On the Space Complexity of Randomized Synchronization

Faith Fich
Computer Science Department
University of Toronto

Maurice Herlihy
Digital Equipment Corporation
Cambridge Research Lab

Nir Shavit
Computer Science Department
Tel-Aviv University

Abstract

The "wait-free hierarchy" defines a deterministic computability separation among multiprocessor synchronization primitives based on the values of n for which they are able to solve n-process consensus. It has been shown that this separation does not hold in a randomized setting; that is, even *read-write* registers suffice in order to solve n-process consensus for arbitrarily large n.

In this paper, we propose a separation for randomized computation based on the *space complexity* of solutions to n-process consensus. We present the first-ever lower bound for randomized wait-free computation by proving that $\Omega(\sqrt{n})$ *read-write* registers are necessary to solve n-process consensus, even if each register has unbounded size.

We then use this result to relate the randomized complexity of basic multiprocessor synchronization primitives such as *shared counters*, *swap* registers, *fetch&add* registers, and *compare&swap* registers. Viewed collectively, our results imply that there is a hierarchy based on space complexity for synchronization primitives in randomized computation, and that its structure differs from that of the deterministic "wait-free hierarchy."

1 Introduction

Traditionally, the theory of interprocess synchronization has centered around the notion of *mutual exclusion*: ensuring that only one process at a time is allowed to modify complex shared data objects. As a result of the growing realization that unpredictable delay is an increasingly serious problem in modern multiprocessor architectures, a new class of wait-free algorithms have become the focus of both theoretical and experimental research [2, 7]. An implementation of a concurrent object is *wait-free* if:

> it guarantees that every process will complete an operation within a finite number of its own steps, independent of the level of contention and the execution speeds of the other processes.

Wait-free algorithms provide the additional benefit of being highly fault-tolerant, since a process can complete an operation even if all $n-1$ others fail by halting.

The "wait-free hierarchy" [11] defines a deterministic computability separation among concurrent objects and multiprocessor synchronization primitives based on the values of n for which they are able to solve n-process consensus. For example, it is impossible to solve n-process consensus using *read-write* registers and *swap* registers, for $n > 2$. It has been shown [1, 8] that this separation does not hold in a randomized setting; that is, even *read-write* registers suffice to solve n-process consensus. This is a rather fortunate outcome, since it opens the possibility of using randomization to efficiently implement highly concurrent objects without resorting to non-resilient mutual exclusion based algorithms [10]. However, it raises the question of evaluating the randomized computational power of known synchronization primitives.

In this paper, we propose a complexity separation for randomized computation, based on the *space complexity* of solutions to n-process consensus. It is a step towards a theory that will allow designers and programmers of multiprocessor machines to use mathematical tools to recognize when certain protocols are impossible, to evaluate the power of alternative synchronization primitives, and to understand the inherent complexity involved in solving a given problem.

12th ACM Symposium on Principles on Distributed Computing, Ithaca NY

Our main result is a proof that $\Omega(\sqrt{n})$ *read-write* registers are necessary to solve n-process consensus, even if the size of each register is unbounded. To the best of our knowledge, this is the first lower bound for randomized wait-free computation, and among the few known lower bounds for fault-tolerant distributed systems allowing randomization. Unlike the known randomized lower bound of Graham and Yao [9] for byzantine agreement, we derive our bound from the asynchronous nature of the computation, not from the byzantine power of the adversary. Our impossibility proof is based on a new method of "cutting" and "splicing together" *interruptible executions*, ones that can be broken into pieces between which executions involving other processes can be inserted.

We then use this result to relate the randomized complexity of basic multiprocessor synchronization primitives such as *shared counters*, *fetch&add* registers, and *compare&swap* registers. By showing that a single *fetch&add* register, which can deterministically solve 2-process consensus, can solve randomized n-process consensus, we are able to separate it from other deterministic 2-consensus primitives such as *swap*, and equate it with a "stronger" primitive such as *compare&swap*. Together with results regarding *shared counters*, our theorems imply that there is a space complexity based hierarchy for synchronization primitives in randomized computation, and that its structure differs from that of the deterministic "wait-free hierarchy."

2 Model

Aspnes and Herlihy [5] give a formal model for randomized asynchronous algorithms, using a simplified form of the I/O automata formalism of Lynch and Tuttle [15, 14]. Here, we give an informal description, focusing on intuition.

A *concurrent system* consists of a collection of n *processes* that communicate by applying operations to shared *objects*. Each process has an additional operation, called a *coin flip*, that returns a random value in $\{0, 1\}$. Both outcomes are equally likely, and the outcomes of all coin flips are jointly independent. We make no fairness assumptions about processes. A process can halt, or display arbitrary variations in speed. In particular, one process cannot tell whether another has halted or is just running very slowly.

Objects are data structures in memory. Each object has a *type*, which defines a set of possible *values* and a set of primitive *operations* that provide the only means to manipulate that object. Each object has a *sequential specification* that defines how the object behaves when its operations are invoked one at a time by a single process. In a concurrent system, however, an object's operations can be invoked by concurrent processes, and it is necessary to give a meaning to interleaved operation executions. An object is *linearizable* [12] if each operation appears to take effect instantaneously at some point between the operation's invocation and response. In this paper we focus primarily on *read-write registers*, which provide linearizable *read* and *write* operations with the obvious semantics.

