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

September 24, 1997

Abstract

The "wait-free hierarchy" provides a classification of multiprocessor synchronization primitives based on the values of n for which there are deterministic wait-free implementations of n-process consensus using instances of these objects and *read-write* registers. In a randomized wait-free setting, this classification is degenerate, since n-process consensus can be solved using only $O(n)$ *read-write* registers.

In this paper, we propose a classification of synchronization primitives based on the *space complexity* of randomized solutions to n-process consensus. A *historyless* object, such as a *read-write* register, a *swap* register, or a *test&set* register, is an object whose state depends only on the last nontrivial operation that was applied to it. We show that, using *historyless* objects, $\Omega(\sqrt{n})$ object instances are necessary to solve n-process consensus. This lower bound holds even if the objects have unbounded size and the termination requirement is *non-deterministic solo termination*, a property strictly weaker than randomized wait-freedom.

We then use this result to relate the randomized space complexity of basic multiprocessor synchronization primitives such as *shared counters*, *fetch&add* registers, and *compare&swap* registers. Viewed collectively, our results imply that there is a separation based on space complexity for synchronization primitives in randomized computation, and that this separation differs from that implied by 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 [27] and experimental research [5, 12]. 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 consensus problem is a computational task which requires each of n asynchronous processes, with possibly different private input values, to decide on one of the inputs as their common output. The "wait-free hierarchy" [20] classifies concurrent objects and multiprocessor synchronization primitives based on the values of n for which they solve n-process consensus in a wait-free manner. For example, it is impossible to solve n-process consensus using *read-write* registers for $n > 1$ or using *read-write* registers and *swap* registers, for $n > 2$ [2, 15, 16, 20, 26]. It has been shown that this separation does not hold in a randomized setting; that is, even *read-write* registers suffice to solve n-process consensus [2, 15]. This is a rather fortunate outcome, since it opens the possibility of using randomization to implement concurrent objects without resorting to non-resilient mutual exclusion [7, 9, 10, 19, 32]. One important application is the software implementation of one synchronization object from another. This allows easy porting of concurrent algorithms among machines with different hardware synchronization support. However, in order to understand the conditions under which such implementations are effective and useful, we must be able to quantify the randomized computational power of existing hardware synchronization primitives.

In this paper, we propose such a quantification by providing a complexity separation among synchronization primitives based on the *space complexity* of randomized wait-free solutions to n-process binary consensus. It is a step towards a theory that would allow designers and programmers of multiprocessor machines to use mathematical tools to evaluate the power of alternative synchronization primitives and to recognize when certain randomized protocols are inherently inefficient.

Randomized n-process consensus can be solved using $O(n)$ read-write registers of bounded size [9]. Our main result is a proof that using only *historyless* objects (for example, *read-write* registers of unbounded size, *swap* registers, and *test&set* registers), $\Omega(\sqrt{n})$ instances are necessary to solve randomized n-process binary consensus. We do so by showing a more general result: there is no implementation of consensus satisfying a special property, *nondeterministic solo termination*, from a sufficiently small number of historyless objects. The nondeterministic solo termination property is strictly weaker than randomized wait-freedom.

Our result thus shows that, for n-process consensus, the number of object instances, rather than the sizes of their state spaces, is the important factor. Furthermore, allowing operations such as SWAP or TEST&SET, in addition to READ and WRITE, does not substantially reduce the number of objects necessary to solve consensus.

A variety of lower bounds for randomized algorithms exist in the literature. A lower bound for the Byzantine Agreement problem in a randomized setting was presented by Graham and Yao[18]. They showed a lower bound on the probability of achieving agreement among 3 processes, one of which might behave maliciously. Their result is derived based on the possibly malicious behavior of a processor.

A randomized space complexity lower bound was presented by Kushilevitz, Mansour, Rabin, and Zuckerman [24]. They prove lower bounds on the size of (i.e. number of bits in) a read-modify-write register necessary to provide a fair randomized solution to the mutual exclusion problem. Their results relate the size of the register to the amount of fairness in the system and the number of processes accessing the critical section in a mutually exclusive manner. A deterministic space lower bound on the number of bits in a compare&swap register necessary to solve n-process consensus was proved by Afek and Stupp [1].

Lower bounds on the space complexity of consensus have also been obtained for models in which *objects* as well as processes may fail [4, 23].

A powerful time complexity lower bound was recently presented by Aspnes [6] for the randomized consensus problem when up to t processes may fail by halting. Aspnes studies the total number of coin flips necessary to create a global shared coin, a mathematical structure that he shows must implicitly be found in any protocol solving consensus, unless its complexity is $\Omega(t^2)$. His bound on the shared coin construction implies an interesting $\Omega(t^2/\log^2 t)$ lower bound on the amount of work (total number of operations by all processes) necessary to solve randomized consensus.

Our proof technique is most closely related to the elegant method introduced by Burns and Lynch to prove a lower bound on the number of read/write registers required for a deterministic solution to the mutual-exclusion problem [14]. Though related, the problem they tackle is fundamentally different and in a sense easier than ours. This is because the object they consider (mutual exclusion) is accessed by each process repeatedly, whereas our lower bound applies to the implementation of general objects and, in particular, consensus (for which each process may perform only a single access).

Because the objects we consider are "single access," our lower bound proofs required the development of a new proof technique. The key element of this technique is a method of "cutting" and "splicing together" *interruptible executions*, executions that can be broken into pieces between which executions involving other processes can be inserted.

