Distributed Computing manuscript No.
(will be inserted by the editor)

Faith Ellen Fich · **Danny Hendler** · **Nir Shavit**

On the Inherent Weakness of Conditional Primitives

Received: date / Accepted: date

Abstract Some well-known primitive operations, such as *compare-and-swap*, can be used, together with *read* and *write*, to implement any object in a wait-free manner. However, this paper shows that, for a large class of objects, including counters, queues, stacks, and single-writer snapshots, wait-free implementations using only these primitive operations and a large class of others primitive operations cannot be space efficient: the number of base objects required is at least linear in the number of processes that share the implemented object. The same lower bounds are obtained for implementations of starvation-free mutual exclusion using only primitive operations from this class. For wait-free implementations of a closely related class of one-time objects, lower bounds on the tradeoff between time and space are presented.

Keywords space lower bounds · object implementation · mutual exclusion

Faith Ellen Fich
Department of Computer Science
University of Toronto
Toronto, Ontario, CANADA
E-mail: faith@cs.toronto.edu

Danny Hendler
Department of Computer Science
University of Toronto
Toronto, Ontario, CANADA
Present address: Faculty of Industrial Engineering and Management
Technion
Haifa, ISRAEL
E-mail: hendler@ie.technion.ac.il

Nir Shavit
Computer Science Department
Tel-Aviv University
Ramat Aviv, ISRAEL
Scalable Synchronization Research Group
Sun Microsystems Laboratories
Burlington, MA
E-mail: shanir@math.tau.ac.il, shanir@sun.com

1 Introduction

A *wait-free* implementation of a concurrent object guarantees that any process can complete an operation in a finite number of its own steps. An object type is *universal* if objects of this type can be used, together with registers, to deterministically implement any object, shared by any number of processes, in a wait-free manner. Herlihy [7] proved that some object types provided in multiprocessor systems, for example object types that support *compare-and-swap*, are universal and others, such as object types that support only *fetch-and-add*, are not universal. Thus, in order to support deterministic, wait-free implementations of any object, an architecture should support at least one universal object.

Here, we investigate what primitive operations a multiprocessor architecture should provide to support *efficient*, deterministic, wait-free implementations of common objects. We are concerned with space complexity, specifically, the number of objects used by an implementation.

We consider implementations of a class of objects that includes many common objects such as counters, stacks, queues, and single-writer snapshots. We call these objects *visible*, because they support operations whose effects can be seen by other processes.

We also define the concept of a *conditional* primitive. These are operations that change the value of an object only if the object has a particular value. *Compare-and-swap* is an example of a conditional primitive. *Read* is a trivial conditional primitive. *Fetch-and-add* (and restrictions of it, such as *fetch-and-increment*) are not conditional. We prove that any wait-free implementation of a visible object from base objects that only support conditional primitives and *write* must use $\Omega(n)$ base objects, where n is the number of processes sharing the implemented object. We prove the same lower bound for implementations of starvation-free mutual exclusion from such objects.

In contrast, some visible objects can be implemented using only a constant number of base objects, if other primitives are available. For example, a single-writer snapshot object with value set $\{0,1\}^n$ is visible, but can be implemented from a single object that supports *fetch-and-add*. Similarly, starvation-free mutual exclusion can be implemented from a register and an object that supports *fetch-and-increment-mod-n*. (See Section 6.) Thus, our results indicate that an architecture providing only conditional primitives and *write* might not be the best design choice.

Some previous research gives complexity lower bounds for the wait-free implementation of objects. Jayanti, Tan, and Toueg [9] showed that any implementation of a *perturbable* object (for example a counter or a single-writer snapshot) from historyless objects and resettable consensus objects requires at least $n-1$ base objects and a process must apply at least $n-1$ steps, in the worst case, to perform an operation on the implemented object. A *historyless* object is an object whose value depends only on the last nontrivial primitive (for example, *write*, *test-and-set*, or *swap*) that was applied to it. A *nontrivial* primitive is an operation that can change the value of the object.

The class of historyless objects and the class of objects that support only conditional primitives are incomparable. For example, *swap* is not a conditional primitive and an object that supports *compare-and-swap* is not historyless. In fact, no historyless object is universal in a system of more than two processes. *Test-and-set* is a conditional primitive and an object that only supports *test-and-set* is historyless.

Mutual exclusion is another problem for which there are good lower bounds. Burns and Lynch [3] proved that at least n registers are required to solve n-process mutual exclusion. We extend their linear lower bound for mutual exclusion to starvation-free implementations from objects that support any conditional primitives.

Cypher [4] also considered implementations of n-process starvation-free mutual exclusion using registers and objects that support conditional primitives. He proved a lower bound of $\Omega(n \log\log n / \log\log\log n)$ on the total number of remote memory references performed by n processes as they each enter and exit the critical section once. Anderson and Kim [1] considered the same class of base objects and proved a lower bound of $\Omega(\log n / \log\log n)$ on the worst-case number of remote memory references a process performs as it enters and exits the critical section. Anderson and Yang [10] showed that starvation-free mutual exclusion can be solved using $O(n)$ registers, where each entry to and exit from the critical section performs $O(\log n)$ remote memory references. Note that these bounds almost match the known lower bounds. In contrast, starvation-free mutual exclusion can be implemented using *fetch-and-add* or *swap* with $O(1)$ remote memory references for each entry to and exit from the critical section [2].

One-time objects are objects on which each process may perform at most one operation. Some visible objects, when restricted in this way, do have more space efficient implementations. (See Section 7.) Fich, Herlihy, and Shavit [6] proved a lower bound of $\Omega(\sqrt{n})$ on the space complexity of randomized wait-free implementations of n-process consensus, which is a universal one-time object. In addition to registers, the implementations they considered could also use historyless objects. Jayanti [8] proved an $\Omega(\log n)$ lower bound on the number of steps to perform *fetch-and-increment* on a one-time object implemented from base objects that support only *load-linked*, *store-conditional*, *swap*, and *move*. He also gets the same result for implementing one-time stacks and queues that initially contain a fixed sequence of n different elements.

We define a class of *one-time-visible* objects analogous to the class of visible objects. For implementations from base objects that only support conditional primitives and *write*, we prove lower bounds on time-space tradeoffs for one-time-visible objects. Specifically, we prove that any such implementation either uses at least $\sqrt{n}$ base objects, or the total number of applications of primitives to base objects, when each process performs one operation on the implemented object, is in $\Omega(n^{3/2})$. For these implementations, we also prove a lower bound on the tradeoff between space and the delay induced by memory contention. We show that any such implementation either uses at least $n^{\frac{2}{3}}$ base objects, or the total number of *memory stalls* incurred by processes when each of them performs one operation on the implemented object is in $\Omega(n^{5/3})$. The concept of memory stalls was defined by Dwork, Herlihy, and Waarts[5] in a paper that introduced a formal model to capture the phenomenon of memory contention in shared memory multiprocessors.

The rest of the paper is organized as follows. We begin, in Section 2, with a description of our shared-memory model. Then, in Sections 3 and 4, we formally define conditional primitives, invisible events, and the classes of visible and one-time-visible objects, give some examples, and prove some simple properties. Section 5 contains our linear space lower bound for implementations of visible objects. This is followed, in Section 6, by our linear space lower bound for starvation-free mutual exclusion. Finally, in Section 7, we prove lower bounds on time-space tradeoffs for implementations of one-time-visible objects.

