Time and Space Lower Bounds for Non-blocking Implementations*
(Preliminary version)

Prasad Jayanti[†] King Tan[‡] Sam Toueg[‡]

Abstract

We show the following time and space complexity lower bounds. Let $\mathcal{I}$ be any randomized non-blocking n-process implementation of any object in A from any combination of objects in set B, where $A = \{$*increment*, *store-conditional bit*, *compare&swap*, *bounded-counter*, *single-writer atomic snapshot*, *fetch&add*$\}$, and $B = \{$*resettable consensus*, *register*, *swap register*$\}$. The space complexity of $\mathcal{I}$ is at least $n-1$. Moreover, if $\mathcal{I}$ is deterministic, both its time and space complexity are at least $n-1$. These lower bounds hold even if objects used in the implementation are of unbounded size.

This improves on some of the $\Omega(\sqrt{n})$ space complexity lower bounds of Fich, Herlihy & Shavit [FHS93]. It also shows the near optimality of some known wait-free implementations in terms of space complexity.

*Research partially supported by NSF grants CCR-9402894 and CCR-9410421, DARPA/NASA Ames grant NAG-2-593, and Dartmouth College Startup grant.

[†]Sudikoff Laboratory for Computer Science, Dartmouth College, Hanover, NH 03755, USA.

[‡]Department of Computer Science, Upson Hall, Cornell University, Ithaca, NY 14853, USA.

1 Introduction

Non-blocking and wait-free implementations of shared objects have been the subject of much research. While there have been several results on when such implementations are feasible and when they are not, results establishing their intrinsic time and space requirements are relatively scarce, especially for randomized implementations. In this paper, we present a technique by which one can obtain a linear lower bound on the space complexity of several randomized non-blocking implementations. The technique also yields a linear lower bound on the time complexity of several deterministic non-blocking implementations.

Specifically, our results are as follows. Let $\mathcal{I}$ be any randomized non-blocking n-process implementation of any object in set A from any combination of objects in set B, where $A = \{$*increment*, *store-conditional bit*, *compare&swap*, *bounded-counter*, *single-writer atomic snapshot*, *fetch&add*$\}$, and $B = \{$*resettable consensus*, *register*, *swap register* $\}$. The space complexity of $\mathcal{I}$ is at least $n-1$. Moreover, if $\mathcal{I}$ is deterministic, both its time and space complexity are at least $n-1$. These lower bounds hold even if objects used in the implementation are of unbounded size.

Some of the results in this paper improve known lower bounds, while others are completely new. In particular, Fich, Herlihy & Shavit proved a $\Omega(\sqrt{n})$ space complexity lower bound for a randomized non-blocking n-process implementation of *binary consensus* from *registers* and *swap registers* [FHS93]. Using this result, they showed that any randomized non-blocking n-process implementation of *compare&swap*, or *fetch&add*, or *bounded-counter* from any combination of *regis-*

ters and *swap registers* requires $\Omega(\sqrt{n})$ instances of such objects. Our results on *compare&swap*, *fetch&add*, and *bounded-counter* are stronger in two ways: (i) They show that at least $n-1$ objects are necessary, and (ii) They show that $n-1$ objects are needed even if the implementation is free to use *resettable consensus objects*, besides the *registers* and *swap registers* allowed by [FHS93].

The results presented in this paper also imply that the following deterministic implementations in the literature are almost space-optimal.

1. Afek *et al* give two wait-free implementations of a *single writer atomic snapshot object* (consisting of n segments, each one written by a different process): one from *unbounded registers* and one from *bounded registers* [AAD+93]. The one that uses *unbounded registers* is of space complexity n. We prove a lower bound of $n-1$.

2. Aspnes gives a wait-free implementation of a n-process *bounded-counter* from a single instance of a *single writer atomic snapshot object* [Asp90]. Combined with the above result of Afek *et al*, this implies that *bounded-counter* can be implemented from n unbounded *registers*. We prove that at least $n-1$ *registers* are necessary when the *bounded-counter* is a modulo k counter, where $k \geq 2n$.

In both cases above, the lower bound of $n-1$ is particularly appealing because it applies to even randomized non-blocking implementations while the upper bound of n holds for deterministic wait-free implementations.

In fact, the lower bounds proved in this paper apply not just to non-blocking implementations, but to any implementation satisfying a progress condition that we call *solo-finish*. Roughly speaking, a deterministic implementation is solo-finish if at every configuration C in a system execution the following holds for all processes p: if p runs alone from C, p's operation on the implemented object will eventually complete. For a randomized implementation to be considered solo-finish we require that for all C and p if p runs alone from C, there is at least one sequence of outcomes for p's coin tosses that will enable p to complete its operation on the implemented object.

It is well-known that a wait-free implementation is also non-blocking. It is clear that a non-blocking implementation is also solo-finish. Thus, the lower bounds that we prove here for solo-finish implementations also apply to non-blocking and wait-free implementations.

2 Road Map

We give an informal model in Section 3. In Section 4, we prove a special case of our results, namely, that the space complexity of any solo-finish randomized implementation (and the time complexity of any solo-finish deterministic implementation) of an *increment object* from *resettable consensus objects*, *swap registers* and *registers* is at least $n-1$. This proof illustrates the basic technique common to all our lower bound proofs. In Section 5, we extend this result by identifying the properties of *increment* that are used in the proof, and showing that a variety of other objects (such as *compare&swap*, *store-conditional bit*, and *single-writer atomic snapshot*) also possess these properties, and so the lower bound applies to implementations of these objects as well. In Appendix A, we present the sequential specifications of the types of objects considered in this paper.