Based on our consensus lower bound, we are able to relate the randomized complexity of basic multiprocessor synchronization primitives such as *counters* [9, 30], *fetch&add* registers, and *compare&swap* registers. For example, *swap* registers and *fetch&add* registers are each sufficient to deterministically solve 2-process consensus, but not 3-process consensus. However, a single *fetch&add* register suffices to solve randomized n-process consensus, whereas $\Omega(\sqrt{n})$ *swap* registers are needed. Furthermore, a primitive such as *compare&swap*, which is deterministically "stronger" than *fetch&add*, is essentially equivalent to it under this randomized complexity measure. Our theorems imply that there is a space-complexity-based separation among synchronization primitives in randomized computation and that their relative power differs from what could be expected given the deterministic "wait-free hierarchy." Our hope is that such separation results will eventually allow hardware designers and programmers to make more informed decisions about which synchronization primitives to use in a randomized setting and how to better emulate one primitive from another.

The structure of our presentation is as follows. In Section 2, we describe our model of computation and how randomization and asynchrony are expressed within it. Section 3 begins by proving a special case of the lower bound, followed by the proof of the general case. Section 4 presents several separation theorems among synchronization primitives based on our main lower bound.

2 Model

Our model of computation consists of a collection of n sequential threads of control called *processes* that communicate by applying operations to shared *objects*.

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. The current value of an object and the operation that is applied to it determine (perhaps nondeterministically) the response to the operation and the (possibly) new value of the object.

For example, a *test&set* register has $\{0, 1\}$ as its set of possible values. Its initial value is 0. The TEST&SET operation responds with the value of the object and then sets the value to 1. A *read-write* register may have any (finite or infinite) set as its set of values. Its operations are READ, which responds with the value of the object, and WRITE(X), for x in the value set, which sets the value of the object to x.

Another example is the *counter* [9, 30]. The integers are its set of values. Its operations are INC and DEC, which increment and decrement the counter, respectively, RESET, which sets the value of the counter to 0, and READ, which responds with the value of the counter, leaving it unchanged. The first three of these operations only respond with a fixed acknowledgement. A *bounded counter* is a counter whose set of possible values is a range of integers and whose operations are performed modulo the size of that range.

Each process has a set of *states*. The operation a process applies and the object to which it is applied depends on the current state of the process. Processes may also have internal operations, such as coin flips. The current state of a process and the result of the operation performed determine the next state of the process. Each such operation is called a *step* of the process. Processes are asynchronous, that is, they can halt or display arbitrary variations in speed. In particular, one process cannot tell whether another has halted or is just running very slowly.

An *execution* is an interleaving of the sequence of steps performed by each process. All objects are *linearizable* or *atomic* in the sense that the processes obtain results from their operations on an object as if those operations were performed sequentially in the order specified by the execution. The *configuration* at any point in an execution is given by the state of all processes and the value of all objects. A process may become faulty at a given point in an execution, in which case it performs no subsequent operations.

A more formal description of our model can be given in terms of I/O automata [27, 28, 29, 31]. The randomized asynchronous computation model, which includes coin flips, is described in [9]. A formal definition of linearizability can be found in [21].

An operation of an object type is said to be *trivial* if applying the operation to any object of the type always leaves the value of the object unchanged. The READ operation is an example of a trivial operation. Two operations on an object type are said to *commute* if the order in which those operations are applied to any object of the type does not affect the resulting value of the object. For example, DECREMENT and FETCH&ADD operations commute with themselves and one another. A trivial operation commutes with any other operation on the same object.

An operation f on an object *overwrites* an operation f' if, starting from any value, performing f' and then f results in the same value (or set of values) as performing just f (i.e. $f(x) = f(f'(x))$ for all possible values x). This implies that, from every configuration, every sequence of operations on that object yields the same sequence of responses when preceded by f as when preceded by f' and f. If the value transition relation associated with an operation is idempotent then the operation overwrites itself. All WRITE, TEST&SET, and SWAP operations on an object overwrite one another.

An object type is *historyless* if all its nontrivial operations overwrite one another. In other words, the value of a historyless object depends only on the last nontrivial operation that was applied to it.

A set of operations on an object is *interfering* if every pair of these operations either commute or overwrite one another. For example, the set of READ, WRITE, and SWAP operations is interfering, but the set of COMPARE&SWAP operations is not.

An *implementation* of an object X is a set of objects $Y_1, \ldots, Y_m$ representing X together with procedures $F_1, \ldots, F_n$ called by processes $P_1, \ldots, P_n$ to execute operations on X. For object types X and Y, we say that X can be implemented from m instances of Y if there exists an implementation of an object of type X by processes $P_1, \ldots, P_n$ using objects $Y_1, \ldots, Y_m$ of type Y.

An implementation is *wait-free* if *each* nonfaulty process P_i always finishes executing its procedure F_i within a fixed, finite number of its own steps, regardless of the pace or failure of other processes. An implementation is *non-blocking* if, for every configuration and every execution starting at that configuration, there is *some* process P_i that finishes executing its procedure F_i within a finite number of steps. Wait-free implies non-blocking. The non-blocking property permits individual processes to starve, but requires the system as a whole to make progress. The wait-free property excludes starvation: any process that continues to execute events will finish its current operation. If each procedure F_i can be called only a finite number of times, then wait-free is the same as non-blocking.

The wait-free and non-blocking properties can be extended to *randomized wait-free* and *randomized non-blocking* by only requiring that P_i finishes executing its procedure F_i within a finite *expected* number of its steps. (See [9] for a more formal definition.) If each procedure F_i can be called only a finite number of times, then randomized wait-free is the same as randomized non-blocking. Furthermore, if an algorithm is wait-free, then it is randomized wait-free and if it is non-blocking, then it is randomized non-blocking.

A *solo execution* is an execution all of whose steps are performed by one process. An implementation has the *nondeterministic solo termination property* if, for every configuration C and every process P_i, there exists a finite solo execution, starting at configuration C, in which P_i finishes executing its procedure F_i. In other words, if P_i has no interference, it will finish performing its operation. If an algorithm is randomized wait-free (or wait-free), then it has the nondeterministic solo termination property, since every state transition having non-zero probability can be viewed as a possible nondeterministic choice.