2 The Shared Memory System Model

We consider a standard model of an asynchronous shared memory system in which deterministic processes communicate by applying operations to deterministic shared objects. An *object* is an instance of an object type. An *object type* is an abstract data type, which is characterized by a set of possible values and by a set of *operations* that provide the only means to manipulate it. The set of possible values of an object need not be bounded.

An implementation of an object that is shared by a set, **P**, of n processes provides a data representation using a set, **B**, of shared *base objects*, each of which is assigned an initial value, and algorithms for each process in **P** to perform each operation on the object being implemented. To avoid confusion, we call operations on the base objects *primitives*. We reserve the term *operations* for the objects being implemented.

We consider *wait-free implementations*, in which each process is guaranteed to complete an operation within a finite number of its own steps. Each step of a process consists of some local computation and one shared memory *event*, which is a primitive applied to a base object in **B**. We say that the process *applies* this event.

An *execution fragment* is a (finite or infinite) sequence of events. An *execution* is an execution fragment that starts from an initial state, the subsequence of events performed by each process is consistent with its algorithm, and each object behaves according to its sequential specification. An *initial state* is a state in which all base objects in **B** have their initial values and all processes are idle. For any finite execution fragment E and any execution fragment E', the execution fragment $E \circ E'$ denotes the concatenation of E and E'. Note that $E \circ E'$ is not necessarily an execution, even if both E and E' are executions. If E and $E \circ E'$ are both executions, we say that E' is an *extension* of E. If $r \in \mathbf{B}$ is a base object and E is a finite execution, then $val(E,r)$ denotes the value of r at the end of E. If no event in E changes the value of r, then $val(E,r)$ is the initial value of r. In other words, in the state resulting from executing E, each base object $r \in \mathbf{B}$ has value $val(E,r)$.

In an execution, each process performs a sequence of operation instances on the implemented object. Each *operation instance* consists of a single operation with specified arguments performed by a single process on a single object. To perform an operation instance, a process applies a sequence of one or more events. The response from an event can influence which events occur later in the sequence. To avoid confusion, we call a sequence of operation instances performed on an implemented object a *history*, and reserve the term *execution* for a sequence of events applied to base objects. We say that a history is *p-free* if none of the operation instances in the history are performed by process p.

A process applies all the events of one operation instance before starting to apply events of another operation instance. In other words, a process can only perform one operation instance at a time. The events of an operation instance performed by a process can be interleaved with events applied by other processes. Thus, one history can give rise to many different executions, depending on how events by different processes are scheduled.

We use $res(H, \Phi)$ to denote the response of an operation instance Φ in a history H. If an execution E contains an event applied by a process while it is performing an operation instance Φ, we say that Φ *occurs* in E. If this event is the last event the process applies while performing Φ, we say that Φ *completes* in E. In this case, we let $res(E, \Phi)$ denote the response returned to the process by the operation instance Φ. We say that a process is *active* at the end of a finite execution E if the process is in the middle of performing some operation instance. This means that the process has begun an operation instance (even if it has not yet applied any events of this operation instance), but this operation instance does not complete in E. If a process is active in the state resulting from a finite execution, it has exactly one *enabled* event, which is the next event it will apply, as specified by the protocol it is using to perform its current operation instance on the implemented object. If p is not active, we say it is *idle*. An idle process has no enabled event.

Two executions E and E' are *indistinguishable* to a set of processes, if, in both executions, all processes in this set apply exactly the same sequence of events and they get the same responses from each of these events, and the values of all objects are the same at the end of both executions.

We consider base objects that support a set of atomic read-modify-write primitives. A read-modify-write (RMW) primitive can be defined by two functions: the *update function*, g, and the *response function*, h. Both take two parameters, the current value v of the object to which the primitive is to be applied and the input w to the primitive, if any. The update function specifies the value of the object after the primitive has been applied and the response function specifies the value that is returned to the process that applied the primitive. The event RMW(p,r,w,g,h), in which process $p \in \mathbf{P}$ applies the RMW primitive to object $r \in \mathbf{B}$, has the effect of atomically performing the code shown in Figure 1.

RMW(p, r, w, g, h)
  $v \leftarrow$ read from r
  write $g(v,w)$ to r
  return $h(v,w)$

Fig. 1 The semantics of a read-modify-write event by process p

Compare-and-swap and *test-and-set* are two examples of RMW primitives. Their update and response functions are illustrated in Figures 2 and 3, respectively. The *test-and-set* primitive does not take an input. We use the '$-$' symbol as the second parameter of its update and response functions to denote this.

$g_{cas}(v, \langle old, new \rangle)$
  if ($v = old$)
    then return *new*
    else return v

$h_{cas}(v, \langle old, new \rangle)$
  if ($v = old$)
    then return true
    else return false

Fig. 2 The update and response functions of the *compare-and-swap* primitive with input $w = \langle old, new \rangle$

$g_{ts}(v, -)$
  return 1

$h_{ts}(v, -)$
  if ($v = 0$)
    then return true
    else return false

Fig. 3 The update and response functions of the *test-and-set* primitive

Write is an example of a RMW primitive. Its update function is $g(v,w) = w$ and its response function always returns *ack*. The event Write(p,r,w) indicates that process $p \in \mathbf{B}$ atomically writes the value w to object $r \in \mathbf{B}$. *Read* is another example of a RMW primitive. It takes no input and its update and response functions are $g(v,-) = h(v,-) = v$. The event RMW(p,r,w,g,h) is *trivial* if $g(v,w) = v$ for all possible values v of object r; otherwise it is *nontrivial*.

3 Conditional Primitives and Invisibility

This section formally defines the class of conditional RMW primitives. It also introduces the concept of *invisible* events and proves some properties of such events.

Definition 1 A RMW primitive $\langle g,h\rangle$ on an object with value set V is *conditional* if, for every possible input w,

$$\left|\left\{v \in V | g(v,w) \neq v\right\}\right| \leq 1.$$

A value $c_w \in V$ such that $g(c_w,w) \neq c_w$ is called a *change point* of the event RMW(p,r,w,g,h). Any value $v \in V$ such that $g(v,w) = v$ is called *a fixed point* of this event.

Compare-and-swap and *test-and-set* are both conditional primitives. *Read* is also a conditional primitive. *Fetch-and-increment* is not conditional. *Write* is not a conditional primitive unless the object to which it is applied has a value set with at most two values.

An object that supports only conditional primitives is called a *conditional object* and an object that supports only *write* and conditional primitives is called a *write-conditional* object. All conditional objects are write-conditional. A register, which supports *read* and *write*, and a resettable consensus object, which supports *propose* and *reset*, are examples of write-conditional objects that are not conditional.

Informally, an invisible event is an event by some process that cannot be observed by other processes. All read events are invisible. A write event is invisible if the value of the object to which it is applied is the same as the value it writes. A write event is also invisible if it is overwritten by another process before the process takes another step and before the value it wrote can be accessed by another process. This is also called an *obliterated* event [3]. We extend this notion to all RMW events in the following definition.

Definition 2 Let e be a RMW event applied by process p to an object r in an execution $E = E_1 \circ e \circ E_2$. We say that e is *invisible in* E if either the value of r is not changed by e or $E_2 = E_2' \circ e' \circ E_3$, $E_2' \circ e'$ is p-free, no events in E_2' are applied to r, and e' is a write event to r.

A RMW event is invisible if the value of the object to which it is applied is a fixed point of the event. Any RMW event applied to an object is also invisible if a write event is subsequently applied to the object before the value of the object is read by another process. If a RMW event e is not invisible in an execution E, we say that e is a *visible* event in E.

The following observations are direct consequences of this definition.

