# On the Inherent Weakness of Conditional Synchronization Primitives

Faith Fich
*University of Toronto*

Danny Hendler
*Tel-Aviv University Laboratories*

Nir Shavit
*Tel-Aviv University and Sun Microsystems Laboratories*

---

The "wait-free hierarchy" classifies multiprocessor synchronization primitives according to their power to solve consensus. The classification is based on assigning a number $n$ to each synchronization primitive, where $n$ is the maximal number of processes for which deterministic wait-free consensus can be solved using instances of the primitive and *read write* registers. Conditional synchronization primitives, such as *compare-and-swap* and *load-linked/store-conditional*, can implement deterministic wait-free consensus for any number of processes (they have consensus number $\infty$), and are thus considered to be among the strongest synchronization primitives. To some extent because of that, compare-and-swap and load-linked/store-conditional have became the synchronization primitives of choice, and have been implemented in hardware in many multiprocessor architectures.

This paper shows that, though they are strong in the context of consensus, conditional synchronization primitives are not efficient in terms of memory space for implementing many key objects. Our results hold for starvation-free implementations of mutual exclusion, and for wait-free implementations of a large class of concurrent objects, that we call *Visible(n)*. Roughly, *Visible(n)* is a class that includes all objects that support some operation that must perform a "visible" write before it terminates. *Visible(n)* includes many useful objects; some examples are: counters, stacks, queues, swap, fetch-and-add, and single-writer snapshot objects. We show that at least $n$ conditional registers are required by any such implementation, even if registers are of unbounded size. We also obtain trade-offs between time and space for $n$-process wait-free implementations of any *one-time* object in *Visible(n)*. All these results hold for both deterministic and randomized implementations.

Starvation-free mutual exclusion and wait-free implementations of some objects in *Visible(n)* (e.g. counters, swap and fetch-and-add) can be implemented by $O(1)$ non-conditional primitives. Thus we believe that basing multiprocessor synchronization solely on conditional synchronization primitives might not be the best design choice.

---

## 1. INTRODUCTION

A *wait-free* implementation of a concurrent object is one that guarantees that any process can complete an operation in a finite number of its own steps. Wait-freedom implies *lock-freedom*, thus it also provides for better robustness, as processes coordinate without using locks, and so the inherent problems of locks, such as deadlock, convoying, and priority-inversion, are avoided. In 1991, Herlihy introduced his influential "wait-free hierarchy" [9], which classifies multiprocessor synchronization primitives according to their power to solve consensus. The classification is based on assigning a number $n$ to each synchronization primitive, where $n$ is the maximal number of processes for which deterministic wait-free consensus can be solved using instances of the primitive and read write registers. The strongest synchronization primitives according to this classification are those that can solve consensus for *any* number of processes, and hence have a consensus number of $\infty$.

Synchronization primitives such as *compare-and-swap* (denoted *CAS*) and *load-linked/store-conditional* (denoted *LL/SC*) have consensus number $\infty$ and are thus among the strongest primitives according to Herlihy's hierarchy; to some extent because of that, they have became the synchronization primitives of choice, and have been implemented in hardware in many multiprocessor architectures. (For example, *CAS* has been implemented in Motorola 680x0, IBM 370 and *SPARC*® architectures; *LL/SC* has been implemented in MIPS®, PowerPC® and DECAlpha® architectures.) *CAS* and *LL/SC* belong to a class of primitives which is called by the slightly misleading name *conditional* synchronization primitives (or simply *conditionals*). Conditionals are primitive operations that modify the value of the register $r$ on which they operate only if its value, just before the operation starts, is a *specific* value, $v_w$, that depends on the input, $w$, of the conditional operation. As an example, if some process issues a *CAS(addr, old, new)* operation, the operation changes the contents of the register *addr* to *new* only if *addr*'s value just before the operation starts equals *old*; otherwise, the operation is not visible to other processes (except that it might have an adverse effect on memory contention and process latency).

Several researchers have previously obtained results suggesting that conditionals are no stronger than read write registers in the context of mutual exclusion. Cypher [6] has obtained a lower bound of $\Omega(\log \log n / \log \log \log n)$ remote memory references for $n$-process mutual exclusion based on read write registers and registers that support conditional operations. This bound was later improved by Anderson and Kim [1] to $\Omega(\log n / \log \log n)$ remote memory references. In their survey of shared-memory mutual exclusion [2], Anderson and Kim comment on the "...unexpected weakness" of conditionals in general, and *CAS* in particular, which is "...still widely regarded as being the most useful of all primitives to provide in hardware". Our work shows that conditionals are relatively weak not just in the context of mutual exclusion time-complexity. Rather, we show that conditionals are inefficient in terms of memory space for implementing many widely used distributed objects. Collectively, these results imply that basing multiprocessor synchronization solely on conditional synchronization primitives might not be the best design choice.

In the article that introduced *covering* arguments [5], Burns and Lynch obtain a lower bound of $n$ on the number of read write registers required to solve mutual exclusion. In the context of mutual exclusion, our work extends their work for *starvation-free* mutual exclusion to obtain bounds for implementations that use conditionals. To do this, we use generalized covering arguments that can be applied

to conditionals. Some of the ideas that we use in our proofs for long-lived objects are also similar to those used in their lower bound proof.

Fich, Herlihy, and Shavit, [8] obtain a lower bound of $\Omega(\sqrt{n})$ on the space complexity of randomized wait-free implementations of $n$-process consensus that use *historyless* objects. These are objects, such as read write registers, *test-and-set*, or *swap*, whose state depends only on the last nontrivial operation (for example, write) that was applied to them. Jayanti, Tan, and Toueg [12] show time and space lower bounds of $n-1$ for lock-free implementations of several objects (e.g. counters and $k$-valued $CAS$) from historyless objects and resettable consensus. Note that test-and-set is a historyless conditional.

A *long-lived* object is one whose operations may be called by a process any number of times. In contrast, the operations of *one-time* objects may be called by a process only once. For one-time objects, we show trade-offs between memory space and the number of writes, and between memory space and the number of *memory stalls* incurred by high-level operations. The concept of memory stalls was defined by Dwork, Herlihy, and Waarts, in a paper [7] that introduced a formal model to capture the phenomenon of memory contention in shared memory multiprocessors.

### 1.1 Our Results

Our results apply to implementations of any object that supports some operation that must perform a "visible" write before it terminates. This is a large class of concurrent objects that includes well-known objects such as counters (both linearizable and non-linearizable), stacks, queues, swap, and single-writer snapshot. For all such long lived objects, we show the following.

—Any $n$-process wait-free implementation that uses either only read write registers or registers that support only conditional operations, requires at least $n$ registers;

—Any $n$-process wait-free implementation that uses registers on which both read write *and* conditional operations can be applied, requires at least $\lceil \frac{n}{2} \rceil$ such registers.