3 Informal Model

3.1 Type

A *type* is a tuple (OP, RES, Q, δ), where OP is a set of operations, RES is a set of responses, Q is a set of states, and $\delta : Q \times OP \rightarrow Q \times RES$ is a function, known as the *sequential specification* of the type. Intuitively, if $\delta(\sigma, op) = (\sigma', res)$ it means the following: applying the operation op to an object of this type in state σ causes the object to move to state σ' and return the response res.

3.2 Implementation

A *randomized implementation of object $\mathcal{O}$* is specified by the following elements:

- the type and the initial value of $\mathcal{O}$ (the initial value of $\mathcal{O}$ is a state of its type).

- a set of objects $O_1, \ldots, O_m$ (from which $\mathcal{O}$ is implemented), their types and their initial values.

- a set of processes $p_1, \ldots, p_n$ that may access $\mathcal{O}$.

- a set of *randomized access procedures* Apply($p_i, op, \mathcal{O}$), for $p_i \in \{p_1, \ldots, p_n\}$ and $op \in OP$, where OP is the set of operations associated with the type of $\mathcal{O}$.

The access procedure Apply($p_i, op, \mathcal{O}$) specifies how p_i should execute the operation op on $\mathcal{O}$ in terms of operations on $O_1, \ldots, O_m$. The value returned by the procedure is deemed to be the response from $\mathcal{O}$. We call $O_1, \ldots, O_m$ the *base objects* of the implementation. The *space complexity* of the implementation is m.

The definitions presented in the rest of this section are with respect to a system that consists of processes $p_1, \ldots, p_n$ and an implemented object $\mathcal{O}$ that $p_1, \ldots, p_n$ may access. We denote such a system by $(p_1, \ldots, p_n; \mathcal{O})$. Each p_i has a distinguished input variable *op-list*$_i$. This variable is initialized (by the user of the system) with any infinite sequence of operations $op_1, op_2, \ldots$ where each op_j is an operation supported by $\mathcal{O}$. Each p_i performs the following actions repeatedly forever: obtain the next operation op from *op-list*$_i$ and execute the access procedure Apply($p_i, op, \mathcal{O}$). Let *comp*$_i$ denote the sequence of completed operations and *rem*$_i$ denote the sequence of operations that have not yet been completed. The *state of process* p_i is specified by (i) *comp*$_i$, (ii) the values of variables associated with p_i's access procedures, namely, Apply($p_i, op, \mathcal{O}$) for all $op \in OP$, and (iii) p_i's program counter.

A process p_i executes an access procedure Apply($p_i, op, \mathcal{O}$) in *steps*. Each step consists of the following actions, all of which occur together atomically:

- p_i tosses a coin. Let *toss-outcome* $\in$ COINSPACE denote the outcome of this toss.

- *toss-outcome* and p_i's present state uniquely determine an operation *oper* and a base object O_j that *oper* should be applied to. Accordingly, p_i applies *oper* to O_j.

- O_j changes state and returns a response. The new state of O_j and the response are uniquely determined by the sequential specification of O_j.

- The response from O_j, together with *toss-outcome* and p_i's present state, uniquely determine the new state of p_i. It is also possible for the procedure Apply($p_i, op, \mathcal{O}$) to terminate, returning some response.

A *configuration* of $(p_1, \ldots, p_n; \mathcal{O})$ is a tuple $(\sigma_1, \ldots, \sigma_n, rem_1, \ldots, rem_n, \tau_1, \ldots, \tau_m)$, where σ_i is the state of p_i and τ_j is the state of base object O_j. Notice that the *initial configuration* is uniquely determined by an assignment of infinite sequences of operations to the input variables *op-list*$_i$ $(1 \le i \le n)$. An *execution* of $(p_1, \ldots, p_n; \mathcal{O})$ is a sequence $C_0, C_1, C_2, \ldots$ of configurations such that C_0 is an initial configuration and C_{k+1} is the configuration that results when some process performs a step in configuration C_k.

A *schedule* is a finite or an infinite sequence $[p_{i_1}, t_1], [p_{i_2}, t_2], \ldots$ where each p_{i_j} is from $\{p_1, \ldots, p_n\}$ and each t_j is from COINSPACE. If C is a configuration and $\alpha = [p_{i_1}, t_1], [p_{i_2}, t_2], \ldots$ is a schedule, EXEC(C, α) denotes the unique sequence $C = C_0, C_1, C_2, \ldots$ of configurations such that each C_{k+1} results from C_k when p_{i_k} takes a step in which the outcome of p_{i_k}'s toss is t_k. A configuration C is *reachable* if there is some initial configuration C_0 and a finite schedule α such that the configuration at the end of EXEC(C_0, α) is C.

3.3 Linearizability

An implementation of $\mathcal{O}$ is *linearizable* if in each execution of $(p_1, \ldots, p_n; \mathcal{O})$, each operation on $\mathcal{O}$ appears to take effect at some "instant" between its invocation and response [HW90]. In this paper, we restrict our attention to linearizable implementations.

3.4 Solo-finish: a progress property

An implementation whose access procedures never terminate is trivially linearizable. Such an implementation, however, can hardly be useful. Thus, in addition to linearizability, implementations often guarantee certain progress properties. *Wait-freedom* and *non-blocking* are the progress conditions that received the most attention recently. Here we state a new, weaker progress property that we call "solo-finish". Informally, an implementation has solo-finish property if for each configuration C and each process p the following holds: if p runs alone from configuration C, then there is at least one sequence of outcomes for p's coin tosses that will enable p to complete an operation on the implemented object.

More formally, a *randomized implementation of $\mathcal{O}$ is solo-finish* if, for all reachable configurations C and all processes p_i, there is some finite schedule $\alpha = [p_i,t_1],[p_i,t_2],\ldots,[p_i,t_k]$ such that p_i completes an operation on $\mathcal{O}$ during $\text{EXEC}(C,\alpha)$.