Proposition 1 *If an event e is invisible in an execution E, then e is invisible in any execution $E \circ E'$.*

Proposition 2 *If e and e' are events by process p, e is invisible in the execution $E \circ e \circ E' \circ e' \circ E''$, and E' is p-free, then e is invisible in execution $E \circ e \circ E'$.*

Proposition 3 *If two executions E and E' are indistinguishable to a set of processes Q, $E \circ E_1$ is an execution, and all of the events in E_1 are applied by processes in Q, then $E' \circ E_1$ is an execution which is indistinguishable from $E \circ E_1$ to the processes in Q.*

Proof Let $E_2 \circ e$ be any nonempty prefix of E_1 and suppose that $E' \circ E_2$ is an execution which is indistinguishable from $E \circ E_2$ to the processes in Q. Since the process that applies e is in the same state at the end of $E \circ E_2$ and $E' \circ E_2$, it follows that $E' \circ E_2 \circ e$ is an execution. Since the object to which e is applied has the same value at the end $E \circ E_2$ and $E' \circ E_2$, it will be updated to the same value by e and e will return the same response when applied at the end of these two executions. The values of no other objects and the states of no other processes are changed by e. Thus $E \circ E_2 \circ e$ and $E' \circ E_2 \circ e$ are indistinguishable to the processes in Q. It follows by induction that $E' \circ E_1$ is an execution which is indistinguishable from $E \circ E_1$ to all processes in Q.

Lemma 1 *If events $e_1,\ldots,e_k$ by process p are invisible in execution $E_0 \circ e_1 \circ E_1 \circ \cdots \circ E_{k-1} \circ e_k \circ E_k$ and $E_1,\ldots,E_k$ are p-free, then $E_0 \circ E_1 \circ \cdots \circ E_{k-1} \circ E_k$ is an execution which is indistinguishable from $E_0 \circ e_1 \circ E_1 \circ \cdots \circ E_{k-1} \circ e_k \circ E_k$ to all processes in $\mathbf{P} - \{p\}$.*

Proof The proof is by induction on k. When $k = 0$, the two executions are the same, so let $k > 0$ and assume that the claim is true for $k-1$. Suppose that events $e_1,\ldots,e_k$ by process p are invisible in execution $E_0 \circ e_1 \circ E_1 \circ \cdots \circ E_{k-1} \circ e_k \circ E_k$ and $E_1,\ldots,E_k$ are p-free.

Let r be the object to which event e_k is applied. If e_k does not change the value of r, then $E_0 \circ e_1 \circ E_1 \circ \cdots \circ e_{k-1} \circ E_{k-1}$ and $E_0 \circ e_1 \circ E_1 \circ \cdots \circ e_{k-1} \circ E_{k-1} \circ e_k$ are indistinguishable to all processes in $\mathbf{P} - \{p\}$.

Otherwise, by definition of invisibility, $E_k = E_k' \circ e' \circ E_k''$, where no events in E_k' are applied to r, and e' is a write event to r. Then executions $E_0 \circ e_1 \circ E_1 \circ \cdots \circ e_{k-1} \circ E_{k-1} \circ E_k' \circ e'$ and $E_0 \circ e_1 \circ E_1 \circ \cdots \circ e_{k-1} \circ E_{k-1} \circ e_k \circ E_k' \circ e'$ are indistinguishable to all processes in $\mathbf{P} - \{p\}$.

In both cases, Proposition 3 implies that $E_0 \circ e_1 \circ E_1 \circ \cdots \circ e_{k-1} \circ E_{k-1} \circ E_k$ is an execution which is indistinguishable from $E_0 \circ e_1 \circ E_1 \circ \cdots \circ e_{k-1} \circ E_{k-1} \circ e_k \circ E_k$ to all processes in $\mathbf{P} - \{p\}$.

For $1 \leq i \leq k-1$, e_i is invisible in $E_0 \circ e_1 \circ E_1 \circ \cdots \circ e_{k-1} \circ E_{k-1} \circ e_k \circ E_k$, so, by Proposition 2, e_i is invisible in

$E_0 \circ e_1 \circ E_1 \circ \cdots \circ E_{i-1} \circ e_i \circ E_i$. Hence, by Proposition 1, e_i is invisible in $E_0 \circ e_1 \circ E_1 \circ \cdots \circ e_{k-1} \circ E_{k-1} \circ E_k$.

Since $E_{k-1} \circ E_k$ is p-free, the induction hypothesis implies that $E_0 \circ E_1 \circ \cdots \circ E_{k-1} \circ E_k$ is an execution which is indistinguishable from $E_0 \circ e_1 \circ E_1 \circ \cdots \circ e_{k-1} \circ E_{k-1} \circ E_k$ to all processes in $\mathbf{P} - \{p\}$. Hence, by transitivity, $E_0 \circ E_1 \circ \cdots \circ E_{k-1} \circ E_k$ and $E_0 \circ e_1 \circ E_1 \circ \cdots \circ e_{k-1} \circ E_{k-1} \circ e_k \circ E_k$ are indistinguishable to all processes in $\mathbf{P} - \{p\}$.

4 Visible and One-Time-Visible Objects

In this section, we define the classes of visible and one-time-visible objects. These are the objects to whose implementations our lower bounds apply. We define both classes in terms of the sequential specifications of the objects, rather than in terms of properties of executions of their implementations. This makes it easier to show that specific objects belong to these classes.

We start with the definition of one-time-visible objects. A one-time-visible object is a one-time object, which means that every process may perform at most one operation instance on it. In addition, every process has an operation instance it may perform that must affect the response of a subsequent operation instance performed by some other process.

Definition 3 A one-time object is *one-time-visible* if for every process $p \in \mathbf{P}$, there is an operation instance Φ_p by p such that, for all processes $q, q' \in \mathbf{P}$ and every q'-free history $H_1 \circ \Phi_q \circ H_2$ consisting of a subset of $\{\Phi_p | p \in \mathbf{P}\}$, there exists an operation instance Φ' performed by process q' so that $res(H_1 \circ \Phi_q \circ H_2 \circ \Phi', \Phi') \neq res(H_1 \circ H_2 \circ \Phi', \Phi')$. The operation instance Φ_p is called p's *witness*.

Note that the operation instance Φ' performed by process q' in Definition 3 may be different from its witness $\Phi_{q'}$.

A *counter* is an object whose values are the non-negative integers. It supports a single parameterless operation, *fetch-and-increment*, which increments the value of the object. A *modulo-k counter* is an object whose set of values is $\{0, 1, \ldots, k-1\}$, where $k > 1$. It supports a single parameterless operation, *fetch-and-increment-mod-k*. This operation increments the value of the object, if its value is less than $k-1$, and resets it to 0, if its value is $k-1$. The operation returns the old value of the object as its response. A *threshold-k counter* is a similar object whose set of values is $\{0, 1, \ldots, k\}$, where $k \geq 1$, and whose single parameterless operation, *fetch-and-increment-upto-k*, increments the value of the object, if its value is less than k, and leaves it unchanged, if its value is k. A one-time version of any of these objects restricts each process from performing more than one operation instance on the object. Note that, in a system with n processes, a one-time counter, a one-time modulo-$(n+1)$ counter, and a one-time threshold-n counter behave the same, if they all have initial value 0, since at most n operation instances can be applied to each of these objects.