Nondeterministic solo termination is a strictly weaker property than wait-freedom and randomized wait-freedom. For example, the simple snapshot algorithm following Observation 1 in [3] is not (randomized wait-free, but satisfies the nondeterministic solo termination property.

We evaluate the randomized space complexity of an object type based on the number of objects of the type that are required to implement n-process *binary consensus* in a randomized wait-free manner. A *binary consensus object* is an object on which each of n-processes can perform one DECIDE operation with input value in $\{0, 1\}$, returning an output value x, also in $\{0, 1\}$. The object's legal executions are those that satisfy the following conditions:

Consistency: The DECIDE operations of all processes return the same value.

Validity: If x is the value returned for some DECIDE operation, then x is the input value for the DECIDE operation of some process.

The first condition guarantees that consensus is achieved and the second condition excludes trivial solutions in which the outcome is fixed ahead of time.

We will use the term *consensus* to mean *n-process binary consensus*. A set of objects is said to solve *randomized consensus* if there is a randomized wait-free implementation of consensus from only that set of objects. No executions of an implementation may give an incorrect answer (i.e. one that violates consistency or validity). In other words, we do not consider Monte Carlo implementations.

An execution of an implementation of consensus is *terminating* if it completes a DECIDE operation. Arbitrarily long and even non-terminating executions are possible in a randomized wait-free implementation. For example, since it is impossible to implement consensus in a wait-free manner for two or more processes from only read-write registers, any randomized wait-free implementation of consensus for two or more processes from only read-write registers must have non-terminating executions. However, these executions must occur with correspondingly small probabilities.

Consider the situation where there is a lower bound on the number of objects to solve randomized consensus, for objects of a particular type. Then this bound can be used to obtain lower bounds on the number of instances of that object type that are necessary to implement objects of other types.

Theorem 2.1 *Let X and Y be object types. Suppose $f(n)$ instances of X solve n-process randomized consensus and $g(n)$ instances of Y are required to solve n-process randomized consensus. Then any randomized non-blocking implementation (and, hence, any randomized wait-free implementation) of X by Y for n processes requires $g(n)/f(n)$ instances of Y.*

Proof: Suppose there exists a randomized non-blocking implementation of X using $h(n)$ instances of Y. Let $\mathcal{A}$ denote a randomized wait-free implementation of consensus using $f(n)$ instances of X. Construct a new randomized wait-free implementation of consensus by replacing each instance of X in $\mathcal{A}$ with a randomized non-blocking implementation using $h(n)$ instances of Y. In total, this implementation uses $f(n)h(n)$ instances of Y. Therefore $f(n)h(n) \geq g(n)$ so $h(n) \geq g(n)/f(n)$. ■

3 Lower Bounds

In this section, we prove an $\Omega(\sqrt{n})$ lower bound on the number of objects required to solve randomized consensus, if each object is historyless. More specifically, we do this by showing that there is no implementation of consensus satisfying nondeterministic solo termination from only a sufficiently small number of historyless objects. Although it is weaker than randomized wait-freedom, the nondeterministic solo termination property is sufficient for proving our lower bound.

An implementation of consensus is required to be consistent in every execution. Therefore, to demonstrate that an implementation of consensus is faulty, it suffices to exhibit an execution in which one process decides the value 0 and another process decides the value 1. This is done by combining an execution that decides 0 with an execution that decides 1. Although our lower bound is stated in terms of a strong model that requires objects to be linearizable, it also applies to all weaker models, such as those which guarantee only sequential consistency.

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

In any configuration of the implementation, a process P is said to be *poised at* object R if P will perform a non-trivial (historyless) operation on R when next allocated a step. In this case, the value of object R can be fixed at any single future point in time by scheduling P to perform that operation. A *block write to a set of objects* V consists of a sequence of v consecutive non-trivial (historyless) operations by v different processes on the v different objects in V. Immediately before a block write to a set of objects V can occur, there must be at least one process poised at each object in V. Using a block write to V, the values of all the objects in V can be fixed. Knowing which processes are poised at each object in a particular configuration enables us to determine which process to schedule next.

Under certain favourable conditions, it is possible to combine an execution that decides 0 with an execution that decides 1. 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 objects. Let C'

be the configuration obtained from C by having the processes in $\mathcal{P}$ perform a block write. Suppose that, from C', there is an execution α deciding 0 by a set of processes $\mathcal{Q}$, none of which are in $\mathcal{P}$. Furthermore, suppose that, from C, there is an execution β deciding 1 that contains no steps of processes in either $\mathcal{P}$ or $\mathcal{Q}$. 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. This is illustrated in Figure 1.

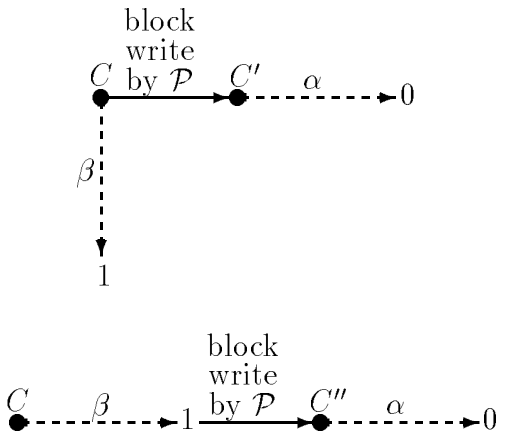


Figure 1: Combining Two Executions

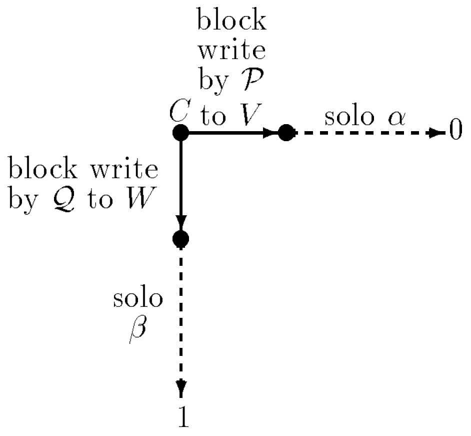
Note that the configurations C' and C'' are indistinguishable to the processes in $\mathcal{Q}$. By writing, the processes in $\mathcal{P}$ have obliterated all traces of β from the objects, so β is invisible to the processes in $\mathcal{Q}$.