The lower bounds proved in this paper apply to solo-finish (and therefore to non-blocking and wait free) implementations.

3.5 Deterministic time complexity

A *deterministic implementation* is a special case of a randomized implementation for which COINSPACE, the set of possible outcomes for a coin toss, is a singleton set. The *solo-finish time complexity of a deterministic implementation of $\mathcal{O}$ is at least k* if for some reachable configuration C and process p_i the following holds: for all schedules α of length $k-1$ that contain only p_i, p_i does not complete an operation on $\mathcal{O}$ during $\text{EXEC}(C,\alpha)$.

3.6 Notation

For a schedule α, $|\alpha|$ denotes its length. We say *α contains process p* if, for some t, $[p,t]$ is in the sequence α. $\text{PSET}(\alpha)$ denotes the set of all processes p such that α contains p. If Σ is a set, Σ^* denotes the set of all *finite* sequences of elements from Σ. The empty sequence, denoted by ϵ, is a member of Σ^* for all sets Σ (even if $\Sigma = \emptyset$, the empty set).

4 The lower bound

We illustrate our lower bound technique by proving that the space complexity of any randomized solo-finish implementation and the time complexity of any deterministic solo-finish implementation of an *increment object*, shared by processes $p_1,\ldots,p_n$, are both at least $n-1$ if base objects are restricted to be (any combination of) *registers*, *swap registers*, and *resettable consensus objects*. (The specifications of these various objects are presented in the appendix.) In the next section, we explain how this technique generalizes to implementations of other types of objects.

Theorem 1 *Let $\mathcal{O}$ be a randomized implementation of an increment object, initialized to 0, from registers, swap registers, and resettable consensus objects. Let $p_1,\ldots,p_n$ be the processes that may access $\mathcal{O}$. If the implementation is linearizable and solo-finish:*

1. *Its space complexity is at least $n-1$.*
2. *If the implementation is deterministic, its solo-finish time complexity is at least $n-1$.*

Below, we prove a key lemma from which both parts of the theorem will be immediate.[1] Let C_0 be the initial configuration of $(p_1,\ldots,p_n;\mathcal{O})$ corresponding to the following assignment of operation sequences to the input variables: for $1 \leq i \leq n-1$, *op-list$_i$* is an infinite sequence of *increment* operations, and *op-list$_n$* is an infinite sequence of *read* operations. Let COINSPACE be any non-empty set.

Lemma 1 *Let $n \geq 1$. For all k, $0 \leq k \leq n-1$, Statement S_k, described in Figure 1, is true.*

Proof By induction. Below, we let $S_k : j$ denote the j^{th} part of Statement S_k.

Base case. We show that S_0 is true.

Let $\Lambda_0 = \Sigma_0 = \Pi_0 = \epsilon$ and let $\mathcal{S}_0 = \emptyset$. It is easy to verify that all of $S_0 : 1-7$ are true. Hence the base case.

Induction step. Suppose $0 \leq k \leq n-2$ and S_k is true. We now show that S_{k+1} is true.

Let $\Lambda_k, \Sigma_k, \Pi_k, \mathcal{S}_k$ be so defined as to make Statement S_k true. We will demonstrate the existence of schedules $\lambda_{k+1}, [p_{k+1}, t_{k+1}], \pi_{k+1}$ and a base object B_{k+1} such that for $\Lambda_{k+1} = \Lambda_k\lambda_{k+1}$, $\Sigma_{k+1} = [p_{k+1},t_{k+1}]\Sigma_k$, $\Pi_{k+1} = \Pi_k\pi_{k+1}$ and $\mathcal{S}_{k+1} = \mathcal{S}_k \cup \{B_{k+1}\}$, Statement S_{k+1} is true. We will do this through the following steps.

[1]There is a simpler proof if we are interested only in the space complexity.

There are schedules Λ_k, Σ_k, Π_k, and a set $\mathcal{S}_k$ of base objects such that the following hold:

1. The schedules Λ_k and Σ_k do not contain p_n: *i.e.*, $\Lambda_k, \Sigma_k \in (\{p_1, p_2, \ldots, p_{n-1}\} \times \text{COINSPACE})^*$.
2. For $k = 0$, $\Sigma_k = \epsilon$. For $k \geq 1$, Σ_k is of the form $[p_{i_k}, t_k][p_{i_{k-1}}, t_{k-1}] \ldots [p_{i_1}, t_1]$, where $p_{i_1}, \ldots, p_{i_k}$ are distinct processes.
3. $\Pi_k \in (\{p_n\} \times \text{COINSPACE})^*$.
4. $|\mathcal{S}_k| = k$.
5. $\mathcal{S}_k$ is exactly the set of base objects that p_n accesses in $\text{EXEC}(C_0, \Lambda_k\Sigma_k\Pi_k)$.
6. In $\text{EXEC}(C_0, \Lambda_k\Sigma_k\Pi_k)$, p_n's first operation on $\mathcal{O}$ has either not completed or just completed.
7. Let $\mathcal{P}_k = \{p_1, p_2, \ldots, p_{n-1}\} - \text{PSET}(\Sigma_k)$. Let γ be any schedule in $(\mathcal{P}_k \times \text{COINSPACE})^*$. The state of each base object in $\mathcal{S}_k$ at the end of $\text{EXEC}(C_0, \Lambda_k\Sigma_k)$ is the same as its state at the end of $\text{EXEC}(C_0, \Lambda_k\gamma\Sigma_k)$.

Figure 1: Statement S_k

1. By S_k : 6, in $\text{EXEC}(C_0, \Lambda_k\Sigma_k\Pi_k)$, p_n's first operation on $\mathcal{O}$ has either not completed or just completed. Let $\pi \in (\{p_n\} \times \text{COINSPACE})^*$ be such that p_n just completes its first operation on $\mathcal{O}$ in $\text{EXEC}(C_0, \Lambda_k\Sigma_k\Pi_k\pi)$, returning some value *res*. Since the implementation is solo-finish, π exists.