Lemma 2 *A one-time modulo-k counter object is one-time-visible.*

Proof Let Φ_p be an instance of *fetch-and-increment-mod-k* for each process $p \in \mathbf{P}$. Let $q, q' \in \mathbf{P}$ and let $H_1 \circ \Phi_q \circ H_2$ be a q'-free history of a one-time modulo-k counter. This history consists of at most $n-1$ instances of *fetch-and-increment-mod-k*. Then $res(H_1 \circ \Phi_q \circ H_2 \circ \Phi_{q'}, \Phi_{q'}) = res(H_1 \circ H_2 \circ \Phi_{q'}, \Phi_{q'}) + 1 \bmod k \neq res(H_1 \circ H_2 \circ \Phi_{q'}, \Phi_{q'})$.

A *queue* is an object whose values are all the finite sequences of elements from a non-empty set V. It supports two operations: *enqueue* and *dequeue*. The *enqueue* operation receives an element of V as input and appends this element to the end of the sequence. It always returns the special value *ack*. If the sequence is not empty, *dequeue* deletes the first element in the sequence and returns it. If the sequence is empty, *dequeue* just returns the special value *empty*. A *one-time queue* is a queue on which each process can perform at most one (*enqueue* or *dequeue*) operation.

Lemma 3 *A one-time queue is not one-time-visible.*

Proof To obtain a contradiction, suppose that there is a witness Φ_p for each process. $p \in \mathbf{P}$. None of these witnesses is a *dequeue*. This is because, in any history beginning with a *dequeue* performed on an initially empty queue, the first operation instance does not change the value of the queue.

Therefore every witness must be an *enqueue*. Let $q, q' \in \mathbf{P}$ and consider any q'-free history $H_1 \circ \Phi_q \circ H_2$ consisting of at most $n-1$ of these operation instances, where H_1 is not empty. Then $res(H_1 \circ \Phi_q \circ H_2 \circ \Phi', \Phi') = res(H_1 \circ H_2 \circ \Phi', \Phi')$ for any operation instance Φ' by q'. This contradicts the assumption that Φ_q is a witness.

We want to extend the definition of one-time-visible to objects that allow processes to perform any number of operation instances. A natural extension is to require that each process has an operation instance it can always perform on the object which must affect the response of the next operation instance performed by one of the other processes. However, this definition does not include all objects supporting operations whose effects can be seen by other processes. For example, consider a queue. The problem with the definition arises when the beginning of the sequence contains n copies of the same element. Then an operation instance Φ_p does not affect the responses given to the other processes when each performs a subsequent operation instance.

One way to generalize the definition is to only require that the operation instance by a process p affects at least one of the responses to some sequence of subsequent operation instances performed by other processes. However, this does not suffice. Consider a *bounded queue* object with *capacity* $k \geq 1$. Its value is any sequence of at most k elements from a non-empty set V. It supports the same operations as a queue, except that the *enqueue* operation returns the special value *full* when performed on an object whose sequence has length k. When the bounded queue is empty, a *dequeue* is invisible and when the bounded queue is full, any *enqueue* is invisible. Thus, there is no single operation that can be used as a witness.

Instead, we use a witness sequence of operation instances for each process and put conditions on the possible finite histories that can arise when each process performs its sequence of operation instances. A *prefix-interleaving* of a finite set of (finite or infinite) sequences is an interleaving of a finite prefix of each of the sequences in the set. We denote the set of all prefix-interleavings of a set of sequences S by $I(S)$. For example, if $S = \{(1,2),(3)\}$ then the set of prefix interleavings of S is $I(S) = \{(), (1), (1,2), (1,3), (1,2,3), (1,3,2), (3), (3,1), (3,1,2)\}$.

Informally, an object is visible if, for every prefix-interleaving of the witness sequences, the last operation instance performed by a process affects at least one of the responses in some sequence of subsequent operation instances performed by other processes. Now we give the formal definition.

Definition 4 An object is *visible* if, for every process $p \in \mathbf{P}$, there is an infinite sequence of operation instances $\Phi_p = \Phi_{p,1}, \Phi_{p,2}, \ldots$, such that, for every finite history $H = H_0 \circ \Phi_{p,i} \circ H_1 \in I(\{\Phi_p | p \in \mathbf{P}\})$, where H_1 is p-free, there is a (possibly empty) p-free extension H_2 of H, with $H \circ H_2$ not necessarily in $I(\{\Phi_p | p \in \mathbf{P}\})$, and an operation instance $\Phi' \in H_1 \circ H_2$ so that $res(H_0 \circ \Phi_{p,i} \circ H_1 \circ H_2, \Phi') \neq res(H_0 \circ H_1 \circ H_2, \Phi')$. The sequence of operation instances Φ_p is called a *witness sequence* for p.

Counters, modulo-k counters, queues, and bounded queues are all examples of visible objects.

Lemma 4 *A modulo-k counter is visible.*

Proof Let Φ_p be an infinite sequence of instances of *fetch-and-increment-mod-k* for each process $p \in \mathbf{P}$. Let $H_0 \circ \Phi_{p,i} \circ H_1 \in I(\{\Phi_p | p \in \mathbf{P}\})$ be a history of a modulo-k counter, where H_1 is p-free. Let H_2 consist of a single *fetch-and-increment-mod-k* operation instance Φ' by a process $q \neq p$. Then $res(H_0 \circ \Phi_{p,i} \circ H_1 \circ H_2, \Phi') = res(H_0 \circ H_1 \circ H_2, \Phi') + 1 \bmod k \neq res(H_0 \circ H_1 \circ H_2, \Phi')$.

Since a modulo-k counter can be simulated using a counter and having each process take the responses it gets modulo k, a counter is also visible. However, a threshold-k counter is not visible, because, in any history consisting of k or more instances of *fetch-and-increment-upto-k*, all operation instances after the k'th return the value k.

Lemma 5 *A queue is visible.*

Proof Let the witness sequence Φ_p, for each process $p \in \mathbf{P}$, be an infinite sequence of *enqueue* operation instances with some element v as input. Let $H_0 \circ \Phi_{p,i} \circ H_1 \in I(\{\Phi_p | p \in \mathbf{P}\})$ be a history of a queue, where H_1 is p-free, and suppose the queue contains e elements at the end of this history. Then $e > 0$. Let H_2 consist of e *dequeue* operation instances performed by some process other than p and let Φ' denote the last operation instance in H_2. Then $res(H_0 \circ \Phi_{p,i} \circ H_1 \circ H_2, \Phi') = v \neq empty = res(H_0 \circ H_1 \circ H_2, \Phi')$.

It is more delicate to prove that a bounded queue with sufficiently large capacity is visible.

Lemma 6 *A bounded queue with capacity at least n is visible.*

Proof Consider a bounded queue with capacity at least n that initially contains f elements. Partition $\mathbf{P}$ into two sets, $\mathbf{P}'$, which contains $\min\{f,n\}$ processes, and $\mathbf{P}''$ which contains the rest. Each witness sequence Φ_p is an infinite sequence of alternating *dequeue* and *enqueue* operation instances. For $p \in \mathbf{P}'$, the witness sequence begins with a *dequeue*, while, for $p \in \mathbf{P}''$, it begins with an *enqueue*.

For any history $H \in I(\{\Phi_p | p \in \mathbf{P}\})$ of this bounded queue and any subset $P \subseteq \mathbf{P}$, let $\Delta_P(H)$ denote the number of processes in P that perform an odd number of operation instances in H. Then the number of elements in the bounded queue at the end of H is $f - \Delta_{\mathbf{P}'}(H) + \Delta_{\mathbf{P}''}(H)$. This number is between $\max\{0, f-n\}$ and $\max\{n, f\}$. Let $H_0 \circ \Phi_{p,i} \circ H_1 \in I(\{\Phi_p | p \in \mathbf{P}\})$ be a history of the bounded queue, where H_1 is p-free. Then $H_0 \circ H_1 \in I(\{\Phi_p | p \in \mathbf{P}\})$.