For all such one-time objects, we show the following trade-offs for any wait-free implementation that uses $m < n$ registers that support only read, write and/or conditional operations.

—Either $m > \sqrt{n}$, or the amortized number of writes performed by high-level operations is in $\Omega(\sqrt{n})$;

—Either $m > n^{\frac{2}{3}}$, or the amortized number of memory stalls incurred by high-level operations is in $\Omega(n^{\frac{2}{3}})$.

All of the above results apply also to starvation-free implementations of mutual exclusion. These results apply to both deterministic and randomized implementations, even if they use registers of unbounded size.

The rest of the paper is organized as follows. In section 2, we present our model of shared-memory systems, and provide a formal definition of conditional primitives. In Section 3 we prove space lower bounds for long-lived objects. In Section 4 we prove time/space trade-offs for implementations of one-time objects. In Section 5 we generalize our definitions to multi-field objects, thus proving that our results hold also for the load-linked/store-conditional operations pair. Finally, Section 6 concludes with a discussion of the results.

## 2. PRELIMINARIES

### 2.1 Shared Memory System Model

Our model of an asynchronous shared memory system is largely based on the model described by Cypher in [6], which is based, in turn, on the model given by Merritt and Taubenfeld [13]. Shared objects are specified as in [9]. An object is an abstract data type. It is characterized by a set of possible values and by a set of *operations* that provide the only means to manipulate it. A shared memory protocol provides a specific data-representation for the object, and a specific implementation for each of its operations. An $n$-process shared memory protocol consists of a non-empty set $\mathbf{E}$ of executions, a set $\mathbf{P}$ of $n$ processes, a set $\mathbf{R}$ of shared memory registers, and a function $\mathbf{I}$ that assigns an initial value to each register in $\mathbf{R}$. Any register may support one or more operations including Read, Write, and various types of atomic Read-Modify-Write operations. No bound is assumed on register size (i.e. the number of different possible values the register can have). An *execution fragment* is a sequence (either finite or infinite) of *event*s, where an event is an operation that a specific process applies to a specific register. An *execution* is an execution fragment that starts from the initial state. An event $e$ can have one of the following three forms:

—Read($p$,$r$): indicates that process $p$ atomically reads the value of register $r$;

—Write($p$,$r$,$w$): indicates that process $p$ atomically writes the value $w$ to register $r$;

—RMW($p$,$r$,$w$,$g$,$h$): indicates that process $p$ atomically performs the code shown in Figure 1. The write function, $g$, and the return-value function, $h$, both receive two parameters, $v$, the value of register $r$ when event $e$ starts to execute, and $w$, the input value to the RMW event, if any. The type of a RMW operation is specified by the ordered pair $<g,h>$ consisting of its write and return-value functions. This definition of a RMW operation is a generalization of the definition of comparison primitives that appears in Section 5 of [1].

```
RMW(p,r,w,g,h)
     v=r.Read()
     r.Write(g(v,w))
     return h(v,w)
```

Fig. 1. The semantics of a Read-Modify-Write event.

*CAS* and *test-and-set* are two examples of RMW operations. Their write and return-value functions are illustrated in Figures 2 and 3, respectively.

```
CAS.g(v,<old,new>)
      if (v=old)
          return new
      else
          return v
```

```
CAS.h(v,<old,new>)
      if (v=old)
         return true
      else
         return false
```

Fig. 2. The write and return-value functions of the *CAS* operation with the sequence of input parameters $w = <old,new>$.

```
test-and-set.g(v,null)
      if (v=0)
          return 1
      else
          return 0
```

```
test-and-set.h(v,null)
      if (v=0)
         return true
      else
         return false
```

Fig. 3. The write and return-value functions of the *test-and-set* operation. The test-and-set operation does not receive input parameters. Thus, the second parameter passed to its write and return-value functions is the special value *null*.

We now define the class of conditional RMW primitive operations we investigate in this paper.

DEFINITION 1. *Let Op =<g,h> be a RMW operation. We say that Op is a* conditional *operation if, for every possible input value $w$, the following holds:*

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

*Let Op =<g,h> be a conditional operation and let $e = RMW(p,r,w,g,h)$ be an event of type Op. We call the single value $v_w \neq g(v_w,w)$ the* change-point *of $e$. We call any other value $v$, $v \neq v_w$, a* fixed-point *of $e$.*

In other words, a RMW operation is conditional if, for every input $w$, there is exactly one value $v_w$ such that $g(v_w,w) \neq v_w$. If $g$ and $w$ are such that $g(v,w) = v$ for all values of $v$, then the event RMW(p,r,w,g,h) is essentially an event that never writes. For any event $e = RMW(p,r,w,g,h)$, the value $v_w$ is the only value of $r$ that will be changed by $e$. We observe that *CAS* and test-and-set are conditional operations by Definition 1. We note that the load-linked/store-conditional operations pair are not conditional by Definition 1. The reason is the following. The *LL/SC* operations can only be modelled by an object whose value consists of multiple fields, and the condition that determines whether the SC operation changes the object's value depends on one of these fields. We provide the formal definition of *LL/SC* and a matching generalization of Definition 1 in Section 5.

We call a register that supports only Read and Write operations a *read write register*; we call a register that supports only conditional operations a *conditional register*; we call a register that supports Read, Write, and one or more conditional operations a *read-write-conditional* register.

We denote the empty execution by $\epsilon$. Given any event $e$, we say that $e$ is a *writing event* if $e$ is a Write event or if $e$ is a *RMW* event; if $e$ is a Read event or if $e$ is a *RMW* event, we say that $read(e)$ holds.

The memory register *accessed* (read and/or written) by $e$ is denoted $mem(e)$. The process that executes $e$ is denoted $proc(e)$. For any two execution fragments $E$ and $E'$, where $E$ is finite, the execution fragment $E \circ E'$ denotes the concatenation of $E$ and $E'$. For any execution fragment $E$, we define $procs(E)$ to be the set of processes that perform some event in $E$. Let $r \in R$ be a memory register and let $E \in \mathbf{E}$ be a finite execution, then $value(E,r)$ denotes the last value written by an event in $E$ to $r$, or $\mathbf{I}(r)$ if there was no such event. Given an execution $E$ and any subset $P \subseteq \mathbf{P}$, we let $proj(E,P)$ denote only those events in $E$ that were performed by processes in $P$. If $P = \{p\}$, we also use the notation $proj(E,p)$ instead of $proj(E,\{p\})$. Let $E \in \mathbf{E}$ be a finite execution, and $e$ be an event. If $E \circ e \in \mathbf{E}$ holds, we say that *$e$ is enabled after $E$*. An execution fragment $E$ is *fair*,

if any process that has an enabled event just before $E$ starts is in $procs(E)$. An execution is fair if all of its suffixes are fair.