2. **Claim 1** *In* $\text{EXEC}(C_0, \Lambda_k\Sigma_k\Pi_k\pi)$, p_n *accesses a base object not in* $\mathcal{S}_k$.

   Proof Suppose the claim is false. Let p_l be any process in $\mathcal{P}_k = \{p_1, p_2, \ldots, p_{n-1}\} - \text{PSET}(\Sigma_k)$. Since $|\text{PSET}(\Sigma_k)| = k$ and $k \leq n-2$, $\mathcal{P}_k$ is non-empty and we can pick p_l. Let $\gamma \in (\{p_l\} \times \text{COINSPACE})^*$ be such that there are at least $res + 1$ more completed increment operations on $\mathcal{O}$ in $\text{EXEC}(C_0, \Lambda_k\gamma)$ than in $\text{EXEC}(C_0, \Lambda_k)$. Since the implementation is solo-finish, γ exists. We now have the following facts.

   F1. The state of each base object in $\mathcal{S}_k$ at the end of $\text{EXEC}(C_0, \Lambda_k\Sigma_k)$ is the same as its state at the end of $\text{EXEC}(C_0, \Lambda_k\gamma\Sigma_k)$.

   This follows from part S_k : 7 of the induction hypothesis.

   F2. The executions $E_1 = \text{EXEC}(C_0, \Lambda_k\Sigma_k\Pi_k\pi)$ and $E_2 = \text{EXEC}(C_0, \Lambda_k\gamma\Sigma_k\Pi_k\pi)$ are indistinguishable to p_n. (That is, p_n's state is the same at the end of E_1 and E_2.)

   This follows from: (i) the schedules $\Lambda_k\Sigma_k$ and $\Lambda_k\gamma\Sigma_k$ do not contain p_n, (ii) the schedule $\Pi_k\pi$ contains only p_n, (iii) the only base objects accessed by p_n in E_1 are the ones in $\mathcal{S}_k$ (by our assumption that Claim 1 is false), and (iv) the states of base objects in $\mathcal{S}_k$ are the same at the end of $\text{EXEC}(C_0, \Lambda_k\Sigma_k)$ and $\text{EXEC}(C_0, \Lambda_k\gamma\Sigma_k)$ (this is F1).

   F3. In $\text{EXEC}(C_0, \Lambda_k\gamma\Sigma_k\Pi_k\pi)$, p_n's first operation on $\mathcal{O}$ completes and returns *res*.

   This follows from F2 and the definition of π in Step 1.

   F4. In $\text{EXEC}(C_0, \Lambda_k\gamma\Sigma_k\Pi_k\pi)$, p_n's first operation on $\mathcal{O}$ cannot return *res*.

   By definition of γ, the number of completed increments in $\text{EXEC}(C_0, \Lambda_k\gamma\Sigma_k)$ is at least $res + 1$. Since the implementation is linearizable, it follows that p_n's first operation on $\mathcal{O}$ (which is a read operation) cannot return *res*.

F3 and F4 are contradictory. Hence the claim. □

3. Let π_{k+1} be the shortest prefix of π such that in $\text{EXEC}(C_0, \Lambda_k\Sigma_k\Pi_k\pi_{k+1})$ p_n accesses a base object not in $\mathcal{S}_k$. (By Claim 1, π_{k+1} exists.) Let B_{k+1} denote this base object. Define $\mathcal{S}_{k+1} = \mathcal{S}_k \cup \{B_{k+1}\}$ and $\Pi_{k+1} = \Pi_k\pi_{k+1}$. Since $|\mathcal{S}_k| = k$ and $B_{k+1} \notin \mathcal{S}_k$, we have $|\mathcal{S}_{k+1}| = k+1$. Since Π_k and π_{k+1} are both from $(\{p_n\} \times \text{COINSPACE})^*$, we have $\Pi_{k+1} \in (\{p_n\} \times \text{COINSPACE})^*$. Thus, we have established parts $S_{k+1} : 3$ and $S_{k+1} : 4$ of the induction step.

4. <u>CONSTRUCTION</u> We will now define schedules λ_{k+1} and $[p_{i_{k+1}}, t_{k+1}]$ based on the type of B_{k+1}.

   <u>Case 1</u> B_{k+1} is a register or a swap register.

   <u>Subcase 1a</u> There is some non-empty schedule $\lambda \in (\mathcal{P}_k \times \text{COINSPACE})^*$ such that the last step in $\text{EXEC}(C_0, \Lambda_k\lambda)$ is a write to B_{k+1}.

   Define λ_{k+1} and $[p_{i_{k+1}}, t_{k+1}]$ so that $\lambda = \lambda_{k+1}.[p_{i_{k+1}}, t_{k+1}]$.

   <u>Subcase 1b</u> There is no such λ.

   Define λ_{k+1} to be ϵ and $[p_{i_{k+1}}, t_{k+1}]$ to be any element of $\mathcal{P}_k \times \text{COINSPACE}$.

   <u>Case 2</u> B_{k+1} is a resettable consensus object.

   <u>Subcase 2a</u> There is some non-empty schedule $\lambda \in (\mathcal{P}_k \times \text{COINSPACE})^*$ such that the last step in $\text{EXEC}(C_0, \Lambda_k\lambda)$ is a reset operation on B_{k+1}.

   Define λ_{k+1} and $[p_{i_{k+1}}, t_{k+1}]$ so that $\lambda = \lambda_{k+1}.[p_{i_{k+1}}, t_{k+1}]$.