If $\Phi_{p,i}$ is a *dequeue* and there are e items in the queue at the end of $H_0 \circ \Phi_{p,i} \circ H_1$, then there are $e+1$ items in the queue at the end of $H_0 \circ H_1$. In this case, let H_2 consist of $e+1$ *dequeue* operation instances by some process other than p and let Φ' be the last operation instance in H_2. Then $res(H_0 \circ \Phi_{p,i} \circ H_1 \circ H_2, \Phi') = empty \neq res(H_0 \circ H_1 \circ H_2, \Phi')$.

Similarly, if $\Phi_{p,i}$ is an *enqueue* and there are e items in the queue at the end of $H_0 \circ \Phi_{p,i} \circ H_1$, then there are $e-1$ items in the queue at the end of $H_0 \circ H_1$. In this case, let H_2 consist of e *dequeue* operation instances by some process other than p and let Φ' be the last operation instance in H_2. Then $res(H_0 \circ \Phi_{p,i} \circ H_1 \circ H_2, \Phi') \neq empty = res(H_0 \circ H_1 \circ H_2, \Phi')$.

Stacks and bounded stacks with capacity at least n are also visible. The same proofs work with *enqueue* replaced by *push* and *dequeue* replaced by *pop*.

A *single-writer snapshot* object with the set of values $V_1 \times \cdots \times V_n$ supports two operations: *scan* and *update*. An $update(v)$ by process p sets the p'th component of the value to $v \in V_p$. A *scan* returns the value of (all n components) of the object.

Lemma 7 *A single-writer snapshot object is visible if each component has at least two possible values.*

Proof Consider a single-writer snapshot object. Let u_p be the initial value of the p'th component of the object and let v_p be another possible value of the p'th component. Let the witness sequence Φ_p for process p be an alternating sequence of $update(v_p)$ and $update(u_p)$ operation instances, starting with $update(v_p)$. Then, in every history $H \in I(\{\Phi_p \mid p \in \{1,\ldots,n\}\})$ of this object, every operation instance changes the value of a single component.

Let $H_0 \circ \Phi_{p,i} \circ H_1 \in I(\{\Phi_p \mid p \in \{1,\ldots,n\}\})$ be a history such that H_1 is p-free. Let H_2 consist of a single *scan* operation instance Φ' by a process other than p. Since $\Phi_{p,i}$

changes the value of the p'th component of the object, but no operation instance in $H_1 \circ H_2$ does, it follows that $res(H_0 \circ \Phi_{p,i} \circ H_1 \circ H_2, \Phi') \neq res(H_0 \circ H_1 \circ H_2, \Phi')$.

A *swap object* with a non-empty set of values V supports a single operation, *swap*, with a single input $w \in V$. This operation changes the value of the object to w and returns its previous value.

Lemma 8 *A swap object with a set of at least* $2n$ *values is visible.*

Proof Consider a swap object with a set of at least $2n$ different values. Let u_0 be its initial value and let $\{u_p \mid p \in \mathbf{P}\} \cup \{v_p \mid p \in \mathbf{P}\}$ be a set of $2n$ different values, where $u_p \neq u_0$ for all $p \in \mathbf{P}$. Let the witness sequence Φ_p for process p be an alternating sequence of $swap(u_p)$ and $swap(v_p)$ operation instances, starting with $swap(u_p)$. Then, in every history $H \in I(\{\Phi_p \mid p \in \mathbf{P}\})$, every operation instance changes the value of the swap object.

Let $H_0 \circ \Phi_{p,i} \circ H_1 \in I(\{\Phi_p \mid p \in \mathbf{P}\})$ be a history such that H_1 is p-free. Let H_2 consist of a single instance of *swap* by a process other than p and let Φ' be the first operation instance of $H_1 \circ H_2$. Then $res(H_0 \circ \Phi_{p,i} \circ H_1 \circ H_2, \Phi') \neq res(H_0 \circ H_1 \circ H_2, \Phi')$.

The definition of a visible object can be generalized to allow the input of each operation instance $\Phi_{p,i}$ to be a function of the responses to operation instances $\Phi_{p,1}, \cdots, \Phi_{p,i-1}$.

Lemma 9 *A swap object with a set of at least* $n+1$ *values is visible (under this more general definition).*

Proof Consider a swap object with a set of at least $n+1$ different values. Let u_0 be its initial value and let $\{u_p \mid p \in \mathbf{P}\}$ be a set of n other values. For each $p \in \mathbf{P}$, let $\Phi_{p,1}$ be an instance of *swap* with input u_p and, for $i > 1$, let $\Phi_{p,i}$ be an instance of *swap* whose input is the value returned to p by $\Phi_{p,i-1}$. Then, in every history $H \in I(\{\Phi_p \mid p \in \mathbf{P}\})$, every operation instance changes the value of the swap object.

Let $H_0 \circ \Phi_{p,i} \circ H_1 \in I(\{\Phi_p \mid p \in \mathbf{P}\})$ be a history such that H_1 is p-free. Let H_2 consist of a single instance of *swap* by a process other than p and let Φ' be the first operation instance of $H_1 \circ H_2$. Then $res(H_0 \circ \Phi_{p,i} \circ H_1 \circ H_2, \Phi') \neq res(H_0 \circ H_1 \circ H_2, \Phi')$.

Definition 4 can be further generalized so that, whenever a process performs sufficiently many consecutive operation instances of its witness sequence, at least one of these operation instances changes the result of some subsequent operation instance performed by another process. The proofs of the results in Sections 5 and 6 hold for this more general definition.

A swap object is historyless, hence it is not perturbable [9]. An object that supports only *compare-and-swap* and has at least n different values is an example of a perturbable object [9] that is not visible, because it is conditional. Thus the classes of visible and perturbable objects are incomparable.

Finally, we prove an important relationship between visible objects and visible events: In any wait-free implementation of a visible object, a process must apply visible events in the course of performing the operation instances of its witness sequence.

Lemma 10 *Consider a wait-free implementation of a visible object with witness sequence Φ_p, for each $p \in \boldsymbol{P}$, and let E be a finite execution in which every process performs a prefix of its witness sequence. If $\Phi_{p,i}$ is the last operation instance completed by p in E and p is idle at the end of E, then, while p performs $\Phi_{p,i}$, it applies an event that is visible in E.*

Proof Let $H \in I(\{\Phi_p | p \in P\})$ be the history from which E was obtained. Since $\Phi_{p,i}$ is p's last operation instance in H, this history can be written as $H_0 \circ \Phi_{p,i} \circ H_1$, where H_1 is p-free. Then Definition 4 implies that there is a p-free extension H_2 of H and an operation instance $\Phi' \in H_1 \circ H_2$ so that $res(H_0 \circ \Phi_{p,i} \circ H_1 \circ H_2, \Phi') \neq res(H_0 \circ H_1 \circ H_2, \Phi')$.

Let E' be any extension of E that results from executing H_2. Then execution $E \circ E'$ can be written as $E_0 \circ e_1 \circ E_1 \circ \cdots \circ e_k \circ E_k$, where $e_1, \ldots, e_k$ is the sequence of events applied by p while performing the operation instance $\Phi_{p,i}$ and $E_1, \ldots, E_k$ are p-free.