Because all objects considered in this paper are linearizable, each process is viewed as executing a sequence of *operations* (both coin flips and data type operations). A *system execution* (or simply an *execution*) is an interleaving of the process executions for all processes. A *system configuration* (or configuration) is given by the state of the memory and the states of the program counters and local variables for each process.

It is convenient to view a randomized algorithm as a game. One side, the processes, tries to achieve agreement against an adversary scheduler. The processes flip coins and apply operations to the shared objects, and the adversary chooses how these operations are interleaved. Our adversary is extremely powerful: it has complete information about the processes' protocols, their internal states, and the state of the shared memory. The adversary cannot, however, predict future coin flips. This notion of adversary is the same as that used by Abrahamson [1], Aspnes and Herlihy [5], Attiya, Dolev, and Shavit [6], and Saks, Shavit, and Woll [16], and is more powerful than that of Chor, Israeli, and Li [8].

An implementation of an object is *randomized non-blocking* if there is always some non-faulty process that completes an operation of this object within a finite *expected* number of its steps when running against the adversary scheduler. It is *randomized wait-free* if every non-faulty process has this property. Randomized wait-free implies non-blocking. The randomized non-blocking property permits the adversary to starve individual processes, but the system as a whole will make progress. The randomized wait-free property excludes starvation; any process that continues to take steps will finish its current operation. For decision problems (such as consensus, discussed below), where each process has only a finite number of operations of the object to perform, the randomized wait-free and non-blocking properties are identical.

A *randomized n-process binary consensus protocol* is a protocol for n asynchronous processes that communicate through a set of shared objects. The processes each start with an input value, either 0 or 1.

Each process communicates with the others by applying operations to shared objects. Eventually, each process decides on an output value and halts. A consensus protocol is required to be:

- *Consistent*: distinct processes never decide on different values.
- *Randomized Wait-free*: each process decides after a bounded *expected* number of steps.
- *Valid*: the common decision value is the input to some process.

The first condition guarantees that the protocol achieves agreement, and the third condition excludes protocols which achieve it trivially by fixing the outcome in advance. For brevity, we use *randomized consensus* to mean *n-process binary consensus*. A set of objects *solves* consensus if there exists a randomized consensus protocol in which processes communicate through those objects.

Theorem 2.1 *Suppose $O(f(n))$ instances of object X solve randomized consensus and $\Omega(g(n))$ instances of object Y are required to solve randomized consensus. Then any randomized non-blocking implementation of X by Y requires $\Omega(g(n)/f(n))$ instances of Y.*

Proof: (*outline*) Suppose there exists a randomized non-blocking implementation of X using $h(n)$ instances of Y. Let $\mathcal{A}$ denote the randomized consensus protocol using $O(f(n))$ instances of X. Construct a new protocol $\mathcal{A}'$ by replacing each instance of X with a randomized non-blocking implementation using $h(n)$ instances of Y. The resulting protocol solves randomized consensus with $O(f(n)h(n))$ instances of Y and therefore $h(n) \in \Omega(g(n)/f(n))$. ■

3 Lower Bounds

A randomized consensus algorithm is required to be consistent on every execution. Therefore, to demonstrate that a randomized consensus algorithm is faulty, it suffices to exhibit an execution in which one process decides the value 0 and another process decides the value 1. Such an execution is specified by an adversary's schedule and a sequence of outcomes for the coin flips that are performed.

Throughout this section, we use the following notation. The number of shared registers is r. If V is a subset of these registers, then $\overline{V}$ denotes the subset of these registers not in V. The sizes of V and $\overline{V}$ are denoted by v and $\overline{v}$, respectively.

A process P is said to be *poised at* register R if P will write to R when next allocated a step. A *block write to a set of registers* V consists of a sequence of v consecutive write operations by v different processes to the v different registers in V. Immediately before a block write to a set of registers V can occur, there must be at least one process poised at each register in V.

We begin by proving a lower bound in the much simpler situation where all processes are identical. In other words, if two processes are in the same state, they perform the same operation on the same register when they are next allocated a step and, if the outcomes of those operations are the same (e.g. the values of their coin flips or the values that they read), their resulting states will be the same as one another. Furthermore, processes with the same input value will be in the same initial state. Although the lower bound proof in this restricted setting is considerably easier than the general case, the overall structure of both are similar and we feel it provides important intuition.

One important idea used in the proof is that, whenever a process writes a value to a register, it is possible to leave behind a group of "clones" all poised to write that value to that register. This is accomplished by giving the process and its clones the same initial state and by having the adversary schedule them as a group. In other words, when one process in the group is allocated a step, each of the other processes in the group is immediately allocated a step. They are also given the same sequence of outcomes for their local coin flips. Then, up to any desired point in the computation, the states of these processes remain the same as one another and they perform exactly the same sequence of operations.