3.1 Read Write Registers and Identical Processes

First, we prove a lower bound in a much simpler situation: the objects are *read-write* registers (i.e. the only operations are READ and WRITE) and all processes are identical. Thus, 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. 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 in the general case, the overall structure of both proofs are similar and we feel the easier proof provides important intuition.

One technique that can only be applied in the restricted setting is *cloning*. Consider any point in some execution at which a process P writes a value v to a register R. Then there is another execution which is the same except that a group of *clones* have been left behind, all poised to write value v to register R. To construct this execution, the clones are given the same initial state as P and P and its clones are scheduled as a group, up to the point at which P performs the write. In other words, whenever P is allocated a step, each of its clones is immediately allocated a step. The outcomes of the clones' internal operations in this execution are specified to be the same as the corresponding outcomes of P's internal operations. Then, up to the point at which P performs the write, the clones have the same state as P and they perform exactly the same sequence of operations. These clones can be scheduled to perform their writes at various subsequent points of time, resetting the contents of the register R to the same value each time.

Provided there are sufficiently many processes available, cloning enables two executions to be combined into an inconsistent execution under the general conditions illustrated in Figure 2.



block write by $\mathcal{P}$ to V
C
solo α
0
block write by $\mathcal{Q}$ to W
solo β
1

Figure 2: Conditions of Lemma 3.1

Lemma 3.1 *Consider any implementation of consensus from r read-write registers using identical processes that satisfies nondeterministic solo termination. Let C be a configuration 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 $\mathcal{P}$ perform a block write to V, next α is performed, and then processes in $\mathcal{Q}$ perform a 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 block 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 the solo execution 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 $V' = V \cup \{R\}$. Note that $R \notin V$, because $R \notin W$ and $V \subseteq W$. Hence $v' = v + 1$. 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-perform the last write that was performed on the register prior to C'. These v clones, together with the process performing the solo execution α (which includes the write to R and the solo execution α'), will form $\mathcal{P}'$. Note that the value of every register in V' is the same in the configuration immediately after the write to R and the configuration immediately after the block write to V' by $\mathcal{P}'$. Thus, starting at either of these two configurations, α' decides the value 0. Furthermore, since $V \subseteq W$, the configurations C and C' are indistinguishable to processes in $\mathcal{Q}$. Therefore, starting at C', if the processes in $\mathcal{Q}$ perform a 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 $r^2 - r + (3v + 3w - v^2 - w^2)/2$.

By symmetry, 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. Such an execution exists by the nondeterministic solo termination property. Since $U \subsetneq V$ and $V \neq \varnothing$, it follows that $u - 1 \geq v \geq 1$. Suppose γ decides 0. (The case when γ decides 1 is symmetric.) Let $\mathcal{P}'$ consist of $\mathcal{P}$ plus a clone of each process in $\mathcal{Q}$ that is poised at a register in $W - V$ in configuration C. This is illustrated in Figure 4. By the induction hypothesis applied to configuration C, sets of registers U and W, and the sets of processes $\mathcal{P}'$ and $\mathcal{Q}$, there is an execution from C which 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.

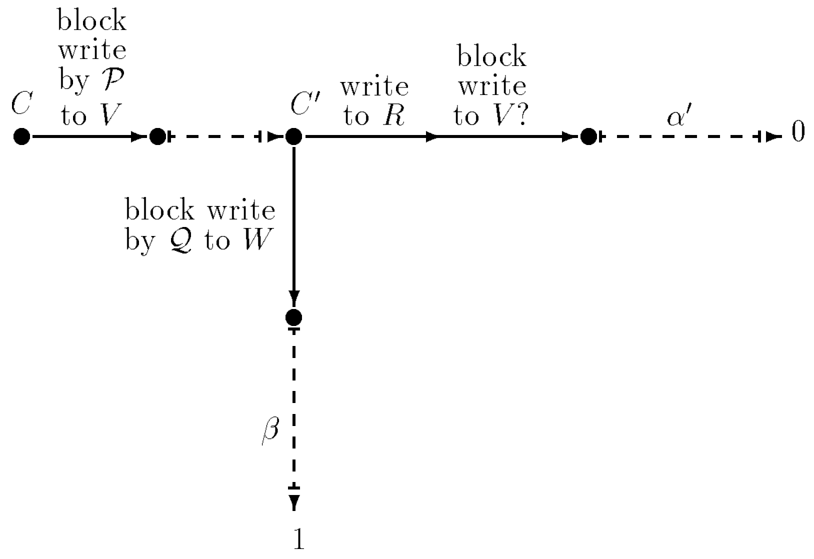

block write by $\mathcal{P}$ to V
C
C'
write to R
block write to V?
α'
0
block write by $\mathcal{Q}$ to W
β
1

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

■

To obtain our lower bound, we show that, for any randomized consensus algorithm, the conditions necessary to apply Lemma 3.1 can be achieved, provided sufficiently many processes are available.