### 2.2 High Level Operations

The execution of a high-level object operation involves, in general, both private- and shared-memory events; in this paper we only deal with shared-memory events. Thus, we view an execution of a high-level operation $Op$ as consisting of a sequence of one or more atomic shared memory events, each of which can be either *Read*, *Write*, or *RMW*. Let $Op$ be an execution of a high-level operation performed by some process in some execution $E$. We assume processes execute at most a single high-level operation at any given time. We denote by $proc(E, Op)$ the process that executes $Op$ in $E$; we denote by $events(E, Op)$ the sequence of memory-events performed by $proc(Op)$ while executing $Op$ in $E$. Whenever $E$ is clear from the context, we simply write $proc(Op)$ and $events(Op)$. We say that a process $p$ is *active* after $E$, and write $active(E, p)$, if, after $E$, $p$ is in the middle of executing some high-level operation $Op$, i.e. $p$ performed at least one event of an instance of $Op$, but has not performed the last event of this instance of $Op$. If $p$ is not active after $E$, we say that $p$ is *idle* after $E$, and write $idle(E, p)$. We say that an execution $E$ is *quiescent* if all processes are idle after $E$. We say that an operation $Op$ *completes* in an execution $E$, and write *completes(E,Op)*, if $Op$ terminates in $E$. If *completes(E,Op)* holds, we denote by $result(E, Op)$ the value returned by $Op$ in $E$. We say that a high-level operation $Op$ is enabled after an execution $E$, and write $enabled(E, Op)$, if $proc(Op)$ has an enabled event $e$ of $Op$ after $E$. Assume $enabled(E, Op)$ holds.

For presentation simplicity, we often refer to an underlying execution as a *state*. Thus, for example, instead of saying that a high-level operation $Op$ is enabled after execution $E$, sometimes we say that after $E$ the system is in a state where $Op$ is enabled.

## 3. LONG-LIVED OBJECTS

In this section, we obtain lower bounds on the memory requirements of wait-free implementations of long-lived objects in a class of objects that we call *Visible(n)*, when implementations use registers that only support Read, Write and/or conditional operations. We also consider starvation-free implementations of mutual exclusion from such registers.

First, we define the key concepts of *invisible* and *visible* events. Informally, an invisible event is an event by some process that cannot be observed by other processes. Obviously, all Read events are invisible. An invisible Write event was termed an *obliterated* event in [5]. We extend this notion to conditional RMW events.

The following two definitions formally define the concepts of invisible RMW and Write events.

DEFINITION 2. *Let $e$ be a $RMW(p, r, w, g, h)$ event in an execution $E$, where $E = E_1 \circ e \circ E_2$. We say that $e$ is* invisible in E*, and write $invisible(E, e)$, if at least one of the following holds:*

—$g(value(E_1, r), w) = value(E_1, r)$;
—$E_2 = e' \circ E_3$, *$e'$ is a Write event and $mem(e') = r$.*

In other words, a $RMW$ event $e$ is invisible, if either the value of the register on which $e$ operates is a fixed-point of $e$, or if $e$ is immediately followed by a Write

event on the same register. We note that $e$ would also be made invisible to other processes in an execution where it is followed by a sequence of events on other registers that terminates with a Write event applied to $r$; however our proofs do not require this broader definition.

DEFINITION 3. *Let $e = Write(p, r, w)$ be an event in an execution $E$ where $E = E_1 \circ e \circ E_2 \circ e' \circ E_3$. We say that $e$ is* invisible in E*, and write invisible$(E, e)$, if the following holds:*

—*$e'$ is a Write event and $mem(e') = r$;*
—*$\forall f \in E_2 : \neg(read(f) \wedge (mem(f) = r) \wedge (proc(f)) \neq proc(e))$.*

In other words, a Write event $e$, issued by some process $p$, is invisible in an execution $E$, if it is overwritten in $E$ by another Write event before it is read by another process. Note that whereas Write events can only be made invisible by *subsequent* Write events, RMW events can be made invisible either by *preceding* writing events, or by an immediately following Write event. If $e$ is a Write or RMW event that is not invisible in $E$, we say that $e$ is a *visible* event in $E$.

We next define the class *Visible(n)*, to which our results apply.

DEFINITION 4. *An object $O$ is in* Visible(n)*, if the following holds for every implementation of $O$. Let $p$ be a process idle in some state $S$, then $p$ can apply a sequence of high-level operations on $O$, $Op_1, \cdots, Op_k$, for some $k \geq 1$, with corresponding inputs $I_1, \cdots, I_k$, so that in every execution fragment $E$ that starts in $S$, where the operations $Op_1, \cdots, Op_k$ (called with their corresponding inputs) are the first operations of $p$ that were completed, $p$ performs at least one visible event. We call this sequence of operations (with their corresponding inputs) a* sign for p in S *and denote it by* Sign(S,p).

In other words, an object $O$ is in *Visible(n)*, if any idle process $p$ can always execute some finite sequence of high-level operations on $O$, with appropriate inputs, that must perform a visible event in the course of its execution.

It is easily proven that many key concurrent objects are in *Visible(n)*: counters, stacks, queues, swap, fetch-and-add, and single-writer snapshot are some examples. As one example, we prove in the following that concurrent counting is in *Visible(n)*.

A concurrent counter object supports a single fetch-and-increment ($FAI$) operation that processes can always call. The counter values returned by different $FAI$ operations in an execution are required to be distinct natural numbers. It is also required that in any quiescent state the values returned by the counter constitute a contiguous range of natural numbers. [1]

LEMMA 1. *Concurrent counting is in* Visible(n).

PROOF. We prove that for every state $S$ and for every process $p$ such that *idle(S,p)* holds, the FAI operation (that receives no parameters) is a sign for p in $S$. Assume otherwise by way of contradiction and let $P$ be a wait-free protocol implementing a concurrent counter. Then there are a state $S$, a process $p$ that is idle in $S$, and an execution fragment $E$ that starts in $S$, so that $p$ completes an

[1] *Linearizable concurrent counters* are also required to be linearizable [10], i.e. if $FAI_i$ and $FAI_j$ are two executions of the $FAI$ operation, and $FAI_i$ entirely precedes $FAI_j$ in an execution $E$, then $result(E, FAI_i) < result(E, FAI_j)$ must hold. The proof of Lemma 1 applies for both linearizable and non-linearizable concurrent counters, thus both counter types are in *Visible(n)*.