Another essential idea is that, under certain favorable conditions, it is possible to combine an execution that decides 0 with an execution that decides 1 to obtain an execution in which both 0 and 1 are decided. For example, suppose there is a configuration C in which there is a set $\mathcal{P}$ of processes, one poised at each of the r registers. Let C' be the configuration obtained from C as a result of letting each process in $\mathcal{P}$ take one step. Assume that, from C', there is an execution α deciding 0 which contains only steps of processes in $\mathcal{P}$ and, from C, there is an execution β deciding 1 which contains only steps of processes not in $\mathcal{P}$. Then the following is an execution from C that decides both 0 and 1. First perform β. Then let the processes in $\mathcal{P}$ perform a block write. Call the resulting configuration C''. Finally, perform α. See Figure 1.

Note that the configurations C' and C'' are indistin-

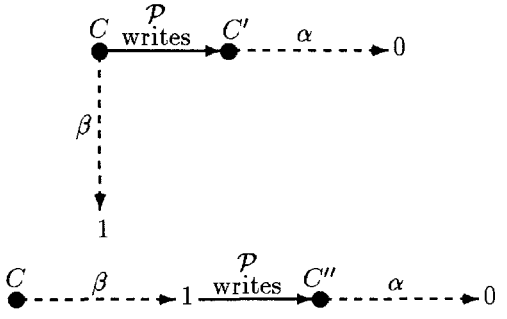

Figure 1: Combining Two Executions

guishable to the processes in $\mathcal{P}$. By writing, they have obliterated all traces of β, so β is invisible to them.

The following lemma shows that this kind of combining can be done under the general conditions illustrated in Figure 2, provided there are sufficiently many processes available. A *solo execution* is a finite execution all of whose steps are taken by the same process.

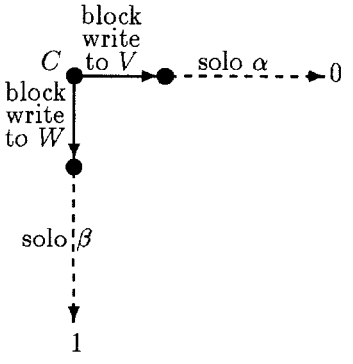

Figure 2: Conditions of Lemma 3.1

Lemma 3.1 *Consider any configuration C in which there is a set of $v \geq 1$ processes $\mathcal{P}$ poised at some set of registers V and a disjoint set of $w \geq 1$ processes $\mathcal{Q}$ poised at some (not necessarily disjoint) set of registers W. Suppose that, after a block write to V by processes in $\mathcal{P}$, there is a solo execution α by a process in $\mathcal{P}$ that decides 0 and, symmetrically, after a block write to W by processes in $\mathcal{Q}$, there is a solo execution β by a process in $\mathcal{Q}$ that decides 1. Then there is an execution from C that decides both 0 and 1 and uses at most $r^2 - r + (3v + 3w - v^2 - w^2)/2$ identical processes.*

Proof: The proof is by induction on $\overline{v} + \overline{w}$.

First, suppose $V \subseteq W$ (which must be the case if $\overline{w} = 0$).

If all writes in α are to registers in W, consider the following execution starting from C. First, processes in $\mathcal{P}$ block write to V, next α is performed, and then processes in $\mathcal{Q}$ block write to W. Note that the resulting configuration is indistinguishable to processes in $\mathcal{Q}$ from the configuration obtained from C by just performing the write to W. Finally, β is performed. This execution decides both 0 and 1 and uses $v + w$ processes. Since $v, w \leq r$, it follows that $v + w \leq r^2 - r + (3v + 3w - v^2 - w^2)/2$.

Otherwise, there is some first point in α at which a register $R \notin W$ is written to. Let C' be the configuration just before this write occurs, let α' denote that part of α occurring after this write, and let $V' = V \cup \{R\}$. Then $v' = v + 1$, because $R \notin W$ and $V \subseteq W$. During the execution from C to C', every register in V is written to at least once. Thus, if there are sufficiently many processes, a clone can be left poised at each register in V, ready to re-write the value it contains at C'. These processes, together with the process poised at R, will form $\mathcal{P}'$. From C', writing to R has the same effect on the registers as performing the block write to V' by $\mathcal{P}'$. Thus, in either case, α' can be performed, deciding the value 0. Furthermore, C and C' are indistinguishable to processes in $\mathcal{Q}$. Therefore, starting at C', if the processes in $\mathcal{Q}$ block write to W and then β is performed, the value 1 is decided. This is illustrated in Figure 3 By the induction hypothesis, there is an execution from C' that decides both 0 and 1 and uses at most $r^2 - r + (3(v+1) + 3w - (v+1)^2 - w^2)/2$ processes. Prepending the execution from C to C' yields an execution from C that decides both 0 and 1 and uses $(v - 1)$ additional processes (those in $\mathcal{P} - \mathcal{P}'$) for a total of at most $r^2 - r + (3v + 3w - v^2 - w^2)/2$.

