mathcal{R}_q$. *Informally:* Process p may have a valid cached copy of a register $r \in \mathcal{R}_q$, and by writing to r process q can invalidate that cached copy without incurring an RMR.
- $K_3(C)$ is the set of all pairs (p,q), $p \neq q$, such that in $E_{\to C}$ process p takes at least one shared memory step, and q writes to a register $r \in \mathcal{R}_p$ before p's terminating read of r. *Informally:* p may learn about q without incurring an RMR by scanning all its registers in $\mathcal{R}_p$.

Let $K(C) = K_1(C) \cup K_2(C) \cup K_3(C)$. We say process p *knows* process q in configuration C if $(p,q) \in K(C)$.

In our inductive construction of an RMR expensive execution, we will sometimes erase processes from the constructed execution. For that reason, if p knows about q, i.e., $(p,q) \in K(C)$, then we will not remove a process q from the execution $E_{\to C}$. We achieve this by ensuring that whenever $(p,q) \in K(C)$, $q \in L(C)$, and as discussed earlier no lost processes will be erased.

However, we have to be careful about cases in which p does not know directly about q. For example, suppose process q writes to register r in execution E, and later some process z overwrites r and finally p becomes poised to read r. In our inductive construction we may want to remove either z or p from the execution, because we do not want z to be discovered by p. However, removing z reveals q on register r, and so now p may discover q. To account for that we introduce the concept of *hidden* processes.

In particular, for a configuration C and a register r we define a set $H_r(C)$ of processes *hidden* on r as follows:

(H1) For $r \notin \mathcal{R}_p$, $p \in H_r(C)$ if and only if either p does not access r in $E_{\to C}$, or p accesses r in $E_{\to C}$ at some point t, and either no process writes r after t, or at least one process that writes r after t is in $L(C)$;
Idea: If p's write to r was overwritten by some processes, then at least one of them has lost and thus will not be erased from the execution. Hence, erasing a process does not reveal p's write to any other process.

(H2) For $r \in \mathcal{R}_p$, $p \in H_r(C)$ if and only if any process other than p that writes to r in $E_{\to C}$ is in $L(C)$.
Idea: If a process q wrote to a register r in p's local memory segment, then q has lost. Therefore, q will not be erased from the execution. This is important because p can read r for free and we have to assume that it does so frequently, so erasing q from the execution might change what p observes in the execution.

Let $H(C) = \bigcap_{r \in \mathcal{R}} H_r(C)$. We say process p is *hidden* in configuration C, if $p \in H(C)$.
We finally define the concept of a safe configuration as follows. Configuration C is *safe*, if

(S1) for any pair $(p, q) \in K(C)$, $q \in L(C)$, and
(S2) if $p \notin H(C)$, then either $p \in L(C)$, or p takes no shared memory step in $E_{\to C}$.
The first property ensures that no process p knows another process q that has not yet lost, and the second property says that all processes that are not hidden must have lost, or not even started participation. As a result, in an execution leading to a safe configuration, we can erase all processes that do not lose, without affecting any other processes. Formally, we will prove for a schedule σ, a safe configuration $C = \mathit{Conf}(\Gamma, \sigma)$ and a set of processes $P \supseteq L(C)$,

- $\mathit{Exec}(\Gamma, \sigma)|P = \mathit{Exec}(\Gamma, \sigma|P^{\Delta})$;
- $\mathit{RMR}_P(\mathit{Exec}(\Gamma, \sigma)) = \mathit{RMR}_P(\mathit{Exec}(\Gamma, \sigma|P^{\Delta}))$; and
- $\mathit{Cache}_p(C) = \mathit{Cache}_p(\mathit{Conf}(\Gamma, \sigma|P^{\Delta}))$ for all $p \in P$.

Moreover, if C is safe, then $\mathit{Conf}(\Gamma, \sigma|P^{\Delta})$ is also safe.

3.3.3 Overview of the Construction

Let $n \geq 4$, $\ell = \lfloor \log n / c \log\log n \rfloor$ for some sufficiently large constant c. We inductively construct a schedule σ_i and a set of processes $P_i \subseteq \mathcal{P}$, for all $i \in \{0, ..., \ell\}$. For the sake of conciseness, let $E_i = \mathit{Exec}(\Gamma, \sigma_i)$, $C_i = \mathit{Conf}(\Gamma, \sigma_i)$, and $L_i = L(C_i)$.