FAI operation $F$ in $E$ without performing a visible event. We assume without loss of generality that $S \circ E$ is quiescent. (Otherwise, as $P$ is wait-free, we can extend $E$ by letting all active processes run until they finish their high-level operations.) Let $V$ be the number of $FAI$ operations completed in $S \circ E$ (including $F$). Then the counter must have returned the values $1, \ldots, V$. We now extend $S \circ E$ by a solo execution fragment, $E_1$, of another FAI operation, $F_1$, by process $q \neq p$. $F_1$ must return value $V + 1$ in $S \circ E \circ E_1$. However, by assumption, there is another execution fragment of $P$, $E'$, such that $p$ has no events in $E'$ and the following holds:

$$\forall p' \neq p : proj(S \circ E', p') = proj(S \circ E, p').$$

Consequently, there exists an execution fragment $E_2 = E' \circ E_1$ and $F_1$ returns $V + 1$ also in $S \circ E_2$. Note that in $S \circ E_2$ the counter has returned only $V$ values. It follows that $S \circ E_2$ is a quiescent state and the values returned by the counter do not constitute a contiguous range, contradicting the semantics of a counter. □

The following definition is key to our proofs.

DEFINITION 5. *Let $A = e_1, e_2, \ldots, e_k$ be a sequence of $k$ writing events by $k$ different processes, pending in some state $S$. We say that $S$ is a* k-levelled state *with $e_j$* at level j *for $j = 1, \ldots, k$, if, in $S$, events of higher levels cannot make events of lower levels invisible. More formally, for all levels $j = 1, \ldots, k$ and for every ordering $\sigma_I$ of every subsequence $I$ of $A$, $I = e_{i_1}, \ldots, e_{i_{|I|}}$, the following holds:*

$$(e_j \in I) \wedge (\forall i < j : e_i \notin I) \Longrightarrow visible(S \circ e_{\sigma(i_1)} \circ \ldots \circ e_{\sigma(i_{|I|})}, e_j)$$

*We call $A$ a* k-levelled-sequence in S.

We also need the following definition.

DEFINITION 6. *Let $O$ be an object in* Visible(n). *Let $S$ be a state and $p$ be a process such that* idle(S,p) *holds. Further assume that, starting from $S$, $p$ executes the operations of* Sign(S,p) *and let $E$ be an execution fragment (possibly empty) that starts from $S$, in which not all the operations in* Sign(S,p) *are completed. Then we denote by* read_solo(S,E,Sign(S,p)) *the execution fragment that results when we let $p$ run solo starting from state $S \circ E$, until it either completes all the operations in* Sign(S,p) *or is about to perform a writing event.*

The organization of the proofs for long-lived objects is as follows. We first prove that all wait-free implementations of objects in $Visible(n)$ and all starvation-free implementations of mutual exclusion can be brought to an $n$-levelled state. We then prove that any protocol that uses only read write registers or only conditional registers, and reaches a $k$-levelled state, uses at least $k$ such registers, and that any protocol that uses only read-write-conditionals registers, and reaches a $k$-levelled state, uses at least $\lceil \frac{k}{2} \rceil$ such registers. Combining these results, we obtain the space lower bounds for long-lived objects.

The following lemma proves that wait-free implementations of objects in $Visible(n)$ can be made to reach $n$-levelled states.

LEMMA 2. *Let $P$ be a wait-free protocol implementing some object $O \in Visible(n)$. Then $P$ can be brought to an $n$-levelled state.*

PROOF. We construct an execution $E$ incrementally, in $n$ phases. In phase $i$, we extend the execution so that $p_i$ is on the verge of performing a visible writing event that cannot be made invisible by processes $p_{i+1}, \ldots, p_n$. Figure 4 shows

```
1: E ← ϵ
2: for (i = 1; i ≤ n; i++) // Construct E in n phases.
   {
3: S ← E
   construct-read-solo:
4: E ← E ∘ read_solo(S, E, Sign(S, p_i))
5: Let e_i denote the pending writing event by p_i
6: if exist E_1, E_2, procs(E_1), procs(E_2) ⊆ {p_{i+1}, ..., p_n} s.t.
       invisible(E ∘ E_1 ∘ e_i ∘ E_2, e_i) holds
       {
7:     E ← E ∘ E_1 ∘ e_i ∘ E_2
8:     goto construct-read-solo
       }
9: else if (i < n)
10:    if any of p_{i+1}, ..., p_n are active
11:        E ← E ∘ E_3, where in E_3 all the active processes in
               p_{i+1}, ..., p_n run until they complete their operations
   }
```

Fig. 4. Pseudo-code for the construction of execution E.

pseudo-code describing the construction of $E$. We next prove that the following claims hold for each phase $i, 1 \leq i \leq n$.

(1) The construction of phase $i$ terminates.
(2) Right after the termination of phase $i$, each of the processes $p_1, \ldots, p_i$ has a pending writing event, $e_i$, that cannot be made invisible by events of processes $p_{i+1}, \ldots, p_n$.
(3) Right after the termination of phase $i$, processes $p_{i+1}, \ldots, p_n$ are idle.

The proof proceeds by induction on the phase number, $i$. Clearly, before the first phase all of the above claims hold vacuously. Let $S$ be the state right after the termination of phase $i-1$. We now assume the claims hold in $S$, and prove for phase $i$. We first show that the construction of phase $i$ terminates. By induction hypothesis, $idle(S, p_i)$ holds, and so, as $O$ is in $Visible(n)$, $p_i$ has a sign for $S$. We let $p_i$ start execute the operations in *Sign(S,$p_i$)* until it is about to write (line 4). Let $E'$ be the resulting execution fragment. From wait-freedom, $E'$ is finite. From Definition 4, $E'$ must terminate with a pending writing event, which we denote $e_i$. Next, we show that the phase construction eventually reaches line 9. There are two cases to consider:

—If there is no extension $E_1 \circ e_i \circ E_2$ of $E$, with $procs(E_1), procs(E_2) \subseteq \{p_{i+1}, \ldots, p_n\}$ that makes $e_i$ invisible, then the construction reaches line 9 immediately.
—Otherwise, there is an extension $E_1 \circ e_i \circ E_2$ (possibly with an empty $E_1$ and/or an empty $E_2$), with $procs(E_1), procs(E_2) \subseteq \{p_{i+1}, \ldots, p_n\}$, that makes $e_i$ invisible. We set $E \leftarrow E \circ E_1 \circ e_i \circ E_2$ (line 7), jump to *construct-read-solo* and let $p_i$ continue the execution of $Sign(S, p_i)$ until it is about to write again. As $P$ is wait-free, and as every jump to *construct-read-solo* implies that $p_i$ performs at least one additional event, we can only jump to *construct-read-solo* a finite number of times, after which we must reach line 9.