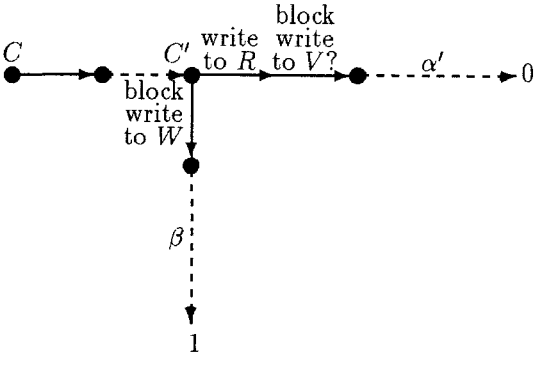

Figure 3: $V \subseteq W$ and α writes to $R \notin W$

Similarly, if $W \subseteq V$, then there is an execution starting from C that decides both 0 and 1. Therefore, suppose that neither V nor W is a subset of the other.

Consider any terminating execution from C that begins with a block write to $U = V \cup W$ and continues with a solo execution γ by one of these u processes. Without loss of generality, suppose γ decides 0. This is illustrated in Figure 4. Let $\mathcal{P}'$ consist of $\mathcal{P}$ plus

a clone of each process in $\mathcal{Q}$ which, at C, is poised at a register in $W - V$. Since $u - 1 \geq v \geq 1$, it follows from the induction hypothesis that there is an execution from C that decides both 0 and 1 and uses at most $r^2 - r + (3u + 3w - u^2 - w^2)/2 \leq r^2 - r + (3v + 3w - v^2 - w^2)/2$ processes. ■

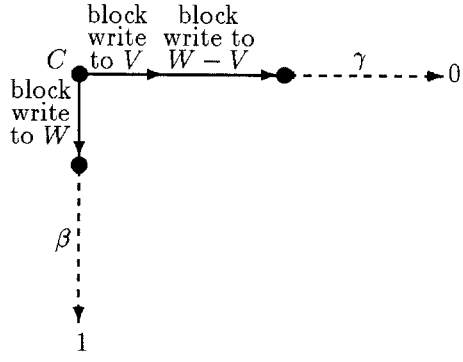

Figure 4: $V \not\subseteq W$, $W \not\subseteq V$, and γ decides 0

Lemma 3.2 *At most $r^2 - r + 1$ identical processes can achieve randomized consensus using r read-write registers.*

Proof: Let P and Q be processes with initial values 0 and 1, respectively. Let α denote any terminating solo execution by P and let β denote any terminating solo execution by Q. By validity, α must decide 0 and β must decide 1.

If one of these executions, say α, contains no writes, then the execution consisting of α followed by β decides both 0 and 1. Therefore, we may assume that both α and β contain at least one write.

Let α' denote the portion of α occurring after the first write and let V be the singleton set consisting of the register P first writes to. Define β' and W analogously. Let γ be the execution consisting of those operations of α and β occurring before their first writes and let C be the configuration obtained from the initial configuration by performing γ. If there are at least $r^2 - r + 2$ processes, it follows by Lemma 3.1 that there is an execution from C that decides both 0 and 1. Prepending this execution by γ yields an execution from the initial configuration that decides both 0 and 1. This violates the consistency condition. ■

Next, we show our main result: $\Omega(\sqrt{n})$ *read-write* registers are necessary in order to solve randomized n-process binary consensus in a wait-free manner. The key to the lower bound is the definition of an interruptible execution. Informally, an interruptible execution is an execution that can be broken into pieces, between which executions involving other processes can be inserted. Each piece of an interruptible execution begins with a block write to a set of registers. Note that executions that do not write to registers outside this set can be inserted immediately before this piece without affecting it.

Definition 3.1 An execution α starting from configuration C is *interruptible* with *initial register set* V and *process set* $\mathcal{P}$ if

- only processes in $\mathcal{P}$ take steps in α.
- at C, there are $\overline{v} + 1$ processes in $\mathcal{P}$ poised at every register in V,
- $\alpha = \alpha'\alpha''$,
- α' begins with a block write to V
- α' contains no writes to registers in $\overline{V}$, and
- if C' is the configuration obtained from C by executing α', then either some process has decided at C' or α'' is an interruptible execution starting from C' with some initial register set $V' \supsetneq V$ and some process set $\mathcal{P}' \subseteq \mathcal{P}$.

In other words, if an execution α is interruptible with initial register set V and process set $\mathcal{P}$, then α can be divided into *pieces* $\alpha = \alpha_1 \cdots \alpha_k$ such that α_i begins with a block write to a set of registers V_i, all writes in α_i are to registers in V_i, and $V = V_1 \subsetneq \cdots \subsetneq V_k$. Furthermore, all the steps of α are taken by processes in $\mathcal{P}$ and, after α has been performed, some process has decided.

Definition 3.2 An interruptible execution α starting from configuration C with initial register set V and process set $\mathcal{P}$ *leaves processes for* a set of registers U if, at C, there are at least u processes not in $\mathcal{P}$ poised at every register in $V \cap U$. Furthermore, if $\alpha = \alpha'\alpha''$ and no process has decided a value by the end of the first piece α', then α'' leaves processes for U.