To obtain a contradiction, suppose that all of the events $e_1, \ldots, e_k$ are invisible in E. Then, by Proposition 1, they are invisible in $E \circ E'$. Hence, by Lemma 1, $E_0 \circ e_1 \circ E_1 \circ \cdots \circ e_k \circ E_k$ and $E_0 \circ E_1 \circ \cdots \circ E_k$ are indistinguishable to all processes in $\mathbf{P} - \{p\}$. This implies $res(E_0 \circ e_1 \circ E_1 \circ \cdots \circ e_k \circ E_k, \Phi') = res(E_0 \circ E_1 \circ \cdots \circ E_k, \Phi')$, which is impossible, since $res(H_0 \circ \Phi_{p,i} \circ H_1 \circ H_2, \Phi') \neq res(H_0 \circ H_1 \circ H_2, \Phi')$.

5 A Space Lower Bound for Visible Objects

In this section, we prove a linear space lower bound for visible objects. To do this, we define the concept of a *levelled* execution. At the end of such an execution, every process has an enabled RMW event that is assigned a distinct level. These events have the property that no event can be made invisible by events at higher levels. Then, we prove that any wait-free implementations of a visible object shared by n processes has a levelled execution. Finally, we show that any implementation that uses only write-conditional base objects and has a levelled execution must use $\Omega(n)$ base objects.

Definition 5 A finite execution E is *levelled* if there is a sequence $e_1, e_2, \ldots, e_n$ of RMW events that are enabled at the end of E such that, each is by a different process in $\mathbf{P}$ and, for every nonempty execution fragment E' consisting of some subset of these events (in any order), e_j is visible in $E \circ E'$, where $j = \min\{i \mid e_i \in E'\}$. We call $e_1, e_2, \ldots, e_n$ a levelled sequence and say that event e_j is at level j.

Lemma 11 *A wait-free implementation of a visible object shared by n processes has a levelled execution.*

Proof Consider a wait-free implementation of a visible object with a set S of witness sequences. We construct a levelled execution $E_0 \circ E_1 \circ \cdots \circ E_n$ that results from a history in $I(S)$, i.e. in which each process performs a prefix of its witness sequence, in order, on the object. This execution is constructed inductively, so that, at the end of $E_0 \circ E_1 \circ \cdots \circ E_i$, processes $p_{i+1}, \ldots, p_n$ are idle, processes $p_1, \ldots, p_i$ are active, and for $j = 1, \ldots, i$, the enabled event e_j of process p_j cannot be made invisible by any sequence of events by processes $p_{j+1}, \ldots, p_n$ as they continue performing their witness sequences. Then the execution $E_0 \circ \cdots \circ E_n$ is levelled.

Let E_0 be the empty execution. Let $1 \leq i \leq n$ and suppose that execution $E_0 \circ \cdots \circ E_{i-1}$ has been constructed. We examine the successive events that process p_i applies as it performs the next operation instance of its witness sequence. Let E be the prefix of E_i constructed so far. Initially E is empty.

While there are finite sequences of events E' and E'' by processes in $\{p_{i+1}, \ldots, p_n\}$, such that the next event e applied by p_i is invisible in the execution $E_0 \circ \cdots \circ E_{i-1} \circ E \circ E' \circ e \circ E''$ and this execution is the prefix of an execution that results from a history in $I(S)$, extend E by $E' \circ e \circ E''$. Note that, if e does not change the value of the base object to which it is applied, then E' and E'' can be empty execution fragments. By Proposition 1, e is invisible in $E_0 \circ \cdots \circ E_{i-1} \circ E \circ E'''$ for every extension E'''.

Since the implementation is wait-free, p_i will eventually complete its operation instance. Then Lemma 10 implies that p_i must have applied a visible RMW event while performing this operation instance. Thus, eventually, the event e of p_i enabled at the end of $E_0 \circ \cdots \circ E_{i-1} \circ E$ cannot be made invisible by any sequence of events by processes $p_{i+1}, \ldots, p_n$ as they continue performing their witness sequences. In this case, let $e_i = e$ and let $E_i = E \circ E'$, where E' is an extension of E in which all of the active processes among $p_{i+1}, \ldots, p_n$ complete their current operation instances and no other processes take any steps. The wait-freedom of the implementation guarantees the existence of E'.

Since processes $p_1, \ldots, p_{i-1}$ are active at the end of $E_0 \circ \cdots \circ E_{i-1}$ and they apply no events in E_i, they are still active at the end of $E_0 \circ \cdots \circ E_{i-1} \circ E_i$. Furthermore, at the end of $E_0 \circ \cdots \circ E_{i-1}$, the enabled event e_j of process p_j, for $j = 1, \ldots, i-1$, cannot be made invisible by any sequence of events by processes $p_{j+1}, \ldots, p_n$ as they continue performing their witness sequences. Thus, the same is true at the end of $E_0 \circ \cdots \circ E_{i-1} \circ E_i$.

Next, we prove that any implementation which has a levelled execution and uses base objects which support only *write* and conditional primitives, uses $\Omega(n)$ objects. The proof relies on the following observation.

Lemma 12 *Suppose e and e' are write or conditional RMW events by different processes that access the same object r. If both are enabled at the end of execution E, e is visible in $E \circ e \circ e'$ and $E \circ e' \circ e$, and e' is visible in $E \circ e'$, then e is a write event and e' is a nontrivial conditional RMW event.*

Proof Since e is visible in $E \circ e \circ e'$, it follows that e' is not a write event. Thus it is a conditional RMW event. Note that e' is nontrivial, because e' is visible in $E \circ e'$.

Since e is visible in $E \circ e \circ e'$ and $E \circ e' \circ e$, we also know that $val(E \circ e, r) \neq val(E, r)$ and $val(E \circ e' \circ e, r) \neq val(E \circ e', r)$. If e is a conditional RMW event, then it has a unique change point, so $val(E \circ e', r) = val(E, r)$. This is a contradiction. Therefore e is a write event.

Theorem 1 *The number of base objects used in an n-process, wait-free implementation of a visible object is at least n if the implementation uses only registers and conditional objects, and at least $\lceil \frac{n}{2} \rceil$ if the implementation uses only write-conditional objects.*

Proof By Lemma 11, any n-process, wait-free implementation of a visible object has a levelled execution, E, with a levelled sequence, $e_1, \ldots, e_n$. Suppose $i < j$. Then e_i is visible in $E \circ e_i \circ e_j$ and $E \circ e_j \circ e_i$, and e_j is visible in $E \circ e_j$. If events e_i and e_j both access the same object r, then, by Lemma 12, e_i is a write event and e_j is a nontrivial conditional RMW event. Thus, each object is accessed by at most two of these events. Hence, there are at least $\lceil \frac{n}{2} \rceil$ base objects.

If only registers and conditional objects are used, then each object is accessed by at most one of these events, so there are at least n base objects.

6 A Space Lower Bound for Starvation-free Mutual Exclusion

In this section, we prove a similar linear lower bound on space for starvation-free mutual exclusion. We model this problem as the implementation of an object that supports two operations, *enter* and *exit*. When a process performs enter, the object (eventually) responds with *crit*. We say that a process is in the *critical section* from the time that it receives a *crit* response from the object until it performs *exit*. When a process performs *exit*, the object (eventually) responds with *rem*. A correct object ensures that at most one process is in the critical section at any point in time.

While a process is in the critical section, it cannot perform *enter*. Moreover, a process can perform *exit* only while it is in the critical section. Formally, when a process is in the critical section, it has the first event of an instance of *exit* enabled. When a process is not in the critical section, nor performing an instance of *enter* or *exit*, either it has the first event of an instance of *enter* enabled or it has no events enabled.