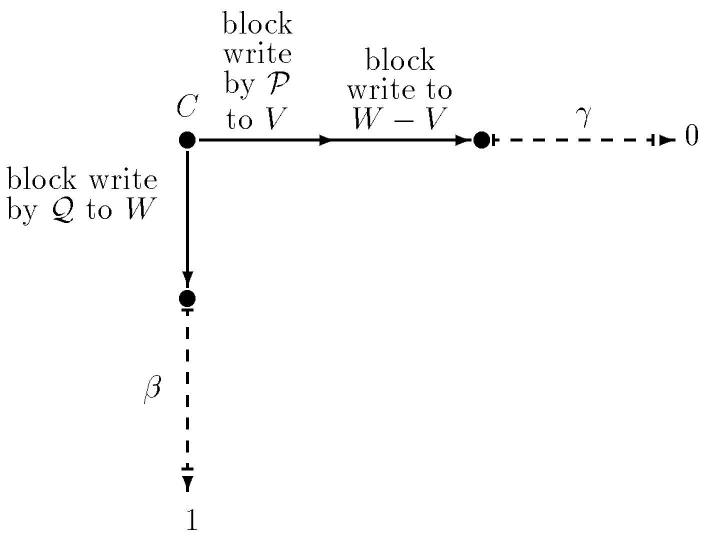
Lemma 3.2 *There is no implementation of consensus satisfying nondeterministic solo termination from r read-write registers using $r^2 - r + 2$ or more identical processes.*

Proof: Consider any implementation of consensus satisfying nondeterministic solo termination from r read-write registers using $r^2 - r + 2$ or more identical processes.

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. The existence of these executions is guaranteed by the nondeterministic solo termination property. Furthermore, 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



block write by $\mathcal{P}$ to V
block write to $W - V$
C
γ
0
block write by $\mathcal{Q}$ to W
β
1

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

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. ■

Theorem 3.3 *At most $r^2 - r + 1$ identical processes can solve randomized consensus using r read-write registers.*

3.2 General Case

Next, we show our main result: $\Omega(\sqrt{n})$ objects are necessary to implement n-process binary consensus in a randomized wait-free manner, if the objects are historyless. 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 objects. Because the objects are historyless, this block write gives these objects particular values, no matter when it is executed. Hence, if an execution performed by other processes only changes the values of objects in the set, then that execution can be inserted immediately before this piece without affecting the rest of the interruptible execution.

Note that, unlike the special case considered in Section 3.1, the resulting state of each process performing the block write may depend on the value of the object it accesses. This could influence a subsequent execution involving this process. Therefore, in the definition of an interruptible execution, processes that participate in a block write take no further steps.

Definition 3.1 *An execution α starting from configuration C is* interruptible *with* initial object set V *and* process set $\mathcal{P}$ *if all steps in α are taken by processes in $\mathcal{P}$ and α can be divided into one or more* pieces $\alpha = \alpha_1 \cdots \alpha_k$ *such that*

- *α_i begins with a block write to a set of objects V_i by processes that take no further steps in α,*
- *all non-trivial operations in α_i are to objects in V_i,*
- *$V = V_1 \subsetneq \cdots \subsetneq V_k$, and*
- *after α has been performed, some process has decided.*

In other words, if an execution α is interruptible with initial object set V then $\alpha = \alpha_1\alpha'$, where α_1 begins with a block write to V by processes that take no further steps in α and α_1 contains no non-trivial operations on objects in $\overline{V}$. Furthermore, if C' is the configuration obtained from C by executing α_1, then either some process has decided at C' and α' is empty or α' is an interruptible execution starting from C' with some initial object set $V' \supsetneq V$.

We say that an interruptible execution *decides* a value if, after α has been performed, some process has decided (i.e. has returned) that value.

A terminating solo execution augmented with a sufficient numbers of clones performing block writes at appropriate points is a special case of an interruptible execution. This is what is used to obtain the lower bound for identical processes. However, when processes are not assumed to be identical, clones cannot necessarily be created. The processes performing the block writes at the beginning of the pieces of an interruptible execution are used instead of clones in many places throughout the following lower bound proof. The excess capacity of an interruptible execution, defined below, is used to satisfy an additional need for clones.

Definition 3.2 *An interruptible execution $\alpha = \alpha_1 \cdots \alpha_k$ starting from configuration C with initial object set V and process set $\mathcal{P}$ has* excess capacity e for object set U *if, at the beginning of each piece α_i, there are at least e processes not in $\mathcal{P}$ poised at each object in $V_i \cap U$.*

Note that if $k > 1$ and C' is the configuration obtained from C by executing α_1, then interruptible execution $\alpha_2 \cdots \alpha_k$ also has excess capacity e for U.

The next lemma shows that, from any configuration in which there are sufficiently many processes poised at certain objects, there is an interruptible execution with desired excess capacity. Note that there was no need to prove an analogous lemma in Section 3.1, since the nondeterministic solo termination property guarantees the existence of terminating solo executions (the analogue of interruptible executions).

Lemma 3.4 *Let U and V be sets of objects and let $\mathcal{P}$ be a set of at least $(r^2 + r - v^2 + v)/2 + e|\overline{V} \cap U|$ processes. Consider any configuration C in which there are at least $\overline{v} + 1$ processes in $\mathcal{P}$ poised at every object in V and at least e processes not in $\mathcal{P}$ poised at every object in $V \cap U$. Then there is an interruptible execution starting from C with initial object set V and process set $\mathcal{P}$ that has excess capacity e for U.*

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

Consider any execution δ starting from configuration C with the following properties:

- there is a set $\widehat{\mathcal{P}} \subseteq \mathcal{P}$ of $v(\overline{v} + 1)$ processes which, at C, contains $\overline{v} + 1$ processes poised at each object in V,
- δ begins with a block write to V by a set $\mathcal{P}_1 \subseteq \widehat{\mathcal{P}}$ of v processes that take no further steps in δ,
- all other steps in δ are taken by processes in $\mathcal{P} - \widehat{\mathcal{P}}$,
- all non-trivial operations in δ are performed on objects in V, and
- at the configuration C' obtained from C by executing δ, either some process has decided or all processes in $\mathcal{P} - \widehat{\mathcal{P}}$ are poised at objects in $\overline{V}$.