Having reached line 9, if $i < n$, we extend $E$ with an execution fragment $E_3$, where in $E_3$ all of the active processes in $\{p_{i+1}, \ldots, p_n\}$, if any, complete their high-level

```
Mutex()
  {
  Entry Section
  Critical Section
  Exit Section
  }
```

Fig. 5. The structure of the mutual-exclusion operation.

operations. The existence of such a finite execution fragment $E_3$ is guaranteed from the wait-freedom of $P$. Thus, claims 1 and 3 are proven for phase $i$. Claim 2 follows directly from the construction and from the induction hypothesis. Let $S$ be the state after the execution of $E$. By construction, for every $1 \leq i \leq n$, $visible(S \circ E_1 \circ e_i \circ E2, e_i)$ holds for any execution fragments $E_1$, $E_2$ consisting of any ordering of any disjoint subsets of the events $e_{i+1}, \ldots, e_n$. Thus $e_1, \ldots, e_n$ is an $n$-levelled-sequence in $S$, with $e_i$ having level $i$, and $S$ is an $n$-levelled state. □

Next we show the same result for starvation-free mutual exclusion. We model mutual exclusion as follows. A mutual exclusion protocol supports a single operation, called *Mutex*, which has the structure shown in Figure 5. The non-critical section is assumed to take place outside of *Mutex* operations. It is assumed that no process halts while executing the *Mutex* operation. The requirements from a mutual exclusion implementation are the following:

—*Exclusion*: at most one process executes its critical section at any time.

—*Starvation-freedom* (also known in the literature as *lockout-freedom*): in any fair execution, if some process is in its entry section, then that process eventually executes its critical section.

—*Finite exit*: in any fair execution, if a process is in its exit section, then it eventually exits the Mutex operation.

We need the following technical lemma.

LEMMA 3. *Let $P$ be a starvation-free mutual exclusion protocol, and let $E$ be an execution of $P$ where some process $p$ enters the critical section, then $E$ contains a visible writing event by $p$.*

PROOF. Assume otherwise by way of contradiction, then there is some execution $E$ after which $p$ is in the critical section, and $E$ does not contain any visible write by $p$. Consequently, there exists another execution, $E'$, that does not contain any events by process $p$, such that:

$$\forall p' \neq p : proj(E', p') = proj(E, p').$$

From starvation-freedom, there is some extension $E''$ of $E'$, such that some process $q \neq p$ enters the critical section in $E' \circ E''$ and $p \notin procs(E' \circ E'')$. Thus, $q$ enters the critical section also in the execution $E \circ E''$, and exclusion is violated, a contradiction. □

LEMMA 4. *Let $P$ be a long-lived starvation-free mutual exclusion protocol. Then $P$ can be brought to an $n$-levelled state.*

The proof of Lemma 4 is almost identical to that of Lemma 2. An execution that results in an $n$-levelled state is constructed in exactly the same way. The only differences between the two proofs are that here we use starvation-freedom instead of wait-freedom and we use Lemma 3 instead of membership in *Visible(n)*.

We next prove that any protocol that reaches a $k$-levelled state and uses registers that support only Read, Write and/or conditional operations, uses $\Omega(k)$ such registers.

LEMMA 5. *Assume that a protocol $P$ can be brought to a $k$-levelled state $S$ for some $k > 0$. Let $SPACE(P)$ denote the number of registers used by $P$.*

*(1) If $P$ uses only read write registers, then $SPACE(P) \geq k$.*

*(2) If $P$ uses only conditional registers, then $SPACE(P) \geq k$.*

*(3) If $P$ uses only read-write-conditional registers, then $SPACE(P) \geq \lceil \frac{k}{2} \rceil$.*

PROOF. Let $S$ be a $k$-levelled state with a $k$-levelled sequence $A = e_1 \cdots e_k$. Let $p_i = proc(e_i)$ and let $R = \{mem(e_i) | i = 1, \ldots, k\}$. Note that no two Write events can be pending on the same register because any of these two events makes the other invisible. Thus, a lower-level event can be made invisible by a higher-level event, contradicting our assumption that $A$ is a $k$-levelled sequence of $S$. This proves (1).

Next we show that no two conditional RMW events can be pending on the same register. Assume otherwise. Let $e_i = RMW(p_i, r, w_i, g_i, h_i)$ and $e_j = RMW(p_j, r, w_j, g_j, h_j)$ be two conditional RMW events pending on the same register $r \in R$. Since $A$ is a $k$-levelled sequence in $S$, both $visible(S \circ e_i, e_i)$ and $visible(S \circ e_j, e_j)$ hold. Consequently $value(S, r)$ is the only change-point of both $e_i$ and $e_j$. This implies that either of the events $e_i$ or $e_j$ can make the other invisible, contradicting our assumption that $A$ is a $k$-levelled sequence of $S$. This proves (2).

No two Write events can be pending on the same register and no two conditional RMW events can be pending on the same register. It follows that at most two events can be pending on any single register $r \in R$: one Write and one conditional RMW. This proves (3). □

Our main result follows directly from the above lemmata.

THEOREM 1. *Let $P$ be an $n$-process protocol that is a starvation-free implementation of mutual exclusion or a wait-free implementation of an object in Visible(n). Let $SPACE(P)$ denote the number of registers used by $P$.*

*—If $P$ uses only read write registers, then $SPACE(P) \geq n$.*

*—If $P$ uses only conditional registers, then $SPACE(P) \geq n$.*

*—If $P$ uses only read-write-conditional registers, then $SPACE(P) \geq \lceil \frac{n}{2} \rceil$.*

PROOF. Immediate from Lemmata 2, 4 and 5 □

## 4. ONE-TIME OBJECTS

A one-time object is an object whose operations can be called by a process only once. It is easily seen that the memory bounds proven in Section 3 for long-lived objects do not hold, in general, for one-time objects. As an example, a one-time wait-free counter shared by $n$ processes *can* be implemented by using a single register that supports the *CAS* operation. However, by Theorem 1, a long-lived wait-free

counter requires at least $n$ such registers. For one-time objects, we prove time-space trade-offs for wait-free implementations that use only read-write-conditional registers. Our proofs apply for a class of objects defined as follows.

DEFINITION 7. *An object $O$ is in $Visible'(n)$, if $O$ supports a high-level operation $Op$, that can be called by every process $p_k$ from the initial state with some input $I_k$, so that the following holds. Let $Op_k$ denote the instance of $Op$ performed by $p_k$. Then, for every wait-free protocol, $P$, implementing $O$, and in any execution $E$ of $P$ such that $completes(E, Op_k)$ holds, $events(E, Op_k)$ includes at least one visible writing event.*

In other words, an object $O$ is in *Visible'(n)*, if it supports some operation $Op$, such that starting from the initial state, if all processes execute $Op$ with appropriate inputs, no process can complete its operation before it performs a visible writing event.

It is easily seen that one-time counters, stacks, queues, swap, fetch-and-add, and single-writer snapshot objects are in $Visible'(n)$. We note that the following trade-offs also apply to starvation-free implementations of one-time mutual exclusion.