   <u>Subcase 2b</u> There is no such λ. However, there is some non-empty schedule $\lambda' \in (\mathcal{P}_k \times \text{COINSPACE})^*$ such that the last step in $\text{EXEC}(C_0, \Lambda_k\lambda')$ is a propose operation on B_{k+1}.

   Define λ_{k+1} to be λ' and $[p_{i_{k+1}}, t_{k+1}]$ to be any element of $\mathcal{P}_k \times \text{COINSPACE}$.

   <u>Subcase 2c</u> Neither λ nor λ' exists.

   Define λ_{k+1} to be ϵ and $[p_{i_{k+1}}, t_{k+1}]$ to be any element of $\mathcal{P}_k \times \text{COINSPACE}$.

5. Define Λ_{k+1} as $\Lambda_k\lambda_{k+1}$ and define Σ_{k+1} as $[p_{i_{k+1}}, t_{k+1}]\Sigma_k$. By definition, $\lambda_{k+1} \in (\mathcal{P}_k \times \text{COINSPACE})^*$ and $[p_{i_{k+1}}, t_{k+1}] \in \mathcal{P}_k \times \text{COINSPACE}$. From this and the induction hypothesis, we have $\Lambda_{k+1}, \Sigma_{k+1} \in (\{p_1, p_2, \ldots, p_{n-1}\} \times \text{COINSPACE})^*$. This establishes part $S_{k+1} : 1$ of the induction step.

   By induction hypothesis, Σ_0 is ϵ and Σ_k is of the form $[p_{i_k}, t_k][p_{i_{k-1}}, t_{k-1}]\ldots[p_{i_1}, t_1]$ (for $k \geq 1$), where $p_{i_1}, \ldots, p_{i_k}$ are distinct processes. Furthermore, since $[p_{i_{k+1}}, t_{k+1}] \in \mathcal{P}_k \times \text{COINSPACE}$ and $\mathcal{P}_k$ does not include any process from $\text{PSET}(\Sigma_k)$, it follows that Σ_{k+1} is of the form $[p_{i_{k+1}}, t_{k+1}]\ldots[p_{i_1}, t_1]$, where $p_{i_1}, \ldots, p_{i_{k+1}}$ are distinct processes. This establishes part $S_{k+1} : 2$ of the induction step.

6. **Claim 2** *Let* $\mathcal{P}_{k+1} = \{p_1, p_2, \ldots, p_{n-1}\} - \text{PSET}(\Sigma_{k+1})$. *Let* γ *be any schedule from* $(\mathcal{P}_{k+1} \times \text{COINSPACE})^*$. *Then we have:*

   (a) The state of B_{k+1} *is the same at the end of* $\text{EXEC}(C_0, \Lambda_k\lambda_{k+1}[p_{i_{k+1}}, t_{k+1}])$ *and* $\text{EXEC}(C_0, \Lambda_k\lambda_{k+1}\gamma[p_{i_{k+1}}, t_{k+1}])$.

   (b) The state of B_{k+1} *is the same at the end of* $\text{EXEC}(C_0, \Lambda_k\lambda_{k+1}[p_{i_{k+1}}, t_{k+1}]\Sigma_k)$ *and* $\text{EXEC}(C_0, \Lambda_k\lambda_{k+1}\gamma[p_{i_{k+1}}, t_{k+1}]\Sigma_k)$.

   (c) For each base object $B \in \mathcal{S}_k$, *B's state is the same at the end of* $\text{EXEC}(C_0, \Lambda_k\Sigma_k)$, $\text{EXEC}(C_0, \Lambda_k\lambda_{k+1}[p_{i_{k+1}}, t_{k+1}]\Sigma_k)$, *and* $\text{EXEC}(C_0, \Lambda_k\lambda_{k+1}\gamma[p_{i_{k+1}}, t_{k+1}]\Sigma_k)$.

   Proof The proof is based on the construction. See [JTT96] for details. □

7. Part $S_{k+1} : 7$ of the induction step is immediate from parts (b) and (c) of Claim 2 and our earlier definitions of Λ_{k+1}, Σ_{k+1}, and $\mathcal{S}_{k+1}$ as $\Lambda_k\lambda_{k+1}$, $[p_{i_{k+1}}, t_{k+1}]\Sigma_k$, and $\mathcal{S}_k \cup \{B_{k+1}\}$, respectively.

8. Recall the definition of π_{k+1} from Steps 1 and 3, and the statement of Claim 1.

   Claim 3 *(a)* $\mathcal{S}_k \cup \{B_{k+1}\}$ *is exactly the set of base objects that* p_n *accesses in* $\text{EXEC}(C_0, \Lambda_{k+1}\Sigma_{k+1}\Pi_k\pi_{k+1})$.

   (b) In $\text{EXEC}(C_0, \Lambda_{k+1}\Sigma_{k+1}\Pi_k\pi_{k+1})$, p_n*'s first operation on* $\mathcal{O}$ *has either not completed or just completed.*

Proof Observe that (1) $\text{EXEC}(C_0, \Lambda_k\Sigma_k)$ and $\text{EXEC}(C_0, \Lambda_{k+1}\Sigma_{k+1})$ are indistinguishable to p_n (since neither $\Lambda_k\Sigma_k$ nor $\Lambda_{k+1}\Sigma_{k+1}$ contains p_n), and (2) For all $B \in \mathcal{S}_k$, B's state is the same at the end of $\text{EXEC}(C_0, \Lambda_k\Sigma_k)$ and $\text{EXEC}(C_0, \Lambda_{k+1}\Sigma_{k+1})$ (this is part (c) of Claim 2). By definition of π_{k+1}, in $\text{EXEC}(C_0, \Lambda_k\Sigma_k\Pi_k\pi_{k+1})$, it is only in the last step that p_n accesses a base object not in $\mathcal{S}_k$, and p_n's first operation on $\mathcal{O}$ has either not completed or just completed. This, together with Observations (1) and (2) stated above, implies the claim. □

9. The parts S_{k+1} : 5 and S_{k+1} : 6 of the induction step are immediate from Claim 3 and our earlier definition of Π_{k+1} as $\Pi_k\pi_{k+1}$.