The construction will satisfy the following invariants for $i \in \{0, ..., \ell\}$:
(I1) C_i is safe.
(I2) $|P_i \setminus L_i| \geq (n-1)/(\log n)^{ci}$.
(I3) $\mathit{RMR}_{P_i \setminus L_i}(C_i) \geq i\,|P_i \setminus L_i| - i$.
(I4) For each process $p \in P_i \setminus L_i$: $\mathit{RMR}_p(C_i) \leq i$.
(I5) For each process $p \in P_i \setminus L_i$, $p^{\top}$ does not appear in σ_i.

Invariant (I2) for $i = \ell$ implies $|P_\ell \setminus L_\ell| \geq 2$. Hence, by (I3) there are at least two processes that each incur $\Omega(\ell) = \Omega(\log n / \log\log n)$ RMRs. Theorem 3 follows.

We now sketch how we construct σ_i and P_i inductively so that the invariants are satisfied. We start with $P_0 = \mathcal{P}$ and the initial configuration C_0. We then schedule processes in rounds. In round i, we choose a subset P_{i+1} of the processes in $P_i \setminus L_i$ and remove all processes in $\mathcal{P} \setminus (P_{i+1} \cup L_i)$ from the execution constructed so far. This does not affect any of the remaining processes, because C_i is safe. Then we schedule the processes in P_{i+1} in such a way that each of them incurs an RMR, and only a small fraction of them lose.

To decide which processes to remove and to schedule the remaining processes, we proceed as follows: First we let each process in $P_i \setminus L_i$ take sufficiently many steps until it is poised to incur an RMR. It is not hard to see that in an execution in which no process incurs an RMR, processes do not learn about each other, so the resulting configuration, D_i, is again safe. Moreover, in a safe configuration processes only know about lost processes, so they cannot lose.

We then distinguish between a high contention write case, where a majority of processes are poised to write to few registers, and a low contention case, where either many registers are covered by processes poised to write, or a majority of processes are poised to read. Let S_i be the set of registers processes in $P_i \setminus L_i$ are poised to access in configuration D_i. The high contention write case occurs if there are few such registers and a majority of processes are poised to write, i.e., $|S_i| = O(|P_i \setminus L_i| / \log n)$, and otherwise the low contention write case occurs.

In the low contention write case, we choose a set Q_i of processes, which contains for each register $r \in S_i$ at most one process poised to write to r in D_i. We consider the step s_p each process $p \in Q_i$ is poised to take. We then create a directed graph G with processes as vertices, and an edge from p to q if in the resulting configuration (I) due to s_p or s_q process p knows q, or (II) due to step s_p process q is not hidden. Each application of rule (I) must be paid for by RMRs in the execution, and for each application of (II) a process p must overwrite some process q. As a result graph G is sufficiently spares, and by Turán's theorem [44] we obtain a large independent set J. We let each process $p \in J$ take one step, s_p, and erase all remaining processes that haven't lost yet from the execution. It is not hard to see that no process loses in any of the steps added, the resulting configuration is safe (this follows from how we added edges to G) and, because of the sparsity of the graph, a sufficiently large number of processes survive. From that we obtain Invariants (I1) and (I2). Since each process p performs an RMR in step s_p and only local steps before that, we get (I3) and (I4). Moreover, we don't abort any processes, so (I5) is true.

In the high contention write case, we erase all readers from the execution. For each register $r \in S_i$, let W_r denote the set of processes poised to write to r. Since this is a high contention case, $|W_r|$ is large for most registers r. For each register r with sufficiently large $|W_r|$, we choose two distinct processes $a, b \in W_r$.

We then argue that, after erasing some $O(\log n)$ processes, we obtain a configuration D_i' and an $\{a, b\}$-only schedule σ such that in execution $Exec(D_i', \sigma)$ processes a and b both lose and see no process other than those in L_i, which have lost already. The argument is based on Lemma 8, but quite involved. We now let, starting from D_i', all processes in $W_r \setminus \{a, b\}$ execute one step, in which they write to r. After that we schedule a and b as prescribed by σ. Then a and b will both first write to r, and thus overwrite the writes by all other processes in W_r, then continue to take steps and lose without seeing any processes that haven't lost, yet. As a result, all processes in $W_r \setminus \{a, b\}$ have taken a step but are now hidden, two processes (a and b) have lost, and $O(\log n)$ processes have been removed. It is not hard to see that the resulting configuration is safe again. We repeat this for all registers r for which $|W_r|$ is large enough. Then, we let P_{i+1} denote the set of all surviving processes and C_{i+1} the resulting configuration.