Provided that a configuration C has the necessary number of processes poised at the specified registers and there are sufficiently many processes available, there is an interruptible execution starting from C.

Lemma 3.3 *Consider any configuration C in which there are at least $\overline{v} + 1$ processes in $\mathcal{P}$ poised at every register in V and at least u processes not in $\mathcal{P}$ poised at every register in $V \cap U$. If $\mathcal{P}$ contains more than $(r^2 - r - v^2 + v)/2 + u \cdot |\overline{V} \cap U|$ processes, then there is an interruptible execution starting from C with initial register set V and process set $\mathcal{P}$ that leaves processes for U.*

Proof: By induction on $\overline{v}$.

Let $\mathcal{Q} \subseteq \mathcal{P}$ be a set of $v\overline{v}$ processes, $\overline{v}$ poised at every register in V. Consider any execution δ starting from configuration C with the following properties:

- only processes in $\mathcal{P} - \mathcal{Q}$ take steps in δ,
- δ begins with a block write to V
- δ contains no writes to registers in $\overline{V}$, and
- at the configuration C' obtained from C by executing δ, either some process in $\mathcal{P} - \mathcal{Q}$ has decided or all processes in $\mathcal{P} - \mathcal{Q}$ are poised at registers in $\overline{V}$.

Such an execution may be obtained by performing a block write to V and then running each process in $\mathcal{P} - \mathcal{Q}$ one at a time, until each is poised at a register in $\overline{V}$ or one of them has decided.

If, at C', some process has decided (which must be the case if $\overline{v} = 0$), then δ is an interruptible execution that satisfies the desired conditions. Therefore assume that, at C', every process in $\mathcal{P} - \mathcal{Q}$ is poised at a register in $\overline{V}$.

For every integer $i \geq 1$, let y_i and z_i be the number of registers in $\overline{V} \cap \overline{U}$ and $\overline{V} \cap U$, respectively, with at least i processes in $\mathcal{P} - \mathcal{Q}$ poised at them. Then $y_i \geq y_{i+1}$ and $z_i \geq z_{i+1}$.

Suppose $y_i + z_{u+i} \leq \overline{v} - i$ for all $1 \leq i \leq \overline{v}$. In particular, $y_{\overline{v}} + z_{u+\overline{v}} \leq 0$, so $y_i = z_{u+i} = 0$ for all $i \geq \overline{v}$. Then

$$\begin{aligned}
|\mathcal{P} - \mathcal{Q}| &= \sum_{i \geq 1} (y_i + z_i) \\
&= \sum_{i=1}^{\overline{v}-1} (y_i + z_{u+i}) + \sum_{i=1}^{u} z_i \\
&\leq \sum_{i=1}^{\overline{v}-1} (\overline{v} - i) + \sum_{i=1}^{u} |\overline{V} \cap U| \\
&= \overline{v}(\overline{v} - 1)/2 + u \cdot |\overline{V} \cap U| \\
&< \overline{v}(\overline{v} - 1)/2 + |\mathcal{P}| - (r^2 - r - v^2 + v)/2 \\
&= |\mathcal{P}| - v\overline{v} = |\mathcal{P} - \mathcal{Q}|.
\end{aligned}$$

This is a contradiction. Therefore, there exists i such that $1 \leq i \leq \overline{v}$ and $y_i + z_{u+i} \geq \overline{v} - i + 1$.

Suppose $Y \subseteq \overline{V} \cap \overline{U}$ and $Z \subseteq \overline{V} \cap U$ are sets of registers such that there are i processes poised at every register in Y, there are $u + i$ processes poised at every register in Z, and $|Y| + |Z| = \overline{v} - i + 1$. Let $V' = V \cup Y \cup Z$. Then $v' = v + \overline{v} - i + 1 = r - i + 1$, so $i = \overline{v'} + 1$. Construct $\mathcal{P}'$ from $\mathcal{P}$ by removing u processes poised at every register in Z. Then, at C', there are at least u processes not in $\mathcal{P}'$ poised at every register in $Z = (V' - V) \cap U$. Since none of the processes outside $\mathcal{P}$ take any steps in δ, there are at least u processes not in $\mathcal{P}$ and, hence, not in $\mathcal{P}'$ poised at every register in $V \cap U$. By definition of Y and Z, at C', there are $i = \overline{v'} + 1$ processes in $\mathcal{P}'$ poised at every register in $Y \cup Z$. Furthermore, there are $\overline{v} \geq \overline{v'} + 1$ processes in $\mathcal{Q} \subseteq \mathcal{P}'$ poised at every register in V.

Finally, since $v < v'$ and

$$\begin{aligned}
|\mathcal{P}'| &= |\mathcal{P}| - u \cdot |Z| \\
&> (r^2 - r - v^2 + v)/2 + u \cdot |\overline{V} \cap U| - u \cdot |Z| \\
&= (r^2 - r - v^2 + v)/2 + u \cdot |\overline{V'} \cap U| \\
&\geq (r^2 - r - (v')^2 + v')/2 + u \cdot |\overline{V'} \cap U|,
\end{aligned}$$

it follows from the induction hypothesis that there is an interruptible execution δ' from C' with initial register set V' and process set $\mathcal{P}'$ that leaves processes for U. In this case, the execution $\delta\delta'$ satisfies the desired conditions. ∎

The next result describes conditions under which two interruptible executions can be combined to form an inconsistent execution. It is analogous to Lemma 3.1.