We have proved all the seven parts of Statement S_{k+1}: parts 1 and 2 in Step 5, parts 3 and 4 in Step 3, parts 5 and 6 in Step 9, and part 7 in Step 7. This completes the induction step. Hence the lemma. □

The first part of Theorem 1 is immediate from part 4 of Statement S_{n-1}. To obtain the second part of the theorem, observe that a deterministic implementation can be viewed as a randomized implementation for which COINSPACE is a singleton set. Since Lemma 1 holds for any non-empty COINSPACE, Statement S_{n-1} is true for any deterministic implementation. By parts 4, 5, and 6 of Statement S_{n-1}, in $\text{EXEC}(C_0, \Lambda_{n-1}\Sigma_{n-1}\Pi_{n-1})$, we have: (i) $|\mathcal{S}_{n-1}| = n - 1$, (ii) p_n accesses all objects in $\mathcal{S}_{n-1}$, and (iii) p_n has either not completed or just completed its first operation on $\mathcal{O}$. This implies that the solo-finish time complexity is at least $n - 1$.

5 Generalization to implementations of other objects

In the proof of Lemma 1, the following were the only places where we used the fact that the object being implemented is an increment object: (i) In the initial assignment of operation sequences to the input variables *op-list*$_i$, (ii) In defining schedule γ in the proof of Claim 1 (see the first paragraph of that proof just before Fact F1 was stated), and (iii) in proving Fact F4 in Claim 1. Thus, to show that Lemma 1 also applies to implementations of other types of objects, all that is required is to specify the initial values for the input variables *op-list*$_i$ and define γ in Claim 1 so that Fact F4 in that claim is true. In the following, we show that this can be done for *modulo 2n counter* and *n-valued compare&swap.* In the full paper, we will show this also for *load-linked store-conditional bit* and *atomic snapshot.* It follows that the space complexity of a randomized implementation or the time complexity of a deterministic implementation of these objects from registers, swap registers, and resettable consensus objects is at least $n - 1$.

5.1 modulo $2n$ counter

For $1 \leq i \leq n - 1$, let *op-list*$_i$ be an infinite sequence of *increment* operations, and *op-list*$_n$ be an infinite sequence of *read* operations.

In the proof of Claim 1, let p_l be any process in $\{p_1, p_2, \ldots, p_{n-1}\} - \text{PSET}(\Sigma_k)$. Let $\gamma \in (\{p_l\} \times \text{COINSPACE})^*$ be the shortest schedule such that there are exactly n more completed increment operations on $\mathcal{O}$ in $\text{EXEC}(C_0, \Lambda_k\gamma)$ than in $\text{EXEC}(C_0, \Lambda_k)$. Since the implementation is solo-finish, γ exists.

We now make the following observations:

1. In $\text{EXEC}(C_0, \Lambda_k\Sigma_k)$, if a process completes an increment on $\mathcal{O}$ in the last $|\Sigma_k|$ steps, then it has no pending increment on $\mathcal{O}$. No process completes more than one increment on $\mathcal{O}$ in the last $|\Sigma_k|$ steps of $\text{EXEC}(C_0, \Lambda_k\Sigma_k)$.

   This follows from the fact that each process appears at most once in the schedule Σ_k.

2. For any execution E, let $\text{NP}(E)$ denote the number of pending increment operations on $\mathcal{O}$ in E. The sum of $\text{NP}(\text{EXEC}(\Lambda_k\Sigma_k))$ and the number of increments that completed in the last $|\Sigma_k|$ steps of $\text{EXEC}(\Lambda_k\Sigma_k)$ is at most $n - 1$.

   Follows from Observation 1 and the fact that the schedule $\Lambda_k\Sigma_k$ contains at most $n - 1$ processes.

3. The sum of $\text{NP}(\text{EXEC}(\Lambda_k\gamma\Sigma_k))$ and the number of increments that completed in the last $|\Sigma_k|$ steps of $\text{EXEC}(\Lambda_k\gamma\Sigma_k)$ is at most $n - 1$.

Similar to Observation 2.

4. For any execution E, let NCI(E) denote the number of completed increment operations on $\mathcal{O}$ in E. Let $\text{NCI}(\text{EXEC}(C_0, \Lambda_k)) = v$. In $\text{EXEC}(C_0, \Lambda_k\gamma\Sigma_k\Pi_k\pi)$, the value *res* that p_n's first operation on $\mathcal{O}$ (which is a read) returns is in the range $[v, v+n-1] \bmod 2n$.

   This follows from Proposition 1 and the following two chains of inequalities:

   $$\text{NCI}(\text{EXEC}(\Lambda_k\Sigma_k)) \geq \text{NCI}(\text{EXEC}(\Lambda_k)) = v$$

   $$\begin{aligned}&\text{NCI}(\text{EXEC}(\Lambda_k\Sigma_k)) + \text{NP}(\text{EXEC}(\Lambda_k\Sigma_k))\\ &= \text{NCI}(\text{EXEC}(\Lambda_k)) + \text{NP}(\text{EXEC}(\Lambda_k\Sigma_k)) +\\ &\quad \text{number of increments that completed in}\\ &\quad \text{the last } |\Sigma_k| \text{ steps of } \text{EXEC}(\Lambda_k\Sigma_k)\\ &\leq v + n - 1\end{aligned}$$

5. In $\text{EXEC}(C_0, \Lambda_k\gamma\Sigma_k\Pi_k\pi)$, p_n's first operation on $\mathcal{O}$ returns a value in the range $[v+n, v+2n-1] \bmod 2n$.