We first need the following lemma, that states that when multiple conditional events and Write events are pending on the same register, they can always be scheduled such that at most two events are visible. [2]

LEMMA 6. *Let $L = \{e_1, \ldots, e_k\}$ be a set of $k$ writing events pending on the same register $r$ in state $S$; then, if all the events are either Write or conditional events, they can be scheduled such that at most two of them are visible.*

PROOF. If all the events in $L$ are Write events, the lemma is immediate, as only the Write event scheduled last may be visible. Otherwise, there are some conditional events in $L$. Let $L_1$ contain all Write events in $L$, and let $L' = L \setminus L_1$ be the non-empty set of conditional events in $L$. Let *cur* be the value of $r$ in state $S$, and consider any event $e \in L'$, $e = RMW(p, r, w, g, h)$. As $<g,h>$ is conditional, there is only a single value $v$ such that $g(v, w) \neq v$; we denote this value by $old(e)$. We partition the events of $L'$ to the following two disjoint sets: $L' = L_2 \cup L_3$, where $L_2 = \{e_j \in L' | old(e_j) \neq cur\}$, and $L_3 = L' \setminus L_2$. In any schedule where the events of $L_2$ are performed first (in any order) followed by all the events of $L_3$ (in any order), we have at most a single visible event (namely the first event scheduled from $L_3$, if any). If $L_1$ is non-empty, we schedule the events in it last (in any order), and the last of these to be scheduled may also be visible. □

The following theorem states a trade-off between the number of read-write-conditional registers used by a protocol and the number of write events it can be made to perform.

THEOREM 2. *Let $O$ be an object in Visible'(n), and let $P$ be an $n$-process wait-free implementation of $O$ that uses only read-write-conditional registers. If $P$ uses at most $m < n$ registers, then $P$ has an execution $E$ in which the total number of writing events performed is in $\Omega(\frac{n^2}{m})$.*

PROOF. For simplicity, we assume that $2 \cdot m$ divides $n$. As $O$ is in $Visible'(n)$, the high-level operations performed by the $n$ processes cannot terminate before they perform a visible write. We construct an execution $E$ in phases. In each phase, we

[2] A claim very similar to Lemma 6 is proven in Section 3.2 of [1].

bring all processes to be on the verge of writing. As $P$ can only apply Writes and conditional RMW operations to registers, from Lemma 6, an adversarial scheduler can arrange the writing events on each register $r$ so that at most two of these events are visible in $E$ at $r$. Consequently, at most $2m$ processes perform a visible write during the phase. We let these processes run to completion. In the second phase there remain $n - 2m$ processes that did not perform a visible write. The same argument can be applied to them. We continue in this manner, until all processes have performed a visible write. Denoting the total number of writes made in $E$ by $W(P)$, we get:

$$W(P) = \sum_{k=0}^{\frac{n}{2m}} (n - 2km) \in \Omega(\frac{n^2}{m}). \tag{1}$$

□

From Theorem 2 we get the following, more specific, trade-off between space and number of write operations:

THEOREM 3. *Let $P$ be an $n$-process wait-free implementation of $O$ that uses only read-write-conditional registers and let $SPACE(P)$, $W_{amortized}(P)$ and $W_{op}(P)$, respectively, denote the number of registers used by $P$, the worst-case amortized number of writing events issued by $n$ concurrent operations, and the worst-case number of writing events issued by a single high-level operation in $P$. Then either $SPACE(P) > \sqrt{n}$ or both $W_{amortized}(P)$ and $W_{op}(P)$ are in $\Omega(\sqrt{n})$.*

The results of Theorems 2 and 3 can be strengthened, by counting not only the number of writes, but also the number of *memory stalls* caused from write contention. In all shared-memory systems, when multiple processes attempt to write to the same memory location simultaneously, the writes are serialized, and waiting operations incur memory stalls. The concept of memory stalls was introduced in [7]. We use the following definition of memory stalls, which is stricter than that of [7], as it counts only stalls caused by contention in *writing*.

DEFINITION 8. *Let $E$ be an execution and let $e_0, \ldots, e_l$ be a maximal sequence of two or more consecutive events in $E$ such that*

*(1) $e_j$ is either a* Write *or a* RMW *event, for $j \in \{0, \ldots, l\}$, and*
*(2) $proc(e_{j_1}) \neq proc(e_{j_2})$ and $mem(e_{j_1}) = mem(e_{j_2})$ for distinct $j_1$, $j_2$ in $\{0, \ldots, l\}$.*

*Let $Op$ be the high-level operation whose execution issued $e_j$. Then we say that $Op$ incurs $j$ memory stalls in $E$ on account of $e_j$.*

We obtain a trade-off between space complexity and the number of memory stalls incurred by a wait-free implementation of any object in $Visible'(n)$ that uses only read-write-conditional registers.

THEOREM 4. *Let $P$ be an $n$-process wait-free implementation of $O$ that uses only read-write-conditional registers. If $P$ uses at most $m < n$ registers, then there is an execution in which the total number of memory stalls incurred by all processes is in $\Omega(\frac{n^3}{m^2})$*

PROOF. For simplicity, we assume that $2 \cdot m$ divides $n$. We consider the same type of phased execution $E$ we constructed in Theorem 2. We denote by $ms(E)$ the

total number of memory stalls incurred by processes during the execution of $E$; we denote by $M_k(E)$, for $1 \leq k \leq n/2m$, the total number of memory stalls incurred by processes on account of their writing events in phase $k$, and for $1 \leq i \leq m$ we denote by $Writers(E, k, i)$ the number of processes that write to register $r_i$ in phase $k$ of $E$. We get:

$$M_k(E) = \sum_{i=1}^{m} \frac{Writers(E,k,i) \cdot (Writers(E,k,i) - 1)}{2} \tag{2}$$

$$\sum_{i=1}^{m} Writers(E,k,i) = n - 2 \cdot k \cdot m. \tag{3}$$

It is easily seen that $M_k(E)$ is minimized in the equations above when the writes are equally distributed among registers (i.e. every register is written by $\frac{n-2\cdot k\cdot m}{m}$ processes), thus we get:

$$M_k(E) \geq \frac{m}{2} \cdot (\frac{n-2km}{m}) \cdot (\frac{n-2km}{m} - 1). \tag{4}$$

Summing over all phases, we get:

$$\begin{aligned} ms(E) &= \textstyle\sum_{k=0}^{\frac{n}{2m}} M_k(E) \\ &\geq \tfrac{m}{2} \cdot \textstyle\sum_{k=0}^{\frac{n}{2m}} (\tfrac{n-2km}{m}) \cdot (\tfrac{n-2km}{m} - 1) \in \Omega(\tfrac{n^3}{m^2}). \end{aligned} \tag{5}$$

□

From Theorem 4 we get the following, more specific, trade-off between space and number of memory steps:

THEOREM 5. *Let $P$ be an $n$-process wait-free implementation of $O$ that uses only read-write-conditional registers. Let $SPACE(P)$, $ms_{amortized}(P)$ and $ms_{op}(P)$, respectively, denote the maximal number of registers used by $P$, the worst-case amortized number of memory stalls incurred by $n$ concurrent operations, and the worst-case number of memory stalls incurred by a single high-level operation in $P$. Then either $SPACE(P) > n^{\frac{2}{3}}$ or both $ms_{amortized}(P)$ and $ms_{op}(P)$ are in $\Omega(n^{\frac{2}{3}})$.*

## 5. EXTENDING OUR RESULTS TO LOAD-LINKED/STORE-CONDITIONAL OPERATIONS

In this section we provide a formal definition of the load-linked/store-conditional ($LL/SC$) operations. We then present a definition of conditional operations that is a generalization of Definition 1. The new definition applies to $SC$ operations, in addition to the $CAS$ and test-and-set operations. We conclude this section by proving that the results we have presented in Sections 3 and 4 hold also when protocols can apply the $LL$ and $SC$ operations in addition to $CAS$, test-and-set, Reads and Writes.