Lemma 3.4 *Let α and β be two interruptible executions, both starting at configuration C. Suppose α and β have initial register sets V and W, have process sets $\mathcal{P}$ and $\mathcal{Q}$, and decide 0 and 1, respectively. If α leaves processes for $\overline{W}$, β leaves processes for $\overline{V}$, $\mathcal{P}$ and $\mathcal{Q}$ are disjoint, $|\mathcal{P}| > (r^2 - r - v^2 + v)/2 + \overline{w} \cdot |\overline{V} \cap \overline{W}|$, and $|\mathcal{Q}| > (r^2 - r - w^2 + w)/2 + \overline{v} \cdot |\overline{V} \cap \overline{W}|$, then there is an execution starting from C that decides both 0 and 1.*

Proof: By induction on $\overline{v} + \overline{w}$.

First, suppose $V \subseteq W$ (which must be the case if $\overline{w} = 0$). Let α' be the first piece of α and let C' be the configuration obtained from C by executing α'. Since all of the writes in α' are to registers in $V \subseteq W$, β begins with one write to each register in W, and $\mathcal{P}$ and $\mathcal{Q}$ are disjoint, the configurations C and C' are indistinguishable to processes in $\mathcal{Q}$. If some process has decided at C', then $\alpha'\beta$ is an execution from C that decides both 0 and 1. Otherwise, there is an interruptible execution α'' starting from C' with some initial register set $V' \supsetneq V$ and process set $\mathcal{P}' \subseteq \mathcal{P}$ that decides 0 and leaves processes for $\overline{W}$. Without loss of generality, $\mathcal{P}'$ can be all of $\mathcal{P}$ except for $\overline{w} \cdot (|V' \cap \overline{W}| - |V \cap \overline{W}|)$ processes that must be removed to ensure that α'' leaves processes for $\overline{W}$. Then

$$\begin{aligned}
|\mathcal{P}'| &\geq |\mathcal{P}| - \overline{w} \cdot (|V' \cap \overline{W}| - |V \cap \overline{W}|) \\
&> (r^2 - r - v^2 + v)/2 - \overline{w} \cdot |V' \cap \overline{W}| \\
&\geq (r^2 - r - (v')^2 + v')/2 - \overline{w} \cdot |V' \cap \overline{W}|.
\end{aligned}$$

By the induction hypothesis applied to α'' and β, there is an execution δ starting from C' that decides both 0 and 1. Hence, $\alpha'\delta$ is an execution starting from C that decides both 0 and 1.

Similarly, if $W \subseteq V$, then there is an execution starting from C that decides both 0 and 1. Therefore, suppose that neither V nor W is a subset of the other.

Let $V' = W' = V \cup W$. Consider the situation at configuration C. Since α is an interruptible execution with initial register set V, there are $\overline{v}+1 > \overline{v'}+1$ processes in $\mathcal{P}$ poised at each register in V. Since β leaves processes for $\overline{V}$, there are $\overline{v} \geq \overline{v'}+1$ processes not in $\mathcal{Q}$ poised at each register in $V'-V$. Form $\mathcal{P}'$ by adding these processes to $\mathcal{P}$. Note that $\mathcal{P}'$ and $\mathcal{Q}$ are disjoint. Furthermore, since α leaves processes for $\overline{W}$, there are $\overline{w}$ processes not in $\mathcal{P}$ poised at each register in $V \cap \overline{W} = V' \cap \overline{W}$. These processes are not in $\mathcal{P}'$ either, because the processes in $\mathcal{P}'-\mathcal{P}$ are poised at registers in $\overline{V}$. Since $V \subsetneq V'$,

$$\begin{aligned} |\mathcal{P}'| \geq |\mathcal{P}| &> (r^2 - r - v^2 + v)/2 + \overline{w} \cdot |\overline{V} \cap \overline{W}| \\ &\geq (r^2 - r - (v')^2 + v')/2 + \overline{w} \cdot |\overline{V'} \cap \overline{W}|. \end{aligned}$$

By Lemma 3.3, there exists an interruptible execution α' starting from configuration C with initial register set V' and process set $\mathcal{P}'$ that leaves processes for $\overline{W}$. If α' decides 0, then it follows from the induction hypothesis applied to α' and β that there is an execution starting from C that decides both 0 and 1. Therefore we may assume that α' decides 1.

Similarly, we may assume that there exists an interruptible execution β' starting from C with initial register set V' and process set $\mathcal{Q}'$ that leaves processes for $\overline{V}$ and decides 0, where $\mathcal{Q}'$ is constructed analogously to $\mathcal{P}'$.