Such an execution may be obtained by performing the block write to V by $\mathcal{P}_1$ and then, for each process in $\mathcal{P} - \widehat{\mathcal{P}}$, performing steps of a solo terminating execution until the process has decided or is poised at an object in $\overline{V}$.

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

For every integer $i \geq 1$, let y_i and z_i be the number of objects in $\overline{V} \cap \overline{U}$ and $\overline{V} \cap U$, respectively, at which at least i processes in $\mathcal{P} - \widehat{\mathcal{P}}$ are poised in configuration C'. Then $y_i \geq y_{i+1}$ and $z_i \geq z_{i+1}$.

Furthermore, there exists $i \in \{1, \ldots, \overline{v}\}$ such that $y_i + z_{e+i} \geq \overline{v} - i + 1$. To see why, suppose to the contrary that $y_i + z_{e+i} \leq \overline{v} - i$ for all $1 \leq i \leq \overline{v}$. In particular, $y_{\overline{v}} + z_{e+\overline{v}} \leq 0$, so $y_i = z_{e+i} = 0$ for all $i \geq \overline{v}$. Then, since $v + \overline{v} = r$ and $\overline{v} > 0$,

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

This is a contradiction.

Consider the situation at configuration C'. Suppose $Y \subseteq \overline{V} \cap \overline{U}$ and $Z \subseteq \overline{V} \cap U$ are sets of objects such that there are at least i processes in $\mathcal{P} - \widehat{\mathcal{P}}$ poised at every object in Y, there are at least $e + i$ processes in $\mathcal{P} - \widehat{\mathcal{P}}$ poised at every object 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 $i = \overline{v'} + 1$.

Let $\mathcal{E} \subseteq \mathcal{P} - \widehat{\mathcal{P}}$ be a set of $e|Z|$ processes, e poised at every object in $Z = (V' - V) \cap U$. Define $\mathcal{P}' = \mathcal{P} - \mathcal{P}_1 - \mathcal{E}$. At C, there are at least e processes not in $\mathcal{P}$ (and hence not in $\mathcal{P}'$) poised at every object in $V \cap U$ and these processes take no steps in δ. Also, none of the processes in $\mathcal{E}$ are in $\mathcal{P}'$. Therefore, at C', there are at least e processes not in $\mathcal{P}'$ poised at every object in $V' \cap U$.

There are at least $i = \overline{v'} + 1$ processes in $\mathcal{P} - \widehat{\mathcal{P}} - \mathcal{E} \subseteq \mathcal{P}'$ poised at every object in $Y \cup Z = V' - V$. Furthermore, since none of the processes in $\widehat{\mathcal{P}} - \mathcal{P}_1$ take any steps in δ, it follows that, at configuration C', there are also at least $\overline{v} \geq \overline{v'} + 1$ processes in $\mathcal{P}'$ poised at every object in V.

Finally, since $v \leq v' - 1$,

$$\begin{aligned}
|\mathcal{P}'| &= |\mathcal{P}| - v - e|(V' - V) \cap U| \\
&\geq (r^2 + r - v^2 + v)/2 + e|\overline{V} \cap U| - v - e|(V' - V) \cap U| \\
&= (r^2 + r - v^2 - v)/2 + e|\overline{V'} \cap U| \\
&\geq (r^2 + r - (v')^2 + v')/2 + e|\overline{V'} \cap U|.
\end{aligned}$$

Then, by the induction hypothesis, there is an interruptible execution δ' starting from C' with initial object set V' and process set $\mathcal{P}'$ that has excess capacity e for U.

Now $\mathcal{P}' \subseteq \mathcal{P} - \mathcal{P}_1$ and $V' \supsetneq V$. Therefore $\delta\delta'$ is an interruptible execution starting from C with initial object set V and process set $\mathcal{P}$ that has excess capacity e for U. ∎

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, but is more difficult, because we cannot just make a sufficient number of clones as we need them. Instead, we use excess capacity in one execution to guarantee a sufficient number of processes poised at the particular objects needed for the other execution.

Lemma 3.5 *Let α be an interruptible execution, starting at configuration C, with initial object set V, process set $\mathcal{P}$, and excess capacity $\overline{w}$ for $\overline{W}$ and let β be an interruptible execution, starting at configuration C, with initial object set W, process set $\mathcal{Q}$, and excess capacity $\overline{v}$ for $\overline{V}$. Suppose $|\mathcal{P}| \geq (r^2 + r - v^2 + v)/2 + \overline{w} \cdot |\overline{V} \cap \overline{W}|$, $|\mathcal{Q}| \geq (r^2 + r - w^2 + w)/2 + \overline{v} \cdot |\overline{V} \cap \overline{W}|$, and $\mathcal{P}$ and $\mathcal{Q}$ are disjoint. If α decides 0 and β decides 1, 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_1 be the first piece of α and let C' be the configuration obtained from C by executing α_1. All of the non-trivial operations in α_1 are to objects in $V \subseteq W$, $\mathcal{P}$ and $\mathcal{Q}$ are disjoint, and β begins with a block write to W by a set $\mathcal{Q}_1$ of w processes that take no further steps in β. This implies that the configurations obtained from C and C' as a result of performing the block write to W are indistinguishable to the processes in $\mathcal{Q} - \mathcal{Q}_1$. Furthermore, all processes poised at objects in $\overline{V}$ at configuration C are poised at the same objects at configuration C'. Therefore, β is an interruptible execution starting from C', with initial object set W and process set $\mathcal{Q}$, that decides 1 and has excess capacity $\overline{v}$ for $\overline{V}$. If some process has decided 0 at C', then $\alpha_1\beta$ is an execution from C that decides both 0 and 1. Otherwise, $\alpha = \alpha_1\alpha'$, where α' is an interruptible execution α' starting from C' with process set

$\mathcal{P}$ and some initial object set $V' \supsetneq V$ that decides 0 and has excess capacity $\overline{w}$ for $\overline{W}$. Since $\overline{V'} \subsetneq \overline{V}$,