### 5.1 Load-Linked/Store-Condition Formal Definition

The $LL$ and $SC$ operations enable a process, $p$, to modify the value of a register, $r$, by using $SC$ only if no other operation has changed $r$'s value since the last time that $p$ applied $LL$ to $r$. The $LL$ and $SC$ operations provide functionality that is

```
RMW(p,r,w,g,h)
   v=r.Read()
   r.Write(g(v,w,p))
   return h(v,w,p)
```

Fig. 6. The generalzied semantics of a Read-Modify-Write event.

similar to that of *CAS*, but is somewhat stronger, as it does not suffer from the ABA problem [3] inherent to *CAS*.

*LL* and *SC* are RMW operations [11]. Unlike the *CAS* and test-and-set operations, however, the *LL* and *SC* operations have to be modelled by an object whose value consists of multiple fields. Let $o$ be such an object. We denote the finite set of fields that $o$ has by $F(o)$. The value of an object is a tuple of its field values, arranged in some arbitrary, but fixed, order. As an example, if $F(o) = \{f_1, \cdots f_k\}$ , then $o$'s value at any given time is of the form $v =< v_1, \cdots, v_k >$, where $v_i$ is the value of field $f_i$, and is denoted by $v|f_i$.

An object that supports the *LL*, *SC* operations-pair consists of $n + 1$ fields: $n$ one bit fields, $Set[p]$ for each process $p \in \mathbf{P}$, in addition to a field *val*, assuming values from some domain. In the following we call objects with such a structure *extended registers*. An *LL* operation applied by process $p$ to an extended register $o$ sets $o.Set[p] \leftarrow 1$ and returns $o.val$. When $p$ performs an *SC(w)* operation on $o$, there are two possibilities: if $o.Set[p] = 1$, the *SC* operation sets $o.Set[q] \leftarrow 0$ for all processes $q \in \mathbf{P}$, sets $o.val \leftarrow w$, and returns *true*. In this case, we say the *SC* operation was successful. Otherwise, *SC* leaves $o$'s state unchanged and returns *false*, in which case we say the *SC* operation was unsuccessful.

The above specification implies that the write and return-value functions of the *LL* and *SC* operations need to receive a third parameter: the id of the process $p$ that performs the operation. Figure 6 presents the semantics of this generalized definition of RMW operations.

Figure 7 presents the write and return-value functions of the *LL* and *SC* operations. Assume process $p$ applies the operation to an extended register $o$. The write function of the *LL* operation sets $o.Set[p]$ and does not change $o$'s other fields. The return-value function of the *LL* operation simply returns the value of $o$'s *val* field.

The write function of the *SC* operation does the following. If $p$'s *Set* field is set, it resets all of $o$'s *Set* fields, to make sure that any subsequent *SC* operation applied to $o$ that corresponds to a preceding *LL* operation would be unsuccessful, and then updates $o$'s *val* field with its input $w$. Otherwise, it does not change the value of $o$.

The value returned by the return-value function of the *SC* operation also depends on the value of $p$'s *Set* field. If the field is set, the compound value *<true, w>* is returned, indicating that the *SC* operation was successful, and that the new value of $o$'s *val* field is $w$. Otherwise, the compound value *<false, o.val>* is returned, indicating that the *SC* operation was unsuccessful and returning the current value of $o$'s *val* field.

We now describe the semantics of some other operations that may be applied to

[3] Assume a process $p$ reads a value $A$ from $r$, and then performs a successful *CAS* operation, which modifies the contents of $r$. Process $p$'s *CAS* should ideally succeed only if the contents of $r$ were not modified since it read value $A$. The ABA problem is the following: if other processes modify $r$'s value to $B$ and then back again to $A$ in the time period between the read and *CAS* performed by $p$, the *CAS* operation still succeeds.

```
LL.g(v,null,p)
       v.Set[p]=true
       return v

SC.g(v,w,p)
       if (v.Set[p]=1)
           {
           for all q in P
             v.Set[q]=0
           v.val=w
           }
       return v
```

```
LL.h(v,null,p)
   return v.val

SC.h(v,w,p)
   if (v.Set[p]=1)
       return <true, w>
   else
       return <false, v.val>
```

Fig. 7. The write and return-value functions of the *LL* and *SC* operations with input $w$. The *LL* operation does not receive an input value, thus the second parameter passed to its write and return-value functions is *null*.

an extended register $o$.

—The Read operation simply returns $o.val$.

—*CAS*, test-and-set and Write operations that are applied to $o$ maintain their regular semantics, except for the following two differences. Firstly, they operate on $o$'s *val* field. Secondly, whenever they change the value of $o$'s *val* field, they reset all of $o$'s $Set[p]$ fields.

We name extended registers that support only the following operations *conditional extended registers*: *LL*, *SC*, *CAS* and test-and-set. We name extended registers that support Reads and Writes in addition to all of the above operations *read-write-conditional extended registers*.

We now provide a definition of conditional operations that applies for extended registers.

DEFINITION 9. *Let Op =<g,h> be a RMW operation supported by an extended register o. We say that Op is a* conditional operation *on o if, for every process $p \in \boldsymbol{P}$, there exists a field of o, $f(o, Op, p) \subset F(o)$, such that whenever an operation of type Op is applied on o by p the following holds:*

*For every possible input $w$, there is a single value of $f(o, Op, p)$, $c_w$, such that:*

$$g(v, w) \neq v \iff o|f(o, Op, p) = c_w.$$

*Let Op =<g,h> be a conditional operation on o and let $e = RMW(p, r, w, g, h)$ be an event of type Op. We call the single value $c_w$ of $f(o, Op, p)$ the* change-point *of e. We call any other value $v$ of $f(o, Op, p)$, $v \neq v_w$, a* fixed-point *of e.*

The following lemma shows that *CAS*, *SC* and test-and-set are conditional operations on extended registers.