   This follows from Proposition 1 and the following two chains of inequalities:

   $$\text{NCI}(\text{EXEC}(\Lambda_k\gamma\Sigma_k)) \geq \text{NCI}(\text{EXEC}(\Lambda_k\gamma)) = v + n$$

   $$\begin{aligned}&\text{NCI}(\text{EXEC}(\Lambda_k\gamma\Sigma_k)) + \text{NP}(\text{EXEC}(\Lambda_k\gamma\Sigma_k))\\ &= \text{NCI}(\text{EXEC}(\Lambda_k\gamma)) + \text{NP}(\text{EXEC}(\Lambda_k\gamma\Sigma_k)) +\\ &\quad \text{number of increments that completed in}\\ &\quad \text{the last } |\Sigma_k| \text{ steps of } \text{EXEC}(\Lambda_k\gamma\Sigma_k)\\ &\leq (v+n) + (n-1)\end{aligned}$$

From the last two observations, we conclude that, in $\text{EXEC}(C_0, \Lambda_k\gamma\Sigma_k\Pi_k\pi)$, p_n's first operation on $\mathcal{O}$ cannot return *res*. Hence we have Fact F4 of Claim 1. We conclude that Lemma 1 and hence Theorem 1 also apply to implementations of *modulo* $2n$ *counter*.

5.2 n-valued compare&swap

See the definition and the properties of n-valued compare&swap presented in the appendix.

Let α_j denote the operation sequence *read*, $c\&s(1,j), c\&s(2,j), \ldots, c\&s(n,j)$. Let β denote the operation sequence $\alpha_1^n\alpha_2^n\ldots\alpha_n^n$ (α^m denotes the sequence α repeated m times). Thus, each $|\alpha_j| = n+1$ and $|\beta| = n^2(n+1)$. For all $1 \leq i \leq n-1$, initialize the input variable *op-list*$_i$ to the infinite sequence $\beta\beta\beta\ldots$, and initialize *op-list*$_n$ to the infinite sequence of *read* operations.

In the proof of Claim 1, let p_l be any process in $\{p_1, p_2, \ldots, p_{n-1}\} - \text{PSET}(\Sigma_k)$. If p_l has any pending operation on $\mathcal{O}$ at the end of $\text{EXEC}(C_0, \Lambda_k)$, let $\gamma' \in (\{p_l\} \times \text{COINSPACE})^*$ be such that p_l just completes that operation in $\text{EXEC}(C_0, \Lambda_k\gamma')$. Otherwise let $\gamma' = \epsilon$. Thus, at the end of $\text{EXEC}(C_0, \Lambda_k\gamma')$, p_l has no pending operation on $\mathcal{O}$, but other processes may. Any such pending operations have to be from processes in $\{p_1, \ldots, p_{n-1}\} - \{p_l\}$. Furthermore, since each process appears at most once in Σ_k, if a process has a pending operation in $\text{EXEC}(C_0, \Lambda_k\gamma')$ then that process cannot initiate a new operation on $\mathcal{O}$ in the last $|\Sigma_k|$ steps of $\text{EXEC}(C_0, \Lambda_k\gamma'\Sigma_k)$. Thus, the sum of the number of pending operations on $\mathcal{O}$ in $\text{EXEC}(C_0, \Lambda_k\gamma')$ and the number of operations on $\mathcal{O}$ initiated in the last $|\Sigma_k|$ steps of $\text{EXEC}(C_0, \Lambda_k\gamma'\Sigma_k)$ is at most $n-2$. Let V be the set of all v such that a $c\&s(v,*)$ operation on $\mathcal{O}$ is either pending in $\text{EXEC}(C_0, \Lambda_k\gamma')$ or is initiated in the last $|\Sigma_k|$ steps of $\text{EXEC}(C_0, \Lambda_k\gamma'\Sigma_k)$. From the above, $|V| \leq n-2$. Let $w \in \{1, 2, \ldots, n\}$ be such that $w \notin V$ and $w \neq res$. Let $\gamma'' \in (\{p_l\} \times \text{COINSPACE})^*$ be the shortest schedule such that, in $\text{EXEC}(C_0, \Lambda_k\gamma'\gamma'')$, we have: (i) there are at least $n(n+1)$ completed operations on $\mathcal{O}$ (by p_l) in the last $|\gamma''|$ steps, and (ii) the sequence of $n(n+1)$ most recent operations of p_l on $\mathcal{O}$ is α_w^n. The definition of *op-list*$_l$ and the fact that the implementation is solo-finish imply that γ'' exists. Let $\gamma = \gamma'\gamma''$.

We now make the following observations:

1. In $\text{EXEC}(C_0, \Lambda_k\gamma'\gamma'')$, consider the sequence α_w^n of the $n(n+1)$ most recent operations of p_l on $\mathcal{O}$. This sequence has n^2 c&s operations, each of the form c&s$(*, w)$. By Proposition 3, at least one of these c&s operations succeeds, returning *true*. In the following, we will refer to this successful c&s operation as OP. Since OP is of the form $c\&s(*, w)$, just after the point where OP is linearized, the state of $\mathcal{O}$ is w.

2. In $\text{EXEC}(C_0, \Lambda_k\gamma'\gamma''\Sigma_k)$, the state of $\mathcal{O}$ never changes from w after the point at which OP

is linearized.