Configuration C_{i+1} is safe, and sufficiently few processes are removed or have lost so that (I1) and (I2) remain true. Moreover, each process that does not lose performs exactly one RMR, so (I3) and (I4) are true. (I5) is true because all processes that received the abort signal lost.

References

1 Zahra Aghazadeh, Wojciech Golab, and Philipp Woelfel. Making objects writable. In *Proceedings of the 33rd SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 385–395, 2014. doi:10.1145/2611462.2611483.

2 Zahra Aghazadeh and Philipp Woelfel. Space- and time-efficient long-lived test-and-set objects. In *Proceedings of 18th International Conference On Principles Of Distributed Systems (OPODIS)*, pages 404–419, 2014. doi:10.1007/978-3-319-14472-6_27.

3 Zahra Aghazadeh and Philipp Woelfel. Upper bounds for boundless tagging with bounded objects. In *Proceedings of the 30th International Symposium on Distributed Computing (DISC)*, pages 442–457, 2016. doi:10.1007/978-3-662-53426-7_32.

4 Marcos Aguilera, Svend Frølund, Vassos Hadzilacos, Stephanie Lorraine Horn, and Sam Toueg. Abortable and query-abortable objects and their efficient implementation. In *Proceedings of the 26th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 23–32, 2007.

5 Dan Alistarh and James Aspnes. Sub-logarithmic test-and-set against a weak adversary. In *Proceedings of the 25th International Symposium on Distributed Computing (DISC)*, pages 97–109, 2011.

6 Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Morteza Zadimoghaddam. Optimal-time adaptive strong renaming, with applications to counting. In *Proceedings of the 30th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 239–248, 2011.

7 Dan Alistarh, James Aspnes, Seth Gilbert, and Rachid Guerraoui. The complexity of renaming. In *Proceedings of the 52nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 718–727, 2011. `doi:10.1109/FOCS.2011.66`.

8 Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast randomized test-and-set and renaming. In *Proceedings of the 24th International Symposium on Distributed Computing (DISC)*, pages 94–108, 2010.

9 James H. Anderson and Yong-Jik Kim. Adaptive mutual exclusion with local spinning. In *Proceedings of the 14th International Symposium on Distributed Computing (DISC)*, pages 29–43, 2000.

10 James H. Anderson and Yong-Jik Kim. An improved lower bound for the time complexity of mutual exclusion. *Distributed Computing*, 15:221–253, 2002.

11 T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1:6–16, 1990.

12 Hagit Attiya, Rachid Guerraoui, Danny Hendler, and Petr Kuznetsov. The complexity of obstruction-free implementations. *Journal of the ACM*, 56(4):24:1–24:33, 2009. `doi:10.1145/1538902.1538908`.

13 Hagit Attiya, Danny Hendler, and Philipp Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing (STOC)*, pages 217–226, 2008.

14 Michael Bender and Seth Gilbert. Mutual exclusion with $O(\log^2 \log n)$ amortized work. In *Proceedings of the 52nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 728–737, 2011.

15 Harry Buhrman, Alessandro Panconesi, Riccardo Silvestri, and Paul Vitányi. On the importance of having an identity or, is consensus really universal? *Distributed Computing*, 18(3):167–176, 2006. `doi:10.1007/s00446-005-0121-z`.

16 Robert Danek and Wojciech Golab. Closing the complexity gap between FCFS mutual exclusion and mutual exclusion. *Distributed Computing*, 23(2):87–111, 2010. `doi:10.1007/s00446-010-0096-2`.

17 Robert Danek and Hyonho Lee. Brief announcement: Local-spin algorithms for abortable mutual exclusion and related problems. In *Proceedings of the 22nd International Symposium on Distributed Computing (DISC)*, pages 512–513, 2008. `doi:10.1007/978-3-540-87779-0_41`.

18 E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8:569, 1965.

19 Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6):779–805, 1997. `doi:10.1145/268999.269000`.

20 Wayne Eberly, Lisa Higham, and Jolanta Warpechowska-Gruca. Long-lived, fast, wait-free renaming with optimal name space and high throughput. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC)*, pages 149–160, 1998.

21 Aryaz Eghbali and Philipp Woelfel. An almost tight RMR lower bound for abortable test-and-set. *CoRR*, abs/1805.04840, 2018. `arXiv:1805.04840`.

22 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985. `doi:10.1145/3149.214121`.