Since $(V'-V)$ and $(W'-W)$ are disjoint and $\mathcal{P}$ and $\mathcal{Q}$ are disjoint, $\mathcal{P}'$ and $\mathcal{Q}'$ are disjoint. Furthermore, $V' \supsetneq V$ and $W' \supsetneq W$ imply that $|\mathcal{P}'| > (r^2 - r - (v')^2 + v')/2 + \overline{w'} \cdot |\overline{V'} \cap \overline{W'}|$, and $|\mathcal{Q}'| > (r^2 - r - (w')^2 + w')/2 + \overline{v'} \cdot |\overline{V'} \cap \overline{W'}|$. It then follows from the induction hypothesis applied to β' and α' that there is an execution starting from C that decides both 0 and 1. ■

Lemma 3.5 *At most $3r^2-r+1$ processes can achieve randomized consensus using r read-write registers.*

Proof: Consider any (randomized) algorithm that purports to achieve wait-free binary consensus among $3r^2-r+2$ processes using r *read-write* registers. Partition these processes into two sets, $\mathcal{P}$ and $\mathcal{Q}$, each containing $(3r^2 - r + 2)/2$ processes. Give each process in $\mathcal{P}$ the initial value 0 and give each process in $\mathcal{Q}$ the initial value 1.

```
RMW(R: register, f: function) returns(value);
    previous := R;
    R := f(R);
    return (previous);
    end RMW
```

Figure 5: The *read-modify-write* operation

Let $V = W = \varnothing$. By Lemma 3.3, there is an interruptible execution α starting from the initial configuration with initial register set V and process set $\mathcal{P}$ that leaves processes for $\overline{W}$. Since the processes in $\mathcal{P}$ all have initial value 0, α must decide 0. Similarly, there is an interruptible execution β starting from the initial configuration with initial register set W and process set $\mathcal{Q}$ that leaves processes for $\overline{V}$ and decides 1. Hence, Lemma 3.4 implies that there is an execution starting from the initial configuration that decides both 0 and 1, violating the consistency condition. ■

The following result is a direct consequence of Lemma 3.5.

Theorem 3.6 *A randomized wait-free implementation of consensus from read-write registers requires $\Omega(\sqrt{n})$ instances of these registers.*

By slightly modifying the definition of interruptible execution, it is possible to extend Theorem 3.6 to registers that allow swaps as well as reads and writes.

Theorem 3.7 *A randomized wait-free implementation of consensus from read-write and swap registers requires $\Omega(\sqrt{n})$ instances of these registers.*

4 Separation Results

We use our main theorem to derive a series of results relating the proposed *randomized wait-free hierarchy* with the deterministic hierarchy of [11]. For brevity, we say that object X is *more powerful* than object Y if n-process randomized consensus requires asymptotically fewer instances of X than Y.

Define a *read-modify-write* operation (Figure 5) to be *non-trivial* if f is not the identity function.

Definition 4.1 Let F be a set of functions indexed by an arbitrary set S. Define F to be *interfering* if for all values v and all i and j in S, either:

1. The functions f_i and f_j commute: $f_i(f_j(v)) = f_j(f_i(v))$.

```
SWAP(R : register, x : value) returns(value);
    previous := R;
    R := x;
    return (previous);
    end SWAP
```

Figure 6: The *swap* operation

2. One function "overwrites" the other: $f_i(f_j(v)) = f_i(v)$ or $f_j(f_i(v)) = f_j(v)$.

Examples of non-trivial and interfering read-modify-write operations include the *swap* operation of Figure 6 and the *fetch&add* operation of Figure 7. The *compare&swap* operation of Figure 8 is non-trivial but not interfering. Theorems 3 and 4 of [11] show that registers with non-trivial and interfering *read-modify-write* operations solve 2-process consensus and, therefore, form a class of objects that are deterministically more powerful than *read-write* registers, which cannot solve 2-process consensus, and less powerful than operations such as *compare&swap*, which can solve n-process consensus.

We first show that, in randomized computation, as in the deterministic case, *compare&swap* is still more powerful than *read-write* and *swap* operations. The *swap* and *fetch&add* operations, however, are not in the same randomized class, even though they are in the same (non-trivial and interfering) deterministic class. Moreover, *swap* is weaker than a *bounded counter*, an object with a deterministic implementation from *read-write* registers, while *fetch&add* is as strong as *compare&swap*.

Herlihy [11, Theorem 5] shows that binary consensus can be implemented deterministically using a single bounded *compare&swap* register. Hence, from Theorem 3.7, we have:

Corollary 4.1 *Any randomized non-blocking bounded compare&swap register implementation using read-write and swap registers requires $\Omega(\sqrt{n})$ instances of these registers.*

A *bounded counter* object has an integer state that can assume a range of values, and it provides the following operations: *inc* and *dec* respectively increment

```
F&A(R: register, x: value) returns(value);
    previous := R;
    R := R + x;
    return (previous);
    end F&A
```

Figure 7: The *fetch&add* operation

```
COMPARE&SWAP(R: register, old: value, new: value)
    returns(value);
    previous := R;
    if previous = old then R := new;
      end if
    return (previous);
    end COMPARE&SWAP
```

Figure 8: The *compare&swap* operation

and decrement the counter (without returning any information), *write* initializes the counter to a fixed state, and *read* returns the counter's current value. Aspnes [3] gives a randomized binary consensus protocol in which n processes share three bounded counters: the first two keep track of the number of processes with preference 0 and 1 respectively, and the third is used as the cursor for a random walk. The first two counters assume values between 0 and n, while the third assumes values between $-3n$ and $3n$. (The first two counters can be eliminated at some cost in performance [4].)