We consider executions where each process alternately performs instances of *enter* and *exit*, starting with *enter*. It is assumed that no process halts while it is performing an instance of *enter* or *exit* or while it is in the critical section. This can be formalized by restricting attention to *fair executions*. These are executions E such that, for each process p, either p applies infinitely many events in E, or E has infinitely many prefixes, at the end of which, p has no enabled events.

The following lemma shows that a process must apply a visible event when it performs *enter*.

Lemma 13 *Let E be a fair execution of an implementation of starvation-free mutual exclusion. Then, while a process performs an instance of* enter*, it must apply a visible event in E.*

Proof To obtain a contradiction, suppose that process p applies no visible event in E while performing some instance Φ of *enter*. Let E_1 be the shortest prefix of E in which Φ completes and all the events that p applies while performing Φ are invisible. Then E_1 contains no events by process p after it completes Φ and p is in the critical section at the end of E_1. Let E_1' be the execution fragment obtained from E_1 by removing all events applied by p while it is performing Φ. Since all these events are invisible in E_1, Lemma 1 implies that E_1' is an execution which is indistinguishable from E_1 to all processes in $\mathbf{P} - \{p\}$.

Consider any p-free extension E_2 of E_1' such that some process $q \neq p$ is in the critical section at the end of $E_1' \circ E_2$. Since every operation instance performed in a fair execution eventually completes, such an execution $E_1' \circ E_2$ exists. By Proposition 3, $E_1' \circ E_2$ and $E_1 \circ E_2$ are indistinguishable to all processes in $\mathbf{P} - \{p\}$, so process q is also in the critical section at the end of $E_1 \circ E_2$. But p is in the critical section at the end of E_1 and E_2 is p-free, so p is also in the critical section at the end of $E_1 \circ E_2$. This contradicts the correctness of the implementation.

Like the linear space lower bound for the implementation of a visible object in Section 5, a linear space lower bound for starvation-free mutual exclusion can be obtained by demonstrating the existence of a levelled execution. The proof of this result is very similar to the proof of Lemma 11. One difference is that Lemma 13 is used instead of Lemma 10. The other difference is that wait-freedom is replaced by the assumption that the mutual exclusion object eventually gives a response to each *enter* and *exit* operation.

Lemma 14 *Any implementation of starvation-free mutual exclusion has a levelled execution.*

The proof of the lower bound for starvation-free mutual exclusion is the same as the proof of Theorem 1, except that Lemma 14 is used instead of Lemma 11.

Theorem 2 *The number of base objects used in an n-process implementation of starvation-free mutual exclusion is at least n if the implementation uses only registers and conditional objects, and at least $\lceil \frac{n}{2} \rceil$ if the implementation uses only write-conditional objects.*

Starvation-free mutual exclusion has a much more space efficient implementation if other primitives are available. For example, it can be implemented using a modulo-n counter and a register with the same set of values. The idea is that a process can perform *enter* by first applying *fetch-and-increment-mod-n* to the modulo-n counter. Then it repeatedly reads the register until the response it received from the modulo-n counter appears there. At this point, the process enters the critical section. To perform *exit*, the process writes the next value modulo n into the register.

The modulo-n counter ensures that all processes that have performed at least one step of some instance of *enter*, but have not yet completed this instance, or are in the critical section have (last received) different values. Thus, only one of these processes, the process that has the value currently in the register, can be in the critical section. Provided the register and the modulo-n counter have the same initial value, processes enter the critical section in the same order in which they apply the *fetch-and-increment-mod-n* primitive.

7 Time-Space Tradeoffs for One-Time-Visible Objects

Even when there is a linear lower bound on the space complexity of implementing an object, it might be possible to implement the object using far fewer base objects, under the restriction that each process can perform at most one operation instance. For example, our lower bound in Section 5 implies that $\Omega(n)$ objects that support only *compare-and-swap* are required for a wait-free implementation of a counter. However, a one-time counter has a wait-free implementation from a single object that supports *compare-and-swap* and has value set $\{0, 1, \ldots, n\}$. To perform *fetch-and-increment*, a process applies $compare-and-swap(0,1)$, $compare-and-swap(1,2)$, ..., $compare-and-swap(n-1,n)$, in order, until one of them returns true.

Since a one-time counter is a one-time-visible object, a space lower bound for one-time-visible objects analogous to Theorem 1 does not exist. Instead, in this section, we provide time-space tradeoffs for wait-free implementations of one-time-visible objects that use only write-conditional base objects.

The following result is analogous to Lemma 10 for visible objects. We restrict attention to histories of length $n-1$ so there can be a subsequent operation instance whose response can be affected by the last change to the value of the object.

Lemma 15 *Consider an n-process, wait-free implementation of a one-time-visible object with witness Φ_p for $p \in \mathbf{P}$. Let E be an execution of a history that consists of $n-1$ different witnesses, each of which completes. Then each process that performs its witness, applies an event that is visible in E.*

Proof Let $H_1 \circ \Phi_p \circ H_2$ be the history from which E was obtained. By Definition 3, there is an operation instance Φ' by a process q so that $H_1 \circ \Phi_p \circ H_2$ is q-free and $res(H_1 \circ \Phi_p \circ H_2 \circ \Phi', \Phi') \neq res(H_1 \circ H_2 \circ \Phi', \Phi')$. Let E' be the extension of E that results when q performs Φ'. Then execution $E \circ E'$ can be written as $E_0 \circ e_1 \circ E_1 \circ \cdots \circ e_k \circ E_k$, where $e_1, \ldots, e_k$ is the sequence of events applied by p (while performing Φ_p) and $E_1, \ldots, E_k$ are p-free.

To obtain a contradiction, suppose that all of the events $e_1, \ldots, e_k$ are invisible in E. Then, by Proposition 1, they are invisible in $E \circ E'$. Hence, by Lemma 1, $E_0 \circ e_1 \circ E_1 \circ \cdots \circ e_k \circ E_k$ and $E_0 \circ E_1 \circ \cdots \circ E_k$ are indistinguishable to process $q \in \mathbf{P} - \{p\}$. This implies $res(E_0 \circ e_1 \circ E_1 \circ \cdots \circ e_k \circ E_k, \Phi') = res(E_0 \circ E_1 \circ \cdots \circ E_k, \Phi')$, which is impossible, since $res(H_1 \circ \Phi_p \circ H_2 \circ \Phi', \Phi') \neq res(H_1 \circ H_2 \circ \Phi', \Phi')$.

The next lemma shows that when multiple conditional events and write events that access the same object are simultaneously enabled, most of them can be made invisible.

Lemma 16 *Any set of write or conditional RMW events that access the same object can be scheduled so that at most one of them is visible.*

Proof Suppose $e_1, \ldots, e_k$ are write or conditional RMW events that access the same object r and are enabled at the end of some finite execution E. Let E' be the execution fragment that consists of the events $e_1, \ldots, e_k$ in the following order. All the events e_i that are invisible in $E \circ e_i$ are scheduled first. These are the write events with argument $val(E, r)$ and the conditional RMW events for which $val(E, r)$ is a fixed point. Next the remaining write events are scheduled. Finally, the remaining conditional RMW events are scheduled.

All of the events in the first part are invisible in $E \circ E'$ since they don't change the value of r. If the second part of E' is not empty, all of its write events, except for the last, are invisible in $E \circ E'$, because each is immediately followed by a write event accessing the same object. In this case, none of the events in the third part of E' is visible in $E \circ E'$, as the value of r is a fixed point of these events. If the second part of E' is empty, then only the first event of the third part of E', if it exists, may be visible in $E \circ E'$, as it changes the value of r to a fixed point of all subsequent events in the third part of E'.