LEMMA 7. *Let o be an extended register, then* CAS, SC *and test-and-set are conditional on o.*

PROOF. An *SC* operation by $p$ on $o$ succeeds if and only if $o.Set[p] = 1$. Thus, $f(o, SC, p) = Set[p]$, and the change point is 1 . A *CAS* operation by $p$ on $o$ with input $w =<old, new>$ succeeds if and only if $o.val = old$. Thus, $f(o, CAS, p) = val$, and the change point for input $w$ is *old*. Finally, a test-and-set operation by $p$ on $o$ succeeds if and only if $o.val = 0$. Thus, $f(o, test - and - set, p) = val$, and the change point is 0. □

The next lemma shows that, similarly to conditional events on regular registers, conditional events on extended registers can also be made invisible by preceding events that change the value of the object.

LEMMA 8. *Let $o$ be an extended register, and let $e = RMW(p, r, w, g, h)$ be a CAS, SC or test-and-set event pending on $o$ in $S$, such that $value(S, o)$|f(o,<g,h>,p) is a change point of $e$. Let $e'$ be an event that changes $o.val$, then $visible(S \circ e', e)$ does not hold.*

PROOF. If $e$ is a *CAS* event with input $w$=<*old,new*>, then $value(S, o.val) = old$ and the claim is obvious. If $e$ is a test-and-set event, then $value(S, o.val) = 0$ and the claim is also obvious. Otherwise, $e$ is of type *SC*. As $e'$ changes $o.val$, $e'$ resets the $Set[p]$ field, thus the claim follows. □

The following fact follows directly from Definition 3.

FACT 1. *Let $e$, $e'$ be two Write events pending on an extended register $o$ in state $S$. Then $visible(S \circ e \circ e', e)$ does not hold.*

By using Lemma 8 and Fact 1 it is straightforward to see that Lemmata 5 and 6 hold also for the operations supported by extended registers. Thus, we get the following.

COROLLARY 1. *The results of Theorems 1 through 5 hold also for conditional and read-write-conditional extended registers.*

## 6. DISCUSSION

Conditional synchronization primitives are among the strongest primitives according to Herlihy's wait-free hierarchy, as they can be used to implement deterministic wait-free consensus for any number of processes. This paper shows that conditional synchronization primitives are relatively inefficient in terms of memory space for wait-free implementations of most non-trivial objects. Our results apply to starvation-free implementations of mutual exclusion and to wait-free implementations of objects in *Visible(n)*, a class that contains objects supporting an operation that must perform a visible write before it terminates. In contrast, starvation-free mutual exclusion and some of the key objects in *Visible(n) can* be implemented using $O(1)$ registers that support other synchronization primitives, such as fetch-and-increment.

Lower bound results for several object-classes similar to *Visible(n)* were obtained in other papers, and it is worthwhile to compare and contrast these object-classes. Attiya et al. [4] define the class of *Sensitive Objects* and prove several space lower bounds on adaptive implementations of sensitive objects from historyless objects. Intuitively, an object is sensitive if every process can always invoke a sequence of operations that must be observed by later operations. Clearly this implies that that sequence of operations must perform a visible writing event. Jayanti et al. [12] define the class of *Perturbable Types* and prove time and space lower bounds on implementations of such types. Similarly to sensitive objects, implementations of perturbable objects can also be repeatedly made to perform operations observable by other operations. Thus, for both sensitive objects and perturbable objects, executions can be constructed where all processes can repeatedly be made to perform visible writes. Consequently, a perturbable or sensitive object is in *Visible(n)*. On the other hand, there are objects in *Visible(n)* that are neither perturbable nor sensitive. This stems from the fact that the definition of the perturbable and

sensitive classes is in terms of the effect of some operations on the result of another operation, whereas the definition of *Visible(n)* is in terms of the necessity of operations to perform a visible write. Let us consider the artificial object *MustWrite*, where any operation has to perform a visible write, but operations do not return a value. *MustWrite* is clearly in *Visible(n)* but is neither perturbable nor sensitive. As it is shown in [4] that every sensitive object is also perturbable but the reverse does not hold, we get:

$$Sensitive \subsetneq Perturbable \subsetneq Visible(n) \tag{6}$$

Several researchers have previously obtained results that indicate the weakness of conditional primitives. Anderson and Kim [1], continuing the work of Cypher [6], proved an $\Omega(\log n / \log \log n)$ remote memory references lower bound on starvation-free mutual exclusion implementations that use $CAS$ and $LL/SC$, "almost" equal to the $\log n$ remote memory references algorithm of Anderson and Yang [3] that uses read-write registers. In contrast, starvation-free mutual exclusion *can* be implemented in $O(1)$ remote memory references by using registers that support fetch-and-increment. Jayanti [11] considers wait-free object implementations that use registers that support read/write and any of the following operations: $LL/SC$, *swap*, and *move*. He proves a lower bound of $\Omega(\log n)$ on the worst-case latency of any such implementation for counters, stacks and queues. This is yet another indication of the weakness of $LL/SC$ in the context of wait-free implementations.

*Multi-valued* objects are objects such that different operations (called by different processes) that are applied to them may return different, yet related, values. Consensus is the key concurrent object that is not multi-valued. The aforementioned results, viewed collectively, imply that even though $CAS$ and $LL/SC$ are strong in the context of solving deterministic wait-free consensus, they are relatively weak in the context of wait-free implementations of most multi-valued objects. The conclusion is therefore, that basing multiprocessor synchronization solely on conditional synchronization primitives is probably not a good design choice.

## REFERENCES


[1] J. Anderson and Y. Kim. An improved lower bound for the time complexity of mutual exclusion. In *ACM Symposium on Principles of Distributed Computing*, pages 90–99, 2001.

[2] J. Anderson and Y. Kim. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16(2-3):75–110, 2003.

[3] J. Anderson and J.-H. Yang. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, 1995.

[4] H. Attiya, F. E. Fich, and Y. Kaplan. Lower bounds for adaptive collect and related objects. In *Symposium on Principles of Distributed Computing (To appear)*, 2004. To Appear.

[5] J. Burns and N. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.

[6] R. Cypher. The communication requirements of mutual exclusion. In *ACM Proceedings of the Seventh Annual Symposium on Parallel Algorithms and Architectures*, pages 147–156, 1995.

[7] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. *Journal of the ACM (JACM)*, 44(6):779–805, 1997.

[8] F. E. Fich and E. Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16:121–163, 2003.

[9] M. Herlihy. Wait-free synchronization. *ACM Transactions On Programming Languages and Systems*, 13(1):123–149, Jan. 1991.

[10] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions On Programming Languages and Systems*, 12(3):463–492, July 1990.

[11] P. Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *Symposium on Principles of Distributed Computing*, pages 201–210, 1998.

[12] P. Jayanti, K. Tan, and S. Toueg. Time and space lower bounds for non-blocking implementations. *Siam J. Comput.*, 30(2):438–456, 2000.

[13] M. Merrit and G. Taubenfeld. Knowledge in shared memory systems. In *ACM Symp. on Principles of Distributed Computing*, pages 189–200, 1991.