23 George Giakkoupis and Philipp Woelfel. On the time and space complexity of randomized test-and-set. In *Proceedings of the 31st SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 19–28, 2012. doi:10.1145/2332432.2332436.

24 George Giakkoupis and Philipp Woelfel. A tight RMR lower bound for randomized mutual exclusion. In *Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC)*, pages 983–1002, 2012. doi:10.1145/2213977.2214066.

25 George Giakkoupis and Philipp Woelfel. Randomized mutual exclusion with constant amortized RMR complexity on the DSM. In *Proceedings of the 55nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2014. To appear.

26 George Giakkoupis and Philipp Woelfel. Randomized abortable mutual exclusion with constant amortized RMR complexity on the CC model. In *Proceedings of the 36th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 221–229, 2017. doi:10.1145/3087801.3087837.

27 Wojciech Golab, Danny Hendler, and Philipp Woelfel. An $O(1)$ RMRs leader election algorithm. *SIAM Journal on Computing*, 39(7):2726–2760, 2010.

28 Wojciech M. Golab, Vassos Hadzilacos, Danny Hendler, and Philipp Woelfel. Constant-RMR implementations of cas and other synchronization primitives using read and write operations. In *Proceedings of the 26th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 3–12, 2007.

29 Wojciech M. Golab, Vassos Hadzilacos, Danny Hendler, and Philipp Woelfel. RMR-efficient implementations of comparison primitives using read and write operations. *Distributed Computing*, 25(2):109–162, 2012. doi:10.1007/s00446-011-0150-8.

30 Danny Hendler and Philipp Woelfel. Randomized mutual exclusion in $O(\log N/\log\log N)$ RMRs. In *Proceedings of the 28th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 26–35, 2009.

31 Danny Hendler and Philipp Woelfel. Adaptive randomized mutual exclusion in sub-logarithmic expected time. In *Proceedings of the 29th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 141–150, 2010.

32 Danny Hendler and Philipp Woelfel. Randomized mutual exclusion with sub-logarithmic RMR-complexity. *Distributed Computing*, 24(1):3–19, 2011. doi:10.1007/s00446-011-0128-6.

33 Prasad Jayanti. Adaptive and efficient abortable mutual exclusion. In *Proceedings of the 22nd SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 295–304, 2003. doi:10.1145/872035.872079.

34 Prasad Jayanti, Srdjan Petrovic, and Neha Narula. Read/write based fast-path transformation for FCFS mutual exclusion. In *31st Conference on Current Trends in Theory and Practice of Informatics (SOFSEM)*, pages 209–218, 2005.

35 Y.-J. Kim and J. Anderson. A time complexity bound for adaptive mutual exclusion. In *Proceedings of the 15th International Symposium on Distributed Computing (DISC)*, pages 1–15, 2001.

36 Yong-Jik Kim and James H. Anderson. Nonatomic mutual exclusion with local spinning. *Distributed Computing*, 19(1):19–61, 2006. doi:10.1007/s00446-006-0003-z.

37 Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient synchronization on multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems*, 10(4):579–601, 1988. doi:10.1145/48022.48024.

38 Hyonho Lee. Transformations of mutual exclusion algorithms from the cache-coherent model to the distributed shared memory model. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS)*, pages 261–270, 2005. doi:10.1109/ICDCS.2005.83.

39 Hyonho Lee. Fast local-spin abortable mutual exclusion with bounded space. In *Proceedings of 14th International Conference On Principles Of Distributed Systems (OPODIS)*, pages 364–379, 2010. doi:10.1007/978-3-642-17653-1_27.

40 Hyonho Lee. *Local-spin Abortable Mutual Exclusion.* PhD thesis, University of Toronto, 2011.

41 Alessandro Panconesi, Marina Papatriantafilou, Philippas Tsigas, and Paul M. B. Vitányi. Randomized naming using wait-free shared variables. *Distributed Computing*, 11(3):113–124, 1998.

42 Abhijeet Pareek and Philipp Woelfel. RMR-efficient randomized abortable mutual exclusion. In *Proceedings of the 26th International Symposium on Distributed Computing (DISC)*, pages 267–281, 2012. doi:10.1007/978-3-642-33651-5_19.

43 Michael L Scott. Non-blocking timeout in scalable queue-based spin locks. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 31–40. ACM, 2002.

44 Paul Turán. Eine extremalaufgabe aus der graphentheorie. *Mat. Fiz. Lapok*, 48(436-452):61, 1941.