The following theorem states a tradeoff between the number of write-conditional base objects used by an implementation and the worst-case total number of nontrivial events applied by processes in an execution of the protocol.

Theorem 3 *Any wait-free n-process implementation of a one-time-visible object from $m < n$ write-conditional base objects has an execution in which the total number of nontrivial events applied by all processes is in $\Omega(n^2/m)$.*

Proof We construct an execution $E_0 \circ E_1 \circ \cdots \circ E_{\lceil (n-1)/m \rceil}$ from any history consisting of $n-1$ witnesses, performed by different processes, each of which completes. This execution is constructed inductively: Execution $E_0 \circ E_1 \circ \cdots \circ E_i$ contains at least $i(n-1-m(i-1)/2)$ nontrivial events, at most $m \cdot i$ processes complete their operation instances, and none of the at least $n-1-m \cdot i$ processes active at the end perform a visible event in this execution.

Let E_0 denote the execution in which $n-1$ processes become active, but do not perform any events. Let $1 \leq i \leq \lceil (n-1)/m \rceil$ and suppose that execution $E_0 \circ E_1 \circ \cdots \circ E_{i-1}$ has been constructed. Let each process that is active at the end of this execution continue to apply events to perform its witness until it is about to apply a nontrivial RMW event to one of the m write-conditional base objects. This must eventually happen, since, by Lemma 15, the process must eventually apply a visible event, and all trivial RMW events are invisible. By Lemma 16, all these enabled events can be scheduled so at most one event applied to each base object is visible. After all of these events are applied, we let the at most m processes that applied visible events run until their witnesses are complete. This concludes E_i.

None of the remaining processes have yet applied a visible event, so they are still active at the end of $E_0 \circ E_1 \circ \cdots \circ E_{i-1} \circ E_i$. There are at least $n-1-m(i-1)-m = n-1-m \cdot i$ such processes.

Since E_{i-1} contains at least $(i-1)(n-1-m(i-2)/2)$ nontrivial RMW events and each of the processes active at the end of $E_0 \circ E_1 \circ \cdots \circ E_{i-1}$ applies at least one nontrivial RMW event in E_i, it follows that $E_0 \circ E_1 \circ \cdots \circ E_{i-1} \circ E_i$ contains at least $(i-1)(n-1-m(i-2)/2) + n-1-m(i-1) = i(n-1-m(i-1)/2)$ nontrivial RMW events.

Let E be obtained from $E_0 \circ E_1 \circ \cdots \circ E_{\lceil (n-1)/m \rceil}$ by letting each process active at the end run until its witness is complete. Then the total number of nontrivial RMW events in execution E is at least $\lceil (n-1)/m \rceil (n-1-m(\lceil (n-1)/m \rceil - 1)/2) \in \Omega(n^2/m)$.

Corollary 1 *Any wait-free n-process implementation of a one-time-visible object from write-conditional base objects either uses at least $\sqrt{n}$ base objects or has an execution in which the average number of nontrivial events each process applies is in $\Omega(\sqrt{n})$.*

The time-space tradeoffs in Theorem 3 and Corollary 1 can be strengthened by counting the number of *memory stalls* caused by contention when multiple processes simultaneously apply nontrivial RMW events to the same object. When this happens, the events are serialized and all events, except the first, incur memory stalls.

Definition 6 Let E be an execution and let $e_0, \ldots, e_l$ be a maximal sequence of two or more consecutive nontrivial events in E by different processes that access the same object r. Then e_j *incurs j memory stalls* in E.

The concept of memory stalls was introduced by Dwork, Herlihy, and Waarts [5]. Unlike us, they count all events, not just nontrivial events. We obtain tradeoffs between the space complexity and the worst case number of memory stalls incurred in an execution for wait-free implementations of one-time-visible objects from write-conditional base objects. These results immediately imply the same results for their definition.

Theorem 4 *Any wait-free n-process implementation of a one-time-visible object from $m < n$ write-conditional base objects has an execution in which the total number of memory stalls incurred by all events is in $\Omega(n^3/m^2)$.*

Proof Consider the execution E constructed in the proof of Theorem 3 with one added requirement: when the at least $n-1-m(i-1)$ enabled nontrivial RMW events are scheduled in E_i, all the events that access the same object are applied consecutively. If r_j of these events access the j'th object, for $j=1,\ldots,m$, then the total number of stalls incurred by these events is $\sum_{j=1}^{m} r_j(r_j-1)/2$. By convexity, this sum achieves its minimum value, $m[(n-1)/m-i+1][(n-1)/m-i]/2$, when $r_1=\cdots=r_m=(n-1)/m-i+1$. Hence the total number of memory stalls incurred by all events in E is at least

$$\sum_{i=1}^{\lceil (n-1)/m \rceil} m[(n-1)/m-i+1][(n-1)/m-i]/2 \in \Omega(n^3/m^2).$$

Corollary 2 *Any wait-free n-process implementation of a one-time-visible object from write-conditional base objects either uses at least $n^{2/3}$ base objects or has an execution in which the total number of memory stalls incurred by all events is in $\Omega(n^{5/3})$.*

8 Conclusions

Many conditional and write-conditional objects are universal, so they can be used to deterministically implement any object shared by any number of processes in a wait-free manner. However, we have proved that any deterministic wait-free implementation of a visible object or starvation-free mutual exclusion from conditional and write-conditional base objects requires $\Omega(n)$ base objects when shared by n processes. So, these implementations cannot be space efficient. This implies that more than just conditional and write-conditional primitives should be provided in the design of asynchronous shared memory systems.

Our techniques can also be applied to prove lower bounds for implementations of visible objects and wait-free mutual exclusion using certain other primitives, such as load-linked and store-conditional. Further work is needed to extend the applicability of these techniques and to understand their limitations. More generally, it is a challenging open question to determine what primitives should be provided to allow time and space efficient implementations of any object.

Acknowledgements This work was supported by the Natural Sciences and Engineering Research Council of Canada and Sun Microsystems.

References

1. Anderson, J.H., Kim, Y.J.: An improved lower bound for the time complexity of mutual exclusion. Distributed Computing **15**(4), 221–253 (2002)
2. Anderson, J.H., Kim, Y.J., Herman, T.: Shared-memory mutual exclusion: major research trends since 1986. Distributed Computing **16**(2-3), 75–110 (2003)
3. Burns, J.E., Lynch, N.A.: Bounds on shared memory for mutual exclusion. Information and Computation **107**(2), 171–184 (1993)
4. Cypher, R.: The communication requirements of mutual exclusion. In: Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 147–156 (1995)
5. Dwork, C., Herlihy, M., Waarts, O.: Contention in shared memory algorithms. Journal of the ACM **44**(6), 779–805 (1997)
6. Fich, F., Ruppert, E.: Hundreds of impossibility results for distributed computing. Distributed Computing **16**(2-3), 121–163 (2003)
7. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems **13**(1), 124–149 (1991)
8. Jayanti, P.: A time complexity lower bound for randomized implementations of some shared objects. In: Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing, pp. 201–210 (1998)
9. Jayanti, P., Tan, K., Toueg, S.: Time and space lower bounds for non-blocking implementations. Siam J. Comput. **30**(2), 438–456 (2000)
10. Yang, J.H., Anderson, J.H.: A fast, scalable mutual exclusion algorithm. Distributed Computing **9**(1), 51–60 (1995)