From the previous observation, the state of $\mathcal{O}$ is w *just* after the point where OP is linearized. $\mathcal{O}$'s state can change subsequently only if some c&s operation is successful and is linearized after OP. Let OP$'$ be the first c&s operation that is linearized after OP and changes the state of $\mathcal{O}$ from w. There are three cases to consider: (i) OP$'$ is a c&s operation from p_l that follows OP, (ii) OP$'$ is a c&s operation which is pending at the end of EXEC$(C_0, \Lambda_k\gamma')$, or (iii) OP$'$ is a c&s operation which is initiated in the last $|\Sigma_k|$ steps of EXEC$(C_0, \Lambda_k\gamma'\gamma''\Sigma_k)$. In Case (i), by definition of γ'' and OP, OP$'$ is of the form *c&s*$(*, w)$. So if OP$'$ is successful, the resulting state is still w, a contradiction. In Cases (ii) and (iii), by definition of w, OP$'$ = *c&s*$(v, *)$ for some $v \neq w$. Since the state of $\mathcal{O}$ just before linearizing OP$'$ is w, it follows that OP$'$ cannot be successful, a contradiction.

3. By the previous observation, the state of $\mathcal{O}$ at the end of EXEC$(C_0, \Lambda_k\gamma'\gamma''\Sigma_k)$ is w. Therefore, in EXEC$(C_0, \Lambda_k\gamma'\gamma''\Sigma_k\Pi_k\pi)$, p_n's first operation on $\mathcal{O}$ cannot return *res* (since by definition of w, $w \neq$ *res*).

Fact F4 that we needed to prove is the same as the last observation. We conclude that Lemma 1 and hence Theorem 1 also apply to implementations of *n-valued compare&swap*.

A Some types and their properties

The following are the operations and the sequential specifications associated with the types considered in this paper.

- *register* Supports *read* and *write v* operations, where v is any natural number. The states are natural numbers. A *write v* operation changes the state to v and returns *ack*, and a *read* operation returns the state, without affecting it.

- *swap register* Supports *read* and *write v* operations, where v is any natural number. The states are natural numbers. A *write v* operation changes the state to v and returns the previous value of the state. A *read* operation returns the state, without affecting it.

- *resettable consensus* Supports *reset* and *propose v* operations, where v is any natural number. The states are $\{\perp\} \cup \mathcal{N}$, where $\mathcal{N}$ is the set of natural numbers. The *reset* operation changes the state to $\perp$ and returns *ack*. The effect of a *propose v* operation depends on whether or not the state is $\perp$: if the state is $\perp$, *propose v* changes the state to v and returns v; if the state is $w \neq \perp$, *propose v* returns w without affecting the state.

- *increment* Supports *increment* and *read* operations. The states are natural numbers. The *increment* operation adds 1 to the state and returns *ack*. The *read* operation returns the state, without affecting it.

  modulo k counter Supports *increment* and *read* operations. The states are $0, 1, \ldots, k-1$. The *increment* operation adds 1 to the state (modulo k) and returns *ack*. The *read* operation returns the state, without affecting it.

  Proposition 1 *Let E be a finite execution of $(p_1, \ldots, p_n; \mathcal{O})$, where $\mathcal{O}$ is a modulo m counter initialized to 0. Let C be the configuration at the end of E and suppose that process p_n, which has no pending operation (on $\mathcal{O}$ in E), runs alone from C and performs a read operation. In E, if the number of completed increments is at least v and the sum of the number of completed increments and the number of pending increments is at most v', the value returned by the read of p_n is in the range $[v, v'] \bmod m$.*

- *k-valued compare&swap* Supports the operations *read* and *c&s*(u, v) for all $u, v \in \{1, 2, \ldots, k\}$. The states are $1, 2, \ldots, k$. The effect of *c&s*(u, v) depends on whether or not the state is u: if the state is u, *c&s*(u, v) changes the state to v and returns *true*; otherwise it returns *false* without affecting the state.

Here are some observations concerning n-valued compare&swap.

Proposition 2 *Let C be any reachable configuration of $(p_1, \ldots, p_n; \mathcal{O})$, where $\mathcal{O}$ is an n-valued compare&swap object. Suppose that process p_l has no pending operations on $\mathcal{O}$ in C. If p_l runs alone from C, completing the sequence* read, c&s$(1, w)$, c&s$(2, w)$, ..., c&s(n, w) *of operations, then one of the following is true:*

1. *One of the c&s operations of p_l returns true.*
2. *Some operation on $\mathcal{O}$ that was pending in C is linearized after the read and before the last c&s of p_l.*

Proof Let v be the value returned by the read operation by p_l. *c&s*(v, w) is one of the n c&s operations that p_l performs following the read. Clearly, if this does not return true, some pending operation must have taken effect after the read and before the c&s(v, w). □

Proposition 3 *Let C be any reachable configuration of $(p_1, \ldots, p_n; \mathcal{O})$, where $\mathcal{O}$ is an n-valued compare&swap object. Suppose that process p_l has no pending operations on $\mathcal{O}$ in C. Suppose further that p_l runs alone from C, completing on $\mathcal{O}$ the following sequence of operations n times:* read, c&s$(1, w)$, c&s$(2, w)$, ..., c&s(n, w). *Then, at least one of the c&s operations returns true.*

Proof Follows by successive application of Proposition 2 and the observation that there can be at most $n - 1$ pending operations on $\mathcal{O}$ in C. □

References

[AAD+93] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, September 1993.

[Asp90] J. Aspnes. Time and space efficient randomized consensus. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, 1990.

[FHS93] F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. In *Proceedings of the 12th Annual Symposium on Principles of Distributed Computing*, pages 241–249, August 1993.

[HW90] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.

[JTT96] P. Jayanti, K. Tan, and S. Toueg. Time and space lower bounds for non-blocking implementations. Technical report, Department of Computer Science, Cornell University, 1996. Also available by anonymous ftp from `ftp.cs.cornell.edu` in `pub/jayanti/podc96.ps`.