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

and β has excess capacity $\overline{v'}$ for $\overline{V'}$. By the induction hypothesis applied to α' and β, there is an execution δ starting from C' that decides both 0 and 1. Hence, $\alpha_1\delta$ 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 β has excess capacity $\overline{v} \geq \overline{v'}+1$ for $\overline{V} \supseteq \overline{V'}$, β has excess capacity $\overline{v'}$ for $\overline{V'}$. In addition, there are $\overline{v'}+1$ processes not in $\mathcal{Q}$ poised at each object in $W \cap \overline{V} = V' - V$. Form $\mathcal{P}'$ by adding these processes to $\mathcal{P}$, if they are not already in $\mathcal{P}$. Note that $\mathcal{P}'$ and $\mathcal{Q}$ are disjoint. There are also at least $\overline{v}+1 \geq \overline{v'}+1$ processes in $\mathcal{P} \subseteq \mathcal{P}'$ poised at each object in V, by assumption. Since α has excess capacity $\overline{w}$ for $\overline{W}$, there are $\overline{w}$ processes not in $\mathcal{P}$ poised at each object in $V \cap \overline{W} = V' \cap \overline{W}$. These processes are not in $\mathcal{P}'$, because the processes in $\mathcal{P}' - P$ are poised at objects in $\overline{V}$. Since $V \subsetneq V'$,

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

By Lemma 3.4, there exists an interruptible execution α' starting from configuration C with initial object set V' and process set $\mathcal{P}'$ that has excess capacity $\overline{w}$ 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. Note that α' has excess capacity $\overline{w'}$ for $\overline{W'}$, since $\overline{W'} \subsetneq \overline{W}$.

Similarly, we may assume that there exists an interruptible execution β' starting from C with initial object set W' and process set $\mathcal{Q}'$ that decides 0 and has excess capacity $\overline{v}$ for $\overline{V}$, where $|\mathcal{Q}'| \geq (r^2+r-(w')^2+w')/2+\overline{v}\cdot|\overline{W'}\cap\overline{V}|$, $\mathcal{Q}' \supseteq \mathcal{Q}$ is disjoint from $\mathcal{P}$, and all processes in $\mathcal{Q}'-\mathcal{Q}$ are poised at objects in $W'-W$.

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

Lemma 3.6 *There is no implementation of consensus satisfying nondeterministic solo termination from r historyless objects using $3r^2+r$ or more processes.*

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

Let $V = W = \varnothing$. By Lemma 3.4, there is an interruptible execution α starting from the initial configuration with initial object set V and process set $\mathcal{P}$ that has excess capacity $\overline{w}$ 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 object set W and process set $\mathcal{Q}$ that has excess capacity $\overline{v}$ for $\overline{V}$ and decides 1. Hence, Lemma 3.5 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 3.6.

Theorem 3.7 *A randomized wait-free implementation of n-process binary consensus requires $\Omega(\sqrt{n})$ objects, if the objects are historyless.*

4 Separation Results

We use our main theorem to derive a series of results comparing the "randomized power" of various synchronization primitives with their "deterministic power" [20]. We say that object type X is *deterministically more powerful* than object type Y if the number of processes for which consensus can be achieved is larger using instances of X than using instances of Y. For randomized computation, we say that object type X is *more powerful* than object type Y if n-process randomized consensus requires asymptotically fewer instances of X than instances of Y.

Objects with only interfering operations cannot deterministically solve 3-process consensus [20]. Hence, they are deterministically less powerful than objects with operations such as COMPARE&SWAP, which can solve n-process consensus.

Consider any object with an operation such that, starting with some particular state, the response from one application of the operation is always different than the response from the second of two successive applications of that operation. (For example, a register with the value 0 returns different values from successive applications of SWAP(1). The operation FETCH&ADD applied starting with any value also has this property.) Then this object can solve 2-process consensus. Therefore, it is deterministically more powerful than the *read-write* register, which cannot solve 2-process consensus.

Herlihy [20, Theorem 5] shows that n-process consensus can be implemented deterministically using a single bounded *compare&swap* register. Hence, from Theorems 2.1 and 3.7, we have the following result.

Corollary 4.1 *Any randomized non-blocking bounded compare&swap register implementation requires $\Omega(\sqrt{n})$ objects, if the objects are historyless.*

Since there are deterministic counter implementations using $O(n)$ read-write registers [9, 30], neither counters nor bounded counters can deterministically solve 2-process consensus [20].

Aspnes [7] gives a randomized algorithm for n process binary consensus using three bounded counters: the first two keep track of the number of processes with input 0 and input 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 [8].

Theorem 4.2 (Aspnes) *There is a randomized consensus implementation using one bounded counter.*

The next result follows from Theorems 2.1, 3.7, and 4.2.

Corollary 4.3 *Any randomized non-blocking bounded counter implementation requires $\Omega(\sqrt{n})$ objects, if the objects are historyless.*

Surprisingly, the lower bounds in Corollaries 4.1 and 4.3 are both independent of the number of values an object can assume: they hold even when the objects (such as *read-write* and *swap* registers) used in the implementation have an infinite number of values, but the object being implemented has a finite number of values.

The same result holds for the implementation of a *fetch&add* register, a *fetch&increment* register, or a *fetch&decrement* register, because a single instance of any of these objects can be easily used to implement a counter.

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

Corollary 4.5 *Any randomized non-blocking implementation of a fetch&add register, a fetch&increment register, or a fetch&decrement register requires $\Omega(\sqrt{n})$ objects, if the objects are historyless.*

Theorem 4.4 is particularly interesting since it shows that *fetch&add* and *compare&swap*, which differ substantially in their deterministic power [20], have similar randomized power in the sense that one instance of each suffices to solve randomized consensus.