Theorem 4.2 (Aspnes) *There is a deterministic bounded shared counter implementation using $O(n)$ read-write registers, and a randomized consensus implementation using one bounded shared counter.*

From Theorems 2.1, 3.7, and 4.2 we have:

Corollary 4.3 *Any randomized non-blocking bounded counter implementation using read-write and swap registers requires $\Omega(\sqrt{n})$ instances of these registers.*

Surprisingly, both lower bounds are independent of the number of values any object can assume: they hold even when the *read-write* and *swap* registers used in the implementation are unbounded, but the counter being implemented is bounded.

Similar results hold for *fetch&increment*, *fetch&decrement*, and *fetch&add*, a constant number of which suffice to implement a counter. In the full paper we show how proper encoding can allow a single instance of a *fetch&add* operation to implement three instances of a shared counter, and so we have that *fetch&add* and *compare&swap*, which are in separate classes in the deterministic hierarchy, have similar randomized power in the sense that one instance of each suffices to solve randomized consensus.

Theorem 4.4 *Randomized consensus can be solved using a single instance of a fetch&add register.*

In addition, by Theorems 2.1 and 3.7 we have:

Corollary 4.5 *Any randomized non-blocking implementation of a fetch&add register using read-write and swap registers requires $\Omega(\sqrt{n})$ instances of these registers.*

Acknowledgments

Part of this work was performed while Faith Fich and Nir Shavit were visiting MIT. Their work was supported by the Natural Sciences and Engineering Research Council of Canada, the Information Technology Research Centre of Ontario, ONR grant N00014-91-J-1046, NSF grant 8915206-CCR, and DARPA grants N00014-92-J-4033 and N00014-91-J-1698.

References

[1] K. Abrahamson, "On Achieving Consensus Using a Shared Memory," *Proc. 7th Annual ACM Symposium on Principles of Distributed Computing*, 1988, pp. 291–302.

[2] J. Alemany and E. Felten, "Performance Issues in Non-Blocking Synchronization on Shared-memory Multiprocessors," *Proc. 11th Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, Canada, August 1992, pp. 125–134.

[3] J. Aspnes, "Time-and-Space Efficient Randomized Consensus," *Proc. 9th Annual ACM Symposium on Principles of Distributed Computing*, Quebec City, Canada, August 1990, pp. 325–331.

[4] J. Aspnes, private communication.

[5] J. Aspnes and M. Herlihy, "Fast Randomized consensus Using Shared Memory," *Journal of Algorithms*, vol. 11, September 1990, pp. 441–461.

[6] H. Attiya, D. Dolev and N. Shavit, "Bounded Polynomial Randomized Consensus," *Proc. 8th Annual ACM Symposium on Principles of Distributed Computing,* Edmonton, Canada, August 1989, pp. 281–293.

[7] B. Bershad, "Practical Considerations for Lock-free Concurrent Objects," *Proc. Symposium on Architectural Support for Programming Languages and Systems*, Boston, November, 1992.

[8] B. Chor, A. Israeli, and M. Li, "On Processor Coordination Using Asynchronous Hardware", *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 1987, pp. 86–97.

[9] R. Graham and A. Yao, "On the Improbability of Reaching Byzantine Agreements," In *Proc. 21st ACM Annual Symposium on the Theory of Computing*, Seattle, May 1989, pp. 467–478.

[10] M.P. Herlihy, "Randomized Wait-free Concurrent Objects," *Proc. 10th Annual ACM Symposium on Principles of Distributed Computing*, Montreal, Canada, August 1991, pp. 11–21.

[11] M. P. Herlihy, "Wait-Free Synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 1, January 1991, pp. 124–129.

[12] M.P. Herlihy and J.M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, July 1990, pp. 463–492.

[13] L. Lamport, "On Interprocess Communication. Part II: Algorithms," *Distributed Computing*, vol. 1, no. 2, 1986, pp. 86–101.

[14] N.A. Lynch and M.R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. *Proc. 6th Annual ACM Symposium on Principles of Distributed Computing*, August 1987, pp. 137–151. Full version available as MIT Technical Report MIT/LCS/TR–387.

[15] N.A. Lynch and M.R. Tuttle. "An Introduction to Input/Output Automata", MIT Technical Report MIT/LCS/TR–373, November 1988.

[16] M. Saks, N. Shavit and H. Woll, "Optimal Time Randomized Consensus – Making Resilient Algorithms Fast in Practice," *Proc. 2nd Annual ACM Symposium on Discrete Algorithms*, January 1991, pp. 351–362.