5 Conclusions

We have presented a separation among multiprocessor synchronization primitives based on the *space complexity* of randomized solutions to n-process binary consensus. Our main result proved that $\Omega(\sqrt{n})$ objects are necessary to solve randomized n-process binary consensus, if the objects are historyless. Randomized n-process consensus can be solved using $O(n)$ read-write registers of bounded size and we conjecture that the true space complexity of this problem is $\Theta(n)$. We believe that, based on our approach, a larger lower bound may be possible by reusing the processes that perform the block writes at the beginning of a piece of an interruptible execution.

We further believe that our lower bound approach can be extended to allow comparisons among other classes of primitives, and help us to better understand the limitations of using randomization to implement various synchronization primitives from one another. The lower bounds presented here only consider the implementation of a "single access" object, but also apply to the implementation of a "multiple use" object, where each process can access the object repeatedly. However, it may be that the implementation of certain multiple use objects, for example, real-world synchronization primitives such as *test&set* and *fetch&add*, is significantly more difficult and that improved lower bounds can be obtained by having some processors access the implemented object many times. Indeed, a recent result by Jayanti, Tan, and Toueg [22] shows that for multiple use objects, it takes $n-1$ instances of objects such as registers or *swap* registers to implement objects such as *increment* registers, *fetch&add* registers, and *compare&swap* registers.

Needless to say, there is also much work to be done in providing efficient upper bounds for randomized implementations of objects.

Acknowledgments

This work was supported by the Natural Science and Engineering Research Council of Canada grant A1976 and the Information Technology Research Centre of Ontario, ONR grant N00014-91-J-1046, NSF grant 8915206-CCR and 9520298-CCR, and DARPA grants N00014-92-J-4033 and N00014-91-J-1698. A preliminary version of this paper appears in the Proceedings of 12th Annual ACM Symposium on Principles of Distributed Computing, August 1993, pages 241–249. Part of this work was performed while Faith Fich and Nir Shavit were visiting MIT.

References

[1] Yehuda Afek and Gideon Stupp. "Synchronization power depends on the register size." *In Proc. of the 34th Annual IEEE Symposium on Foundations of Computer Science*, November 1993, pp. 196-205.

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

[3] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic Snapshots of Shared Memory", JACM, vol. 40, no. 4, pp. 873–890.

[4] Y. Afek, D. Greenberg, M. Merritt, and G. Taubenfeld, "Computing with Faulty Shared Objects", *Journal of the ACM*, November 1995, pp. 1231-1274.

[5] 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.

[6] J. Aspnes, "Lower Bounds for Distributed Coin-Flipping and Randomized Consensus", *Proc. 29th Annual ACM Symposium on Theory of Computing*, May 1997, pp. 559–568.

[7] 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.

[8] J. Aspnes, private communication.

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

[10] J. Aspnes and O. Waarts. "Randomized Consensus in Expected $O(n \log^2 n)$ operations per processor," In Proceedings of the *33rd Annual Symposium on the Foundations of Computer Science*, October 1992, pp. 137–146.

[11] 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.

[12] B. Bershad, "Practical Considerations for Non-Blocking Concurrent Objects," *Proceedings of the 13th International Conference on Distributed Computing Systems*, pp. 264-274, IEEE Computer Society Press, May 1993.

Practical Considerations for Non-Blocking Concurrent Objects Proceedings of the 13th International Conference on Distributed Computing Systems, .

[13] G. Bracha and O. Rachman, "Randomized Consensus in Expected O($n^2 logn$) Operations," *5th International Workshop on Distributed Algorithms,* Delphi, Greece, October 1991. Lecture Notes in Computer Science, vol. 579, Springer-Verlag, 1992, pp. 143–150.

[14] J. Burns and N. Lynch. "Mutual Exclusion Using Indivisible Reads and Writes," *Proc. 18th Annual Allerton Conference on Communication, Control and Computing*, 1989, pp. 833–842.

[15] 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.

[16] D. Dolev, C. Dwork and L. Stockmeyer, "On the Minimal Synchrony Needed for Distributed Consensus," *Journal of the ACM,* vol. 34, no. 1, January 1987, pp. 77–97.

[17] Cynthia Dwork and Maurice Herlihy and Serge A. Plotkin and Orli Waarts, "Time-lapse snapshots," *Stanford University, Department of Computer Science*, Technical Report, STAN//CS-TR-92-1423, 1992.

[18] 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.

[19] 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.

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

[21] 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.

[22] P. Jayanti, K. Tan, and S. Toueg, "Time and Space Lower Bounds for Non-blocking Implementations (Preliminary Version)", *Proc. 15th Annual ACM Symposium on Principles of Distributed Computing*, May 1996, pp. 257–266.

[23] P. Jayanti, T. Chandra, and S. Toueg, "Fault-tolerant wait-free shared objects," TR 92-1281, Department of Computer Science, Cornell University, April 1992.

[24] E. Kushilevitz, Y. Mansour, M. Rabin, and D. Zuckerman. "Lower Bounds for Randomized Mutual Exclusion," In *Proc. 25th Annual ACM Symposium on the Theory of Computing*, San-Diego, May 1993, pp. 154–163.

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

[26] M. Loui and H. Abu-Amara, "Memory Requirements for Agreement Among Unreliable Asynchronous Processes," *Advances in Computing Research*, vol. 4, JAI Press, Inc., 1987, pp. 163–183.

[27] N.A. Lynch, "Distributed Algorithms." Morgan Kaufmann, San Francisco, 1996.

[28] 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.

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

[30] S. Moran, G. Taubenfeld, and I. Yadin , "Concurrent Counting", *Proc. 11th Annual ACM Symposium on Principles of Distributed Computing*, August 1992, pp. 59–70.

[31] A. Pogosyants, R. Segala, and N. Lynch, "Verification of the Randomized Consensus Algorithm of Aspnes and Herlihy: a Case Study," Unpublished manuscript, MIT, 1996.

[